

Université de Montpellier
Licence 2 Informatique



Intelligence Artificielle :
Puissance 4

Rapport du TER

Etudiants :
Dabachil Ali
Farah Sabrine
Garcia Léa
Harti Abdellah

Encadrant :
Mr. Pompidor Pierre

Table des matières

1 Présentation du projet	3
1.1 Introduction	3
1.2 Motivations en lien avec notre parcours scolaire	3
1.3 L'objectif final	3
1.4 Règles du jeu du puissance 4 en général	4
1.5 Règles du jeu du tic - tac - toe	4
1.6 Langage et outils utilisés	5
1.7 Notre version du puissance 4	5
2 Organisation du projet	6
2.1 Les étapes du projet	6
2.2 Fonctionnement du groupe	6
3 Algorithme mini-max	8
3.1 Première version de la fonction explore()	8
3.2 Principe du mini-max	9
3.3 L'algorithme du mini-max dans notre programme	9
3.3.1 Deuxième version de la fonction explore()	9
3.3.2 Fonction min()	10
3.3.3 Fonction max()	10
4 Explications du programme	11
4.1 Fonction "estGagnant"	11
4.1.1 Première version	11
4.1.2 Avant optimisation	11
4.1.3 Après optimisation	12
4.2 Fonctions "possibles()" et "gravite()"	13
4.3 Les autres fonctions du programme	14
4.3.1 Fonction "affiche()"	14
4.3.2 Fonction "coupHumain()"	14
4.3.3 La boucle de jeu	15
4.3.4 Les arguments attendus en ligne de commande	15
5 Perspectives avec la méthode Monte-Carlo	16
5.1 Une autre perspective d'approche : <i>La méthode de Monte-Carlo</i>	16
5.1.1 Rappels sur le minimax et choix de la perspective envisagée	16
5.1.2 Introduction de la méthode de Monte-Carlo	16
5.1.3 Une approche simplifiée	18
5.1.4 Stratégie et recherche arborescente MCTS Monte-Carlo Tree Search	18
6 Complexité des algorithmes	20
6.1 Analyse des algorithmes	20
6.2 Ordre de complexité	20
6.3 O	20

6.4	Complexité des fonctions du code	21
7	Interface Graphique	24
7.1	Présentation de l'environnement PyGame	24
7.1.1	Intérêt d'utilisation de la bibliothèque PyGame	24
7.1.2	Exemples d'utilisation	24
7.1.3	Interface graphique proposée pour le Puissance 4	26
7.1.4	Tests effectués sur une grille traditionnelle 6 * 7	29
8	Bilan de notre projet	30
9	Bibliographie	31
10	Annexe	32

Chapitre 1

Présentation du projet

1.1 Introduction

Dans le cadre du module HLIN405 : Projet informatique, du second semestre de deuxième année de licence d'informatique, nous avons choisi de travailler sur le développement d'une intelligence artificielle capable de jouer contre un humain au jeu du puissance 4.

Nous formons un groupe composé d'Ali Dabachil, Sabrine Farah, Léa Garcia et Abdellah Harti. Notre encadrant est Monsieur Pierre Pompidor, que nous remercions d'avoir su nous guider et nous aider tout au long de ce semestre.

1.2 Motivations en lien avec notre parcours scolaire

Durant cette deuxième année de licence, nous avons été amenés à étudier plusieurs langages informatiques et à les appliquer lors de séances de TD et TP. Mais jusqu'ici nous n'avions jamais eu l'occasion de mettre en pratique nos connaissances dans un but précis avec un résultat concret et abouti à la clé. Si nous avons choisi ce projet c'est d'une part pour pouvoir approfondir nos connaissances dans un langage informatique bien précis, pour pouvoir l'utiliser de manière assez régulière et nous permettre de bien le maîtriser. D'une autre part, certains d'entre nous ont pour but de continuer leurs licences par un master en Intelligence Artificielle, c'était donc un sujet particulièrement intéressant pour en savoir plus sur ce domaine de l'informatique.

1.3 L'objectif final

Le but de ce projet est d'aboutir à un programme représentant une intelligence artificielle capable de jouer face à un humain au jeu du puissance 4. Le but de cette IA est d'abord de gagner face au joueur adverse, si elle ne le peut pas elle cherchera à obtenir une égalité avec aucun gagnant et dans le pire des cas, elle perdra. L'autre joueur quant à lui, pourra indiquer à l'ordinateur où il veut placer ses pions via un terminal. Le programme affichera l'avancement du jeu avec l'état de la grille et indiquera qui remporte la partie ou s'il y a une égalité entre l'IA et l'autre joueur. Le jeu final sera évidemment soumis aux vraies règles du jeu du puissance 4. Le programme acceptera de placer les pions du joueur adverse dans la grille, si et seulement si, leurs placements respectent les règles du jeu.

Idéalement, notre projet devrait être de développer une IA qui joue au puissance 4 classique que nous connaissons, mais un objectif plus raisonnable est de la faire jouer avec la même règle de gravité mais sur des grilles plus petites. Nous avons donc en première partie du projet, avant de passer au développement du puissance 4, développé une IA capable de jouer au jeu du tic - tac - toe sur une grille 3 x 3. Le but de notre projet était également de mettre en oeuvre l'algorithme du mini-max, et découvrir l'algorithme du Monte-Carlo (MCTS) que nous allons tous les deux voir dans les chapitres suivants.

1.4 Règles du jeu du puissance 4 en général

Le jeu du puissance 4 se joue normalement sur une grille placée à la verticale. Chacun des deux joueurs possède 21 pions d'une couleur et se place d'un des deux côtés de la grille. La grille est de taille 6 x 7 avec 7 étant le nombre de colonnes et 6 étant le nombre de lignes. Les pions doivent être déposés depuis le haut de la grille et la gravité se charge de placer le pion au bon endroit. Le pion se positionne soit tout en bas de la grille, soit sur un ou plusieurs autres pions déjà positionnés. Les deux joueurs jouent chacun à leurs tours et déposent un pion dans la grille. Le but est simple : Il suffit d'aligner sur une ligne, sur une colonne ou sur une diagonale, une suite consécutive de 4 pions d'une même couleur. L'objectif pour chacun des deux joueurs est de gagner en plaçant 4 pions (de sa couleur) d'une des trois manières possibles mais en même temps, d'empêcher l'autre joueur de gagner. Si la grille est pleine et qu'aucun des deux joueurs n'a gagné, la partie est déclarée nulle.

Voici les quatre positionnements possibles pour gagner (pions jaunes gagnants) :

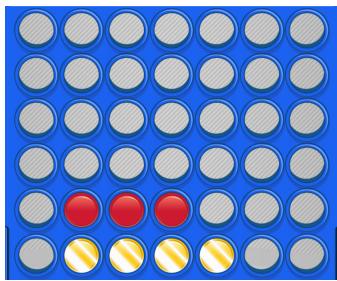


FIGURE 1.1 – Horizontal

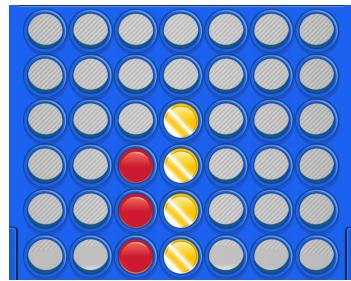


FIGURE 1.2 – Vertical

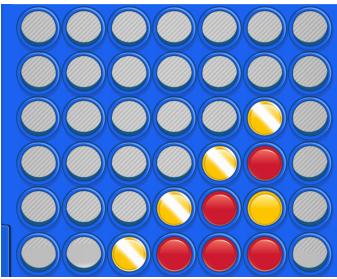


FIGURE 1.3 – Diagonale avant

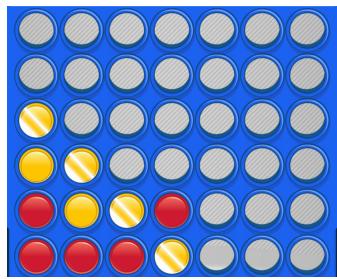


FIGURE 1.4 – Diagonale arrière

1.5 Règles du jeu du tic - tac - toe

Maintenant que nous savons en quoi consistent les règles du jeu du puissance 4, intéressons nous à celles du jeu du tic - tac - toe. Nous verrons par la suite, que c'est en implémentant ce jeu que nous avons commencé notre projet et nous expliquerons quels avantages cela a représenté.

Le jeu du tic - tac - toe se joue sur une grille à l'horizontale, de taille 3 x 3 dans sa version la plus classique, mais il se joue aussi sur des grilles carrées de taille 4 x 4 et 5 x 5 voire 6 x 6. Chaque joueur possède 5 pions (dans la version 3 x 3) de même couleur (ou même signe) dans la version 3 x 3. Chacun des deux joueurs pose à son tour un pion de sa couleur (ou de son signe), où bon lui semble dans la grille. Le but est d'aligner trois pions identiques, soit sur une ligne, soit sur une colonne soit sur une diagonale. Les règles sont presque identiques à celles du puissance 4, la plus grande différence est la gravité présente dans le puissance 4. Ensuite, pour ce qui est des grilles 4 x 4 et 5 x 5, le nombre de pions à aligner passe de 3 à 4 comme pour le puissance 4.

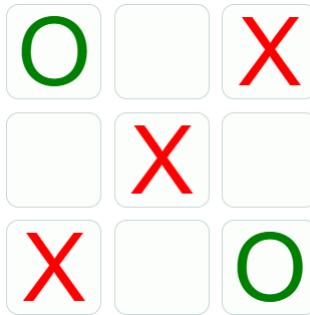


FIGURE 1.5 – Grille classique de tic - tac - toe avec des pions X et O

1.6 Langage et outils utilisés

Concernant le langage utilisé, c'est notre encadrant qui nous a proposé et nous avons été d'accord avec lui pour choisir le langage python3. C'est un langage de programmation intéressant, plutôt simple, en constante évolution, qui possède une large bibliothèque et qui est souvent utilisé pour programmer des jeux comme le nôtre. De plus, contrairement à certains langages que nous avons découverts depuis le début de notre licence, python nous a permis de ne pas nous soucier de l'allocation en mémoire lorsque l'on programmat.

Concernant les outils utilisés, nous avons principalement travaillé sous l'éditeur de texte Sublime Text. Les structures de données que nous avons le plus manipulées sont les listes et les listes de listes. Elles nous ont permis de représenter la grille de jeu, stocker les pions posés par les joueurs, représenter toutes les dispositions possibles de la grille de jeu, récupérer tous les couples de coordonnées possibles de la grille... Ce sont des structures facile à manipuler qui nous ont permis de représenter toutes les tailles de grilles dont on avait besoin : petites, grandes, carrées et rectangulaires ; sans que cela demande trop de changements dans nos algorithmes. Ensuite, il faut savoir que les "arbres" dont nous allons parler tout le long du projet, sont des arbres fictifs, ils sont créés et parcourus par une fonction que nous allons voir. L'utilisation d'un type fictif a pour but de réduire l'espace mémoire pris par le programme puisque ce dernier ne stocke que des données linéaires.

1.7 Notre version du puissance 4

Notre puissance 4 se différencie un peu du vrai jeu. En effet, même si théoriquement le programme peut faire jouer les joueurs sur une grille de taille 6 x 6 avec 4 pions pour gagner (choix arbitraire de notre part), il faudrait un ordinateur assez puissant pour que l'IA place ses pions assez rapidement pour qu'on puisse jouer. La grille la plus grande sur laquelle nous avons pu jouer en tant que le joueur humain, contre l'IA, est de taille 4 x 4 avec 3 pions gagnants. Le programme ne répond pas assez vite pour permettre de jouer sur une grille plus grande ou avec 4 pions pour gagner. Une solution à ce problème est l'algorithme du Monte-Carlo : Nous aborderons ce sujet avec plus de précisions dans la partie "Perspectives" de notre rapport.

Ensuite, la seconde différence est à propos de la gravité, celle-ci est implémentée mais un pion posé trop haut ne tombe pas automatiquement en bas de la colonne : Le joueur adverse peut demander au programme de poser le pion où il veut, par contre le pion ne sera posé que s'il respecte la règle essentielle du puissance 4, à savoir la gravité. Le joueur adverse ne pourra donc poser que des pions tout en bas de la grille ou au dessus d'un ou plusieurs pions qui sont eux-mêmes en bas de la grille.

Troisième point un peu différent, peu importe la version du jeu nous avons fait un choix arbitraire : L'IA commence et joue en posant des "1" alors que l'adversaire joue en posant des "2" et une case vide est représentée par un 0.

Chapitre 2

Organisation du projet

2.1 Les étapes du projet

Durant toute la durée du projet nous avons réalisé avec l'aide de notre encadrant, un certain nombre de tâches liées que nous allons lister puis expliquer plus en détails dans les chapitres suivants :

- Écrire une fonction qui liste les positions jouables, possibles()
- Écrire une fonction estGagnant() d'abord pour le tic - tac - toe en taille 3 x 3 puis pour une grille carrée de taille n x n
- Écrire l'algorithme mini-max
- Écrire une boucle de jeu et une fonction coupHumain() pour lier toutes les fonctions, faire intervenir l'humain et commencer à mettre en place le vrai jeu
- Modifications pour estGagnant qui doit être capable de recevoir le nombre de pions alignés qu'il faut pour pouvoir gagner
- Passage du tic - tac - toe vers le puissance 4 en implémentant la gravité avec la fonction gravité() qui est ensuite appelée dans la fonction possibles()
- Optimisation de la fonction minimax() avec l'élagage alpha-bêta
- Optimisation de estGagnant afin de limiter le temps que l'IA met pour jouer et espérer pouvoir la faire jouer sur une grille de taille 4 x 4, 5 x 5 et 6 x 6
- Création d'une interface graphique
- Ecriture du rapport du TER

2.2 Fonctionnement du groupe

Dès la première semaine, après la réunion avec notre encadrant, nous avons commencé à travailler à quatre sur un même problème en "brain storming". Les idées étaient bonnes et nous formions un bon groupe, mais malheureusement seule une personne sur les quatre programmait effectivement. Nous avons donc, sur les conseils de notre encadrant, décidé de nous diviser en deux sous-groupes de travail, chargés de réaliser les deux mêmes tâches mais séparément pour se mettre un peu en rivalité. C'est à notre avis, la méthode qui a le mieux fonctionné. Ensuite, à partir du moment où nous n'avons plus pu nous voir, nous avons suggéré à notre encadrant de nous donner quatre tâches distinctes afin que nous puissions nous les répartir. C'est de cette manière que nous avons travaillé jusqu'à la fin du projet. Chacun réalisait ce qui lui était demandé de son côté et une fois qu'ils étaient validés, nous rassemblions nos travaux.

Concernant les échanges avec notre encadrant, nous avons pu convenir de 4 rendez-vous durant les mois de Janvier et Février. Lors de ces rendez-vous nous avons pu montrer ce sur quoi nous avions travaillé et notre encadrant a pu nous expliquer ce qu'on ne comprenait pas. On se mettait d'accord ensemble sur le travail à réaliser pour la prochaine fois où on se verrait. Par la suite, nous avons continué les échanges avec notre encadrant par email et nous avons également convenu d'une session d'appel sur Discord pour discuter de la fin du projet et l'écriture de ce rapport.

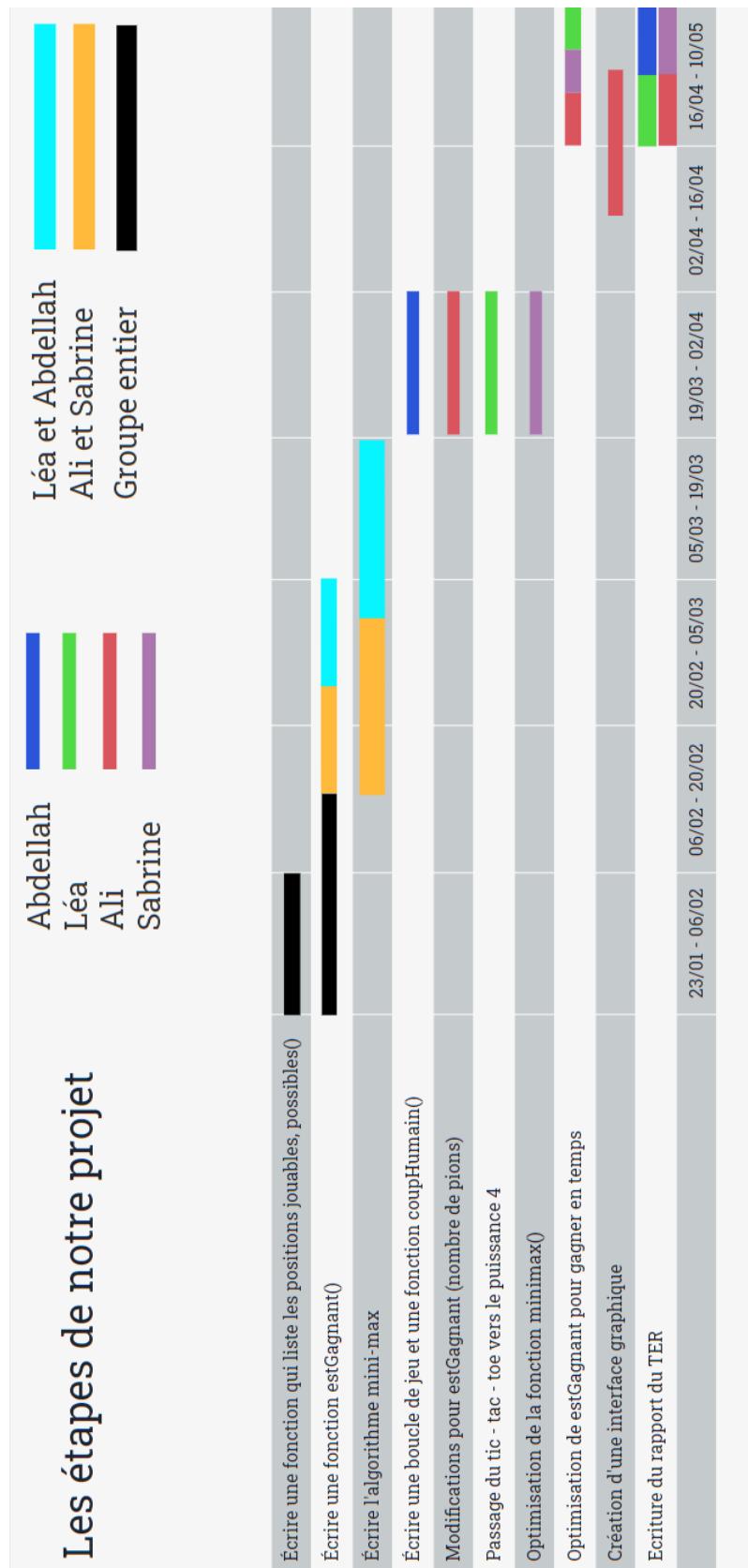


FIGURE 2.1 – Diagramme de Gantt pour présenter les étapes du projet

Chapitre 3

Algorithme mini-max

3.1 Première version de la fonction explore()

La première fonction que nous avons implémenté est la fonction `explore()`. Cette fonction prenait en paramètres : La grille dans son état actuel, un entier pour représenter la profondeur, et un entier pour représenter le joueur.

Dans sa première version, le but de cette fonction était de développer l'espace de recherche pour la grille passée en paramètre ; pour faciliter les explications on choisit de partir de la grille vide. La fonction développait l'arbre de toutes les possibilités de grilles différentes et les comptait. Elle partait d'une grille vide remplie de 0, la racine de l'arbre, plaçait le pion de l'IA dans une case de la grille, représentée par le premier noeud pour cette profondeur, puis la fonction repartait de la grille vide et plaçait de nouveau le pion de l'IA mais à un emplacement différent. Pour le premier niveau de profondeur, elle fait ça neuf fois car il y a neuf cases vides différentes.

Ensuite la récursivité fait que la fonction s'appelle elle-même avec une profondeur augmentée de 1 et le joueur inverse pour chacune des neufs grilles différentes. L'arbre continue de se déployer à partir du noeud où se trouve la grille dans laquelle le tout premier pion de l'IA est placé, il place huit fois le pion de l'humain des huit façons différentes dont il peut le placer et appelle à chaque fois la récursivité pour qu'elle explore le niveau de profondeur supérieure. L'arbre développe à partir du premier noeud, huit autres noeuds qui développeront chacun 7 nouveaux noeuds et ainsi de suite.

Toutes ces possibilités forment l'espace de recherche pour le jeu du tic - tac - toe. Il faut savoir qu'au maximum la fonction alloue neufs noeuds en même temps donc elle descend, sur un noeud, atteint la dernière disposition de la grille et remonte peu à peu en allouant neuf noeud à la fois. A chaque fois qu'elle atteint un noeud ou une feuille, elle l'ajoute à la somme avec "somme + = 1". La somme du nombre de noeuds dans l'arbre en partant d'une grille 3 x 3 vide, est de 623530

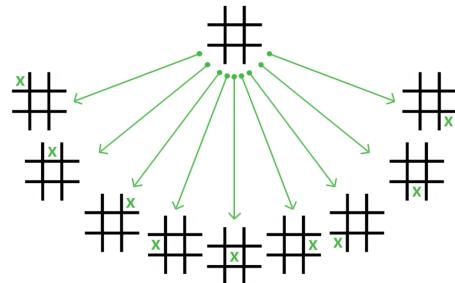


FIGURE 3.1 – Représentation de la profondeur 1 de l'arbre. Chacun des 9 noeuds donnera 8 noeuds...

3.2 Principe du mini-max

Le mini-max en général, est un algorithme qu'on utilise pour les jeux à deux joueurs. Il permet d'envisager toutes les possibilités différentes du jeu grâce à une mise en place de l'espace de recherche à l'aide d'un arbre comme fait la fonction `explore()` que nous avons vue. L'algorithme donne à chaque noeud (qui représente déjà une certaine grille plus ou moins remplie) une valeur, qui tient compte des bénéfices que cela peut apporter au joueur mais également à son adversaire. Chaque joueur, fera alors ses choix en fonction des différents noeuds qui exposeront trois situations possibles : Le noeud va le mener vers des situations gagnantes pour lui, le noeud va le mener vers des situations nulles ou le noeud va le mener vers des situations qui vont le faire perdre

Ces situations sont créées grâce à des scores associés aux grilles lorsqu'elles sont soient pleines, soit avec un gagnant. Les deux possibilités sont : Soit l'algorithme remonte les scores des situations finales vers la racine de l'arbre, soit il remonte les scores vers des situations intermédiaires qui sont évaluées par une fonction d'évaluation. Notre algorithme utilisera la première façon. L'algorithme dans les deux cas, remonte grâce à deux fonctions min et max, les meilleurs scores possibles (pour le joueur récurrent), peu à peu vers le haut de l'arbre. Pour la première solution, il les remonte vers le noeud où la grille se trouve, c'est à dire le noeud qui représente la situation de la grille avec les pions qu'elle contient déjà (pour notre situation). L'algorithme expose donc au joueur plusieurs possibilités de choix de coups possibles, associées à des scores et la fonction max se chargera de retourner le coup qui est le plus susceptible de mener à une victoire.

3.3 L'algorithme du mini-max dans notre programme

3.3.1 Deuxième version de la fonction `explore()`

La fonction `explore()` est la fonction récursive qui explore l'espace de recherche. Nous avons modifié la première version pour qu'elle serve en tant qu'algorithme du mini-max. Elle prend désormais, dans sa version finale, la grille de jeu, le nombre de pions qu'il faut pour gagner, la profondeur et le joueur qui doit jouer au moment où la fonction `explore()` est appelée.

La fonction renvoie une liste de deux éléments : Si c'est le premier appel, elle renvoie l'ordonnée et l'abscisse du meilleur coup possible pour l'IA, du moins bon coup possible pour l'humain. Si ce sont les appels récursifs à l'intérieur de la fonction, quand à eux, ils renvoient des scores associés aux situations de grilles (1, -1 et 0). Ces scores, mis dans des listes, vont tous être utilisés par les fonctions min et max pour faire remonter les meilleurs scores puis déterminer quel est la place du meilleur des coups dans la liste de tous les coups possibles pour la situation actuelle de la grille.

La fonction `explore()` est construite avec une boucle `for` dans laquelle il y a une récursivité où la fonction `explore()` s'appelle elle-même avec les bons paramètres (sauf à certaines conditions). Voici la boucle `for` qui choisit, tour à tour, un coup possible dans la liste des coups possibles appelée '`coupPossibles`' : '`for coup in coupPossibles :`'. La liste des `coupPossibles` est donnée par la fonction `possibles()` que nous verrons plus tard, elle donne une liste composée de sous listes contenant chacune les coordonnées d'une case vide de la grille.

Dans la boucle `for` qui va donc chercher à chaque tour, une sous liste dans la liste '`coupPossibles`', le pion du joueur passé en paramètre est placé selon les coordonnées trouvées dans la sous-liste prise dans la liste des possibles. Ensuite, la fonction teste si ce pion provoque une victoire pour le joueur :

- Si c'est le cas et que c'est l'IA, on ajoute alors à une nouvelle liste qui se crée, un 1
- Si c'est l'humain on ajoute un -1 à cette liste créée
- Si il n'y a aucune victoire et qu'il ne reste pas de place dans la grille, on met un 0 dans la liste créée pour spécifier que c'est une égalité.
- Si il reste encore des coups dans la liste des coups possibles, alors là, on ajoute à la liste créée, l'appel de la fonction `explore()` elle même avec comme argument, la grille, le nombre de pions gagnants, la profondeur augmentée de 1 et le joueur inverse. Cette récursivité va chercher à se développer, créer elle-même des listes pour chaque profondeur, tant qu'elle n'obtiendra pas une situation de victoire pour le joueur passé en paramètre. Si elle n'en trouve aucune même en arrivant sur les feuilles de l'arbre, elle cherchera à trouver une égalité, et dans le pire des cas prendra une situation où le joueur perd. La fonction remonte ensuite le meilleur des cas, grâce aux scores, dans la liste précédemment laissée

vide. Les fonctions `min()` et `max()` interviennent lorsque la liste des scores est complète et que c'est le moment de choisir parmi les coups, lequel est le meilleur pour le joueur donné. C'est le rôle de ces deux fonctions que nous allons voir...

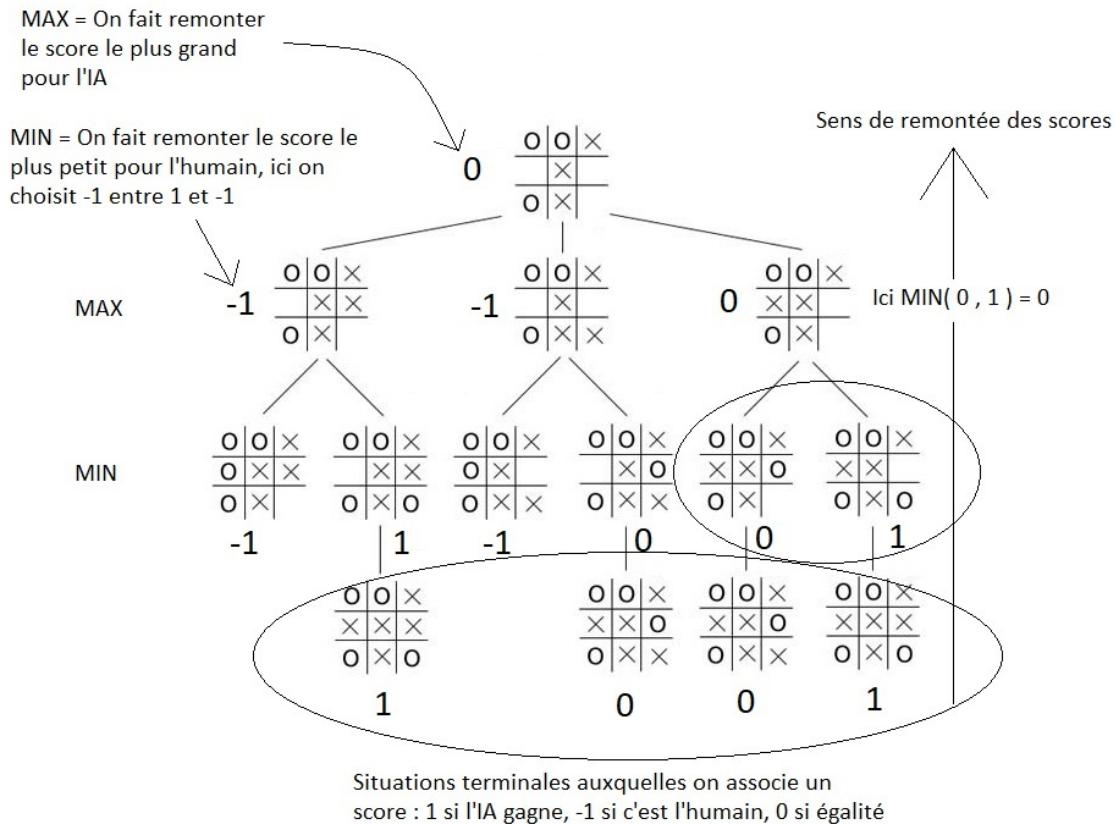


FIGURE 3.2 – Arbre représentant des grilles finales ainsi que les scores associés, remontés vers la racine

3.3.2 Fonction `min()`

La fonction `min()` prend en paramètre une liste appelée "scores", en retour de l'exploration des 9 premiers coups, cette liste contient 9 scores (si on suppose qu'on a donné la grille vide en paramètre), égaux à 0, 1 ou -1. Cette fonction est appelée dans le cas où on veut minimiser les chances, c'est à dire pour l'humain. Elle parcourt la liste et choisit le score minimal : -1 en premier, 0 en deuxième et 1 en dernier recours. Son résultat est utilisé dans la fonction `explore()` que nous avons vue. La fonction `min()` donne l'indice du chiffre dans la liste qui est le minimum de la liste et qui donnera une fois, corrélée à la liste des coups possibles, le meilleur coup que l'humain puisse jouer.

3.3.3 Fonction `max()`

La fonction `max()` marche quasiment de la même façon, mais opposée. C'est à dire qu'elle prend aussi en paramètre une liste appelée "scores", qui contient aussi 9 scores (ou moins si la grille n'est pas vide), égaux à 0, 1 ou -1. Cette fonction est appelée dans le cas où on veut maximiser les chances, c'est à dire lorsque c'est le tour de l'IA. Elle parcourt la liste et choisit le score maximal cette fois : 1 en premier, 0 en deuxième et -1 en dernier recours. Son résultat est aussi utilisé dans la fonction `explore()` que nous avons vue. La fonction `max()` donne l'indice du chiffre dans la liste qui est le maximum de la liste et qui donnera une fois, corrélée à la liste des coups possibles, le meilleur coup que l'IA puisse jouer.

Chapitre 4

Explications du programme

Dans ce chapitre nous allons parler des différentes fonctions qui composent notre programme.

4.1 Fonction "estGagnant"

La fonction `estGagnant()` est une fonction très importante pour le programme. C'est elle qui détermine à chaque fois qu'un pion est posé, s'il y a un gagnant parmi les deux joueurs. En premier lieu, rappelons que nous avons d'abord implémenté le jeu du tic - tac - toe avant d'implémenter les particularités du puissance 4. La grille 3 x 3 nous a permis de faire nos premiers programmes plus facilement que si cela avait été sur une grande grille et avec les règles de la gravité. De plus les changements qui ont été fait pour passer d'un jeu du tic - tac - toe à un jeu du puissance 4 ont été minimes, seule la gravité a dû être ajoutée car toutes les autres fonctions pouvait déjà s'appliquer à des grilles de taille n.

4.1.1 Première version

La toute première version que nous avons programmée était pour le tic - tac - toe. Elle prenait en paramètre la grille et le joueur. Elle renvoyait `True` si le joueur donné en argument avait un alignement gagnant dans la grille, `False` sinon. Nous avions utilisé l'extension du langage python : "NumPy" pour utiliser les matrices. La grille de jeu était donc représentée par une matrice à deux dimensions. Nous avions programmé cette version pour qu'elle fonctionne sur une grille taille 3 x 3. Elle récupérait facilement (étant donné que la taille était fixe), grâce à plusieurs fonctions, les lignes, colonnes et diagonales. La fonction cherchait ensuite, dans toutes ces lignes, colonnes et diagonales, un alignement de trois "1" si c'était l'IA le joueur donnée ou de trois "2" si c'était l'humain. Malheureusement le temps qu'allait coûter cet algorithme pour des grilles plus grandes allait devenir précieux pour la suite. Nous avons donc dû abandonner les matrices implémentées car selon notre encadrant, sont d'une manière, un peu trop coûteuses.

4.1.2 Avant optimisation

Nous avons donc modifié la fonction `estGagnant()`. Désormais, la grille de jeu était représentée par une liste de listes et non plus une matrice à double dimensions. De plus, nous l'avons modifié pour qu'elle fonctionne sur des grilles carrées plus grandes, comme 4 x 4, 5 x 5, 6 x 6 et même n. La complexité pour la fonction a alors beaucoup augmenté. Les paramètres ont dû être modifiés et la fonction prenait : La grille, le joueur et le nombre de pions alignés identiques qu'il fallait pour gagner ; et renvoyait toujours un booléen. L'idée principale est restée la même mais nous avons dû prendre en compte le nombre de pions alignés qu'il fallait pour repérer les vrais alignements gagnants dans la grille. Nous avons donc récupéré toutes les listes représentant toutes les lignes, colonnes et diagonales susceptibles de gagner, c'est à dire dont le nombre de cases était au minimum égal au nombre de pions passé en argument de la fonction. Puis nous avons cherché si un alignement du bon nombre de pions existait parmi toutes les listes récupérées. Nous verrons dans le chapitre sur la complexité que cette fonction a atteint ses limites dès que la grille devenait plus grande que 3 x 3. Nous avons donc optimisé la fonction !

4.1.3 Après optimisation

Pour optimiser `estGagnant()` nous avons éloigné notre réflexion de l'idée de base, c'est à dire celle de vérifier la grille dans son intégralité pour trouver un alignement. Nous avons ajouté un paramètre à la fonction : le dernier coup joué. Ce dernier coup est une liste de deux éléments comprenant l'abscisse et l'ordonnée de la case qui vient d'être remplie par le joueur qui vient de jouer et pour qui on va chercher un alignement gagnant. Sachant que les derniers appels de `estGagnant` ont renvoyé "False", nous savons que seul le dernier pion posé dans la grille peut permettre d'avoir un alignement gagnant. Cette optimisation a donc comme but, de non plus vérifier la grille entière, mais seulement la ligne, la colonne et les deux diagonales de la case où le dernier pion vient d'être posé.

La version finale de `estGagnant` se divise en trois parties :

- Les deux premières pour la ligne et pour la colonne :

Ces deux parties sont presque identiques : Une boucle while partout la ligne (respectivement la colonne), si le pion de la case est égal aux pions que pose le joueur donné en paramètre, on augmente un compteur '`nbPionsAlignes`' de 1, ensuite si le pion suivant n'est pas égal au précédent, on remet le compteur à zéro et on passe à la case suivante. A chaque fois que le compteur est augmenté on teste si le compteur est égal au nombre de pions qu'il faut pour gagner. Si il y a égalité, la fonction `estGagnant` renvoie directement `True`. Sinon, dès que le `j` (respectivement `i` pour la colonne) dépasse la taille de la ligne (respectivement de la colonne) on sort de la boucle pour passer à la suivante.

```

nbPionsAlignes = 0
j = 0
while j < dim :
    if etat[ligne][j] == pion :
        nbPionsAlignes += 1
        if nbPionsAlignes == nbPions : return True
    else : nbPionsAlignes = 0
    j += 1

```

FIGURE 4.1 – Boucle while pour tester la ligne avec (`etat[i][colonne] == pion`) pour la version colonne)

- La troisième pour les deux diagonales

Cette troisième partie s'occupe de tester la diagonale et l'anti-diagonale de la case où le dernier pion a été posé. Pour la diagonale, le principe est de reculer en diagonale depuis la case du coup joué, jusqu'à la case la plus à gauche en allant vers le haut, grâce à une première boucle while. Une fois cette case atteinte, on entre dans une deuxième boucle while où on parcourt la diagonale en ajoutant +1 au compteur de la ligne et +1 au compteur de la colonne et on réalise exactement le même test que pour la ligne et la colonne grâce au compteur. Si il y a égalité entre le compteur et le nombre de pions qui doivent être alignés, la fonction renvoie `True`, si le compteur dépasse la taille de la grille on sort de la boucle pour passer aux deux dernières. Le principe est le même pour l'anti-diagonale, sauf que pour atteindre la case de départ on va vers la gauche mais en descendant, également grâce à une première boucle while. Une fois cette case atteinte, pour parcourir l'anti-diagonale, on entre dans la deuxième boucle et on remonte d'une ligne à chaque fois, donc on ajoute -1 au compteur de ligne et on ajoute +1 au compteur de la colonne. Comme pour la diagonale, on fait le même test, si il y alignment la fonction renvoie `True`, sinon si le compteur dépasse la taille de la grille, on sort du while et comme c'est celui-ci le dernier, `estGagnant` renvoie "False" avec le `return`.

```

nbPionsAlignes = 0
lignetest = ligne
colonnetest = colonne
while lignetest > 0 and colonnetest > 0 :
    lignetest -= 1
    colonnetest -= 1
while lignetest < dim and colonnetest < dim :
    if etat[lignetest][colonnetest] == pion :
        nbPionsAlignes += 1
        if nbPionsAlignes == nbPions : return True
    else : nbPionsAlignes = 0
    lignetest += 1
    colonnetest += 1
return False

```

FIGURE 4.2 – Boucles while pour tester la diagonale (haut gauche de la grille vers le bas à droite) principe inverse pour tester l'anti-diagonale (bas gauche de la grille vers le haut à droite)

4.2 Fonctions "possibles()" et "gravite()"

La fonction possibles() est une fonction plutôt simple mais dont l'utilité est immense. Elle attend en paramètre la grille de jeu qui est une liste de listes et renvoie une nouvelle liste qui contient des sous-listes. Cette nouvelle liste peut être vide, ce qui signifie alors que la grille est pleine, que toutes les cases sont occupées par les jetons de l'IA ou de l'humain. Chaque sous-liste de la nouvelle liste créée représente un "coup", c'est à dire une certaine case de la grille, que l'un des deux joueurs pourra choisir. Cette case est représentée par une sous-liste qui contient deux éléments : l'abscisse et l'ordonnée de la case. La fonction possibles() s'occupe de parcourir la grille grâce à deux boucles while imbriquées. Elle teste si la case courante est vide ou pleine. Si elle est pleine elle passe à la suivante. Si elle est vide elle teste si on est en mode puissance 4 ou en mode tic - tac - toe. Si on est en puissance 4 elle teste si cette case respecte la gravité grâce à la fonction gravite() que nous allons voir, si elle la respecte, elle ajoute une sous liste pour cette case, sinon elle passe à la suivante. Si on est en mode tic - tac - toe elle rajoute directement une sous liste pour cette case dans la liste finale renvoyée. Au final, la fonction possibles() retourne une liste, composée d'autant de sous-listes qu'il y a de cases vides dans la grille, avec chaque sous-liste contenant les coordonnées d'une case.

La fonction possibles() fait appel à la fonction gravité() pour tester une certaine condition : Elle regarde si la case dont on est en train de s'occuper (et de voir si elle est pleine ou non), respecte la gravité. C'est à dire que cette case est placée soit tout en bas de la grille soit au dessus d'une autre case qui est elle même déjà remplie par un jeton d'un des deux jours et qui elle même soit se trouve en bas de la grille soit sur une case remplie. Plus précisément, gravité() prend en paramètre la grille, et les deux coordonnées d'une case : i et j. Elle teste si la case (i,j) est en bas de la grille avec " if i == len(etat)-1 : " ou si la case en dessous de la case (i,j) est une case non vide avec " if etat[i+1][j] != 0 : ". Cette fonction renvoie un booléen qui contrôle la condition de la création de la sous-liste qui elle, donne les coordonnées d'une case jouable.

```

def gravite(etat , i , j) :
    if i == len(etat)-1 :
        return True
    if etat [ i+1 ][ j ] != 0 :
        return True
    return False

```

FIGURE 4.3 – Code de la fonction gravité() qui est appelée dans possibles()

```

def possibles(etat):
    dim = len(etat)
    possibles = []
    i = 0
    while i < dim :
        j = 0
        while j < len(etat[i]) :
            if etat[i][j] == 0 :
                if typeJeu == "p4" :
                    if gravite(etat, i, j) :
                        possibles.append([i,j])
                else :
                    possibles.append([i,j])
            j += 1
        i += 1
    return possibles

```

FIGURE 4.4 – Code de la fonction possibles()

4.3 Les autres fonctions du programme

A présent, nous allons expliquer le rôle des autres fonctions, un peu moins importantes mais qui restent essentielles au bon fonctionnement du programme :

4.3.1 Fonction "affiche()"

La fonction affiche() sert à afficher l'état de la grille de jeu. Cette grille est représentée par une liste de liste, qu'on donne en paramètre à affiche(). La fonction parcourt l'ensemble de la liste et des sous-listes et affiche sur le terminal chaque pion (0, 1 ou 2). Deux boucles while sont imbriquées afin de permettre de parcourir toutes les sous-listes et à chaque fois qu'une ligne de la grille est représentée dans son intégralité, un retour à la ligne est effectué grâce à un "end" pour que le résultat affiché soit représentatif d'une vraie grille de puissance 4.

4.3.2 Fonction "coupHumain()"

La fonction coupHumain sert à coordonner tout ce qui doit se passer lorsque c'est le tour de l'humain. Elle affiche dans le terminal deux questions à l'attention du joueur humain. Elle attend que celui ci, après avoir choisi, lui donne les coordonnées de la case dans laquelle il veut placer son pion "2". La fonction teste si les deux entrées sont des chiffres entiers, si ils sont supérieurs ou égaux 0 et inférieur ou égaux à la taille de la grille (pour que la position soit dans la grille) et enfin, si la case est vide, c'est à dire qu'elle contient un "0". Si c'est le cas elle teste si on est en mode puissance 4 ou en mode tic - tac - toe. Si on est en puissance 4 elle teste si la gravité a été respectée et place le pion "2" dans la case demandée. Si on est en tic - tac - toe, elle place le pion "2" dans la case demandée. Si les entrées du joueur humain ne sont pas acceptées, elle signale au joueur que le coup demandé n'est pas possible, soit parce qu'il n'a pas respecté la gravité si on est en mode puissance 4, parce que les coordonnées ne font pas partie de la grille.

4.3.3 La boucle de jeu

La boucle de jeu est celle qui fait marcher le programme, c'est elle qui assemble les fonctions ensemble et envoie sur le terminal les informations dont le joueur humain a besoin. Elle appelle la fonction récursive mini-max, s'occupe de récupérer le résultat qui est donc une liste de deux coordonnées, celles du meilleur coup que l'IA va pouvoir placer en fonction de l'état de la grille actuelle. Ensuite la boucle s'occupe de placer dans la grille à la bonne case le "1" de l'IA et de tester si l'IA a gagné. Elle affiche pour l'humain la grille comprenant le coup que l'IA vient de placer grâce à la fonction affiche(). La boucle teste également si la grille est pleine à chaque fois qu'un coup de l'humain est posé. Puis, elle fait intervenir coupHumain() qui récupère le coup voulu pour l'humain et le place. Elle rappelle affiche() une seconde fois pour faire état de la grille avec le coup placé et teste si l'humain à gagner. Si ce n'est pas le cas elle refait un tour de boucle.

4.3.4 Les arguments attendus en ligne de commande

Pour fonctionner notre programme attend 3 informations que l'humain devra donner en ligne de commande, en même temps qu'il fera appel au programme entier avec python3. Ces trois informations sont : Le type de jeu (p4 ou ttt), la taille de la grille en un entier (3, 4, 5 ou 6), le nombre de pions qu'il faut pour gagner (3 ou 4). Trois variables globales attendent ces trois arguments pour pouvoir les stocker et les donner à des fonctions comme la fonction possibles() qui utilise 'typeJeu' pour conditionner ce qu'elle va renvoyer.

Chapitre 5

Perspectives avec la méthode Monte-Carlo

5.1 Une autre perspective d'approche : *La méthode de Monte-Carlo*

5.1.1 Rappels sur le minimax et choix de la perspective envisagée

Conformément à la troisième partie de ce rapport, l'algorithme Minimax a été employé dans le cadre de ce projet.

Cet algorithme consiste à passer en revue tout le champ des possibles dans un arbre de recherche comme structure de données, pour un nombre limité de coups.

Brièvement, l'implémentation d'un tel algorithme a pour but de maximiser les chances d'une IA de remporter la victoire au cours d'une partie face à l'humain via l'assignation d'un max et d'un min pour chaque état correspondant à la grille de jeu pour le Puissance 4.

Des tests ont été réalisés sur des grilles de différentes dimensions avec un nombre de pions à aligner défini. Via l'implémentation du minimax, la boucle de jeu est effective pour une grille $3*3$ et $4*4$ avec un nombre de pions à aligner inférieur à 4.

L'algorithme Minimax se base sur un arbre de recherche, en prenant pour exemple une grille de dimension $3 * 3$, on compte au maximum 623530 noeuds à explorer ce qui est relativement important.

En élargissant le champ des possibles avec une grille de dimension bien supérieure à celle proposée en exemple on parvient à obtenir un nombre très conséquent, ce qui est lourd à prendre en charge pour le processeur. Il est alors nécessaire d'être équipé d'un processeur très performant pour réaliser des tests sur des grilles de dimension supérieure à $4 * 4$ et de permettre à l'IA de placer ses pions rapidement.

La solution à cela concerne l'approche de **Monte-Carlo**, qui sera expliquée bien plus en détails dans la sous-partie suivante.

5.1.2 Introduction de la méthode de Monte-Carlo

L'illustration ci-dessous constitue l'espace de recherche du Puissance 4.

L'espace en question n'est pas complet sur l'illustration concernée puisque le nombre de possibilités est bien plus élevé suivant les coups joués par un joueur donné. Il y a en effet au départ sept configurations existantes pour le coup joué par le premier joueur (jeton rouge).

On retrouve aussi sept configurations possibles pour le coup joué par le premier joueur dans les six autres états de jeu existants en partant du sommet de l'espace de recherche, ils ne sont pas représentés sur l'illustration précédente.

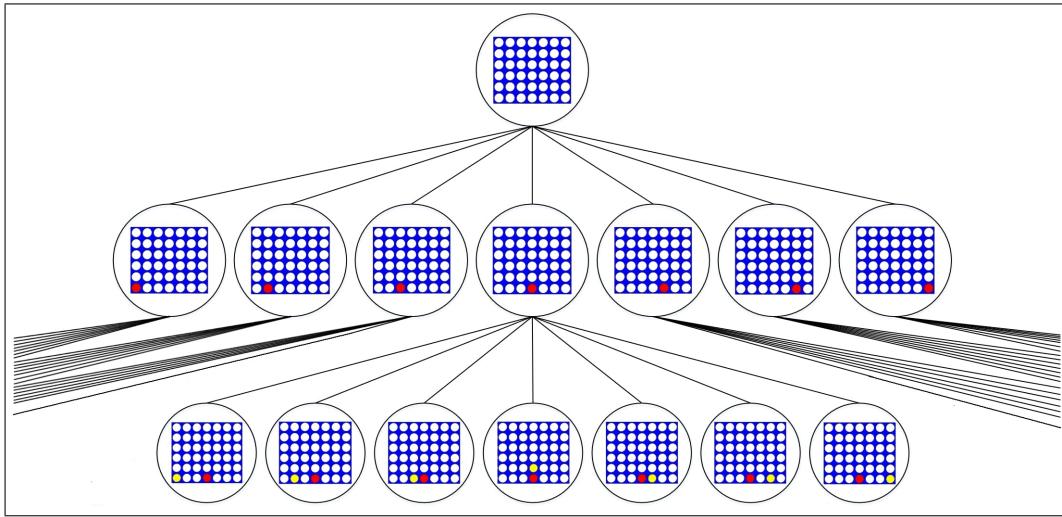


FIGURE 5.1 – Espace de recherche du Puissance 4

L'image ci-dessous représente les configurations possibles pour le premier état de jeu en partant du sommet. La démarche reste la même pour les autres états de jeu.

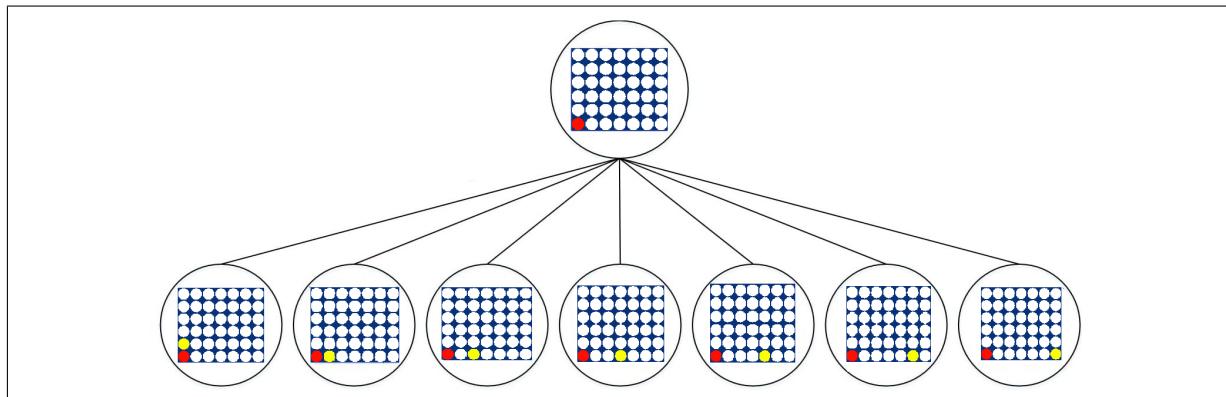


FIGURE 5.2 – Configurations possibles 1er état de jeu

Il est alors question ici, d'envisager une autre approche algorithmique qui est celle de la méthode de **Monte-Carlo**. La méthode de Monte-Carlo désigne au sens large une famille de méthodes algorithmiques, visant à calculer une valeur numérique approchée en faisant appel à des techniques dites "probabilistes". Le nom de cette méthode fait allusion aux jeux de hasard pratiqués au Casino de Monte-Carlo situé à Monaco.

Cette méthode a été développée dans les années 1950 par un groupe de chercheurs menés par *Nicholas Metropolis*

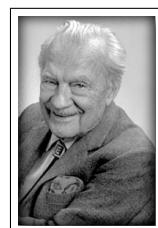


FIGURE 5.3 – Nicholas Metropolis

On retrouve la méthode de Monte-Carlo dans plusieurs domaines d'étude :

- Finance (estimation de la probabilité de la performance en bourse)
- Mathématiques (détermination de la valeur du nombre π / calcul d'intégrales)
- Algorithmique (estimation de la valeur d'un coup au jeu de Go / estimation de la valeur d'un coup aux échecs)

5.1.3 Une approche simplifiée

L'approche de Monte-Carlo est utilisée dans plusieurs jeux tour par tour tels que les échecs, le jeu de Go et le Puissance 4. L'approche classique du Monte-Carlo consiste simplement à limiter la profondeur de l'espace d'exploration qui sera représentée par un arbre de recherche, en étudiant les positions après un nombre de coups fixés. Les feuilles de l'arbre de recherche correspondent précisément à une position terminale du jeu. Pour le Puissance 4, il y a 3 positions terminales possibles :

- Victoire de l'IA
- Match Nul se traduisant par une saturation de la grille
- Victoire de l'humain

Cette approche consiste ainsi à explorer de façon systématique une branche de l'arbre de recherche jusqu'à une feuille (position terminale). Étant donné le nombre astronomique de parties possibles, cette approche ne peut pas explorer exhaustivement toutes les possibilités : il faut choisir un sous-ensemble des parties possibles

En partant du principe de l'exploration partielle d'un arbre de recherche jusqu'aux feuilles, l'implémentation d'un algorithme Monte-Carlo efficace doit répondre à deux problèmes :

- Le choix des séquences à explorer : Comment choisir celles qu'on explore ?
- L'évaluation des coups joués : Comment déterminer le meilleur coup possible après une succession de coups joués ?

5.1.4 Stratégie et recherche arborescente MCTS Monte-Carlo Tree Search

A partir d'une situation donnée, on joue une partie jusqu'à une situation finale (gain, perte ou une égalité).

On choisit aléatoirement un noeud d'une branche de l'arbre.

- Pas d'heuristique utilisée pour trier les noeuds et élaguer l'arbre
- L'IA joue contre elle-même un grand nombre de fois et mémorise les chemins statistiquement intéressants

L'illustration ci-dessus décrit l'arbre de jeu du Puissance 4. Il est question ici de comprendre les différentes phases sur lesquelles se base l'algorithme MCTS Monte-Carlo Tree Search. On va notamment s'intéresser aux coups les plus prometteurs et aux coups sur lesquels on a le moins d'information. L'algorithme MCTS Monte-Carlo Tree Search peut se décomposer en 4 phases :

- (1) Sélection
- (2) Expansion
- (3) Simulation
- (4) Rétropropagation

La phase (1) va consister à déterminer quelles sous-branches de l'arbre sont intéressantes à explorer. On peut partir par exemple de la racine de l'arbre et descendre progressivement pour voir quels coups sont les meilleurs parmi ceux qu'on aurait le plus envie de jouer (ceux qui ont le plus haut score). Cette phase existe séparément de la phase suivante car souvent elle est associée à des heuristiques de jeu positionnelles.

Une fois les sous-branches sélectionnées, il sera question de choisir de façon aléatoire les n coups (étant pré-calculés pour maîtriser le temps de réponse) à développer linéairement. Il s'agit ici de la phase d'**expansion**.

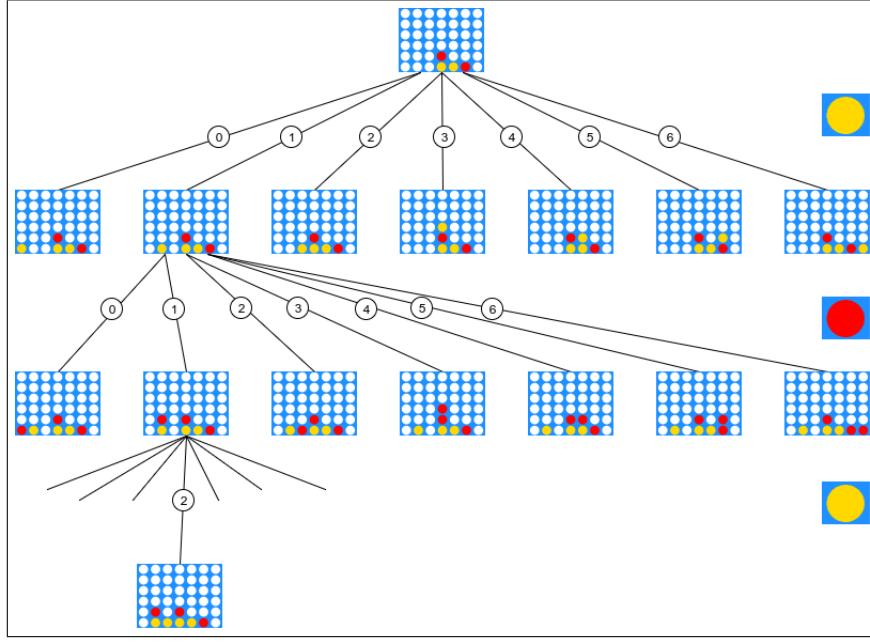


FIGURE 5.4 – MCTS : Monte Carlo Tree Search

Durant la phase de **simulation**, on n'ajoute aucun nœud à l'arbre. On essaie juste de simuler une partie aléatoire aussi vite que possible pour obtenir un résultat rapidement. De plus, ajouter tous les nœuds à l'arbre le rendrait trop grand en mémoire et ajouterait des nœuds qu'on a choisi aléatoirement, alors que c'est précisément ce que l'on veut éviter.

Lorsque l'état final du jeu est atteint, on calcule le score des joueurs puis on le reporte sur les nœuds sélectionnés dans l'arbre, avec un pourcentage équivalent au nombre de fois où ils ont été sélectionnés. C'est ce que l'on appelle la **rétropropagation**.

L'algorithme MCTS Monte-Carlo Tree Search porte bien son nom car ce dernier revient à construire un arbre de recherche en utilisant l'algorithme de Monte Carlo.

Cet algorithme a vraiment été un important du point de vue de l'intelligence artificielle pour les jeux tour à tour. À une époque où les machines étaient incapables de jouer au jeu de go, l'algorithme MCTS Monte-Carlo Tree Search a permis de trouver un moyen autre que celui des fonctions d'évaluation et ainsi de progresser. Il a aussi donné un cadre dans lequel sont venus s'inscrire les algorithmes d'apprentissage. On pourrait citer d'autres techniques d'apprentissage tels que le Q-learning ou le Deep Reinforcement Learning.

Les différentes phases de l'algorithme MCTS Monte-Carlo Tree Search peuvent être illustrées par le diagramme ci-dessous :

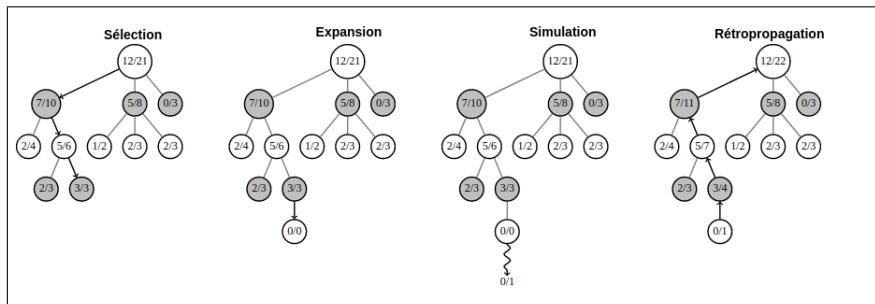


FIGURE 5.5 – Diagramme MCTS

Chapitre 6

Complexité des algorithmes

6.1 Analyse des algorithmes

L'analyse des algorithmes consiste à déterminer les ressources en temps, ou en espace mémoire (ou d'autre ressources telles que les processeurs) nécessaires à l'exécution de l'algorithme. Ce calcul dépend de la taille de la donnée, qui peut être sa grandeur (par exemple calcul de nième nombre de, Fibonacci ou du nième nombre premier), ou son nombre d'éléments (par exemple ordonner un tableau de n éléments).

Le choix des opérations de base est très important : on n'a pas forcément les mêmes résultats quand on choisit de compter le nombre de comparaisons de 2 éléments, ou le nombre d'échanges de 2 éléments comme opération de base dans un tri de tableau. Exemple : tri d'un tableau de n éléments.

- L'algorithme de tri d'un tableau par sélection et échange, a un comportement en n , quel que soit le tableau donné.
- L'algorithme de tri d'un tableau par l'algorithme de tri rapide, a un comportement en n^2 dans le pire cas, un comportement en $n \times \log_2(n)$ dans le meilleur cas et également en moyenne.

6.2 Ordre de complexité

On précise ici que temps estimé pour $n = 65536$, $k = 2$, avec une opération de base coûtant 0,01s

- Temps constant indépendant de la donnée : 1
- Temps logarithmique : $\log_2 n$ (0,16s)
- Temps linéaire : n (655,36s)
- Temps quadratique : n^2 (497 jours)
- Temps polynomial : n^k avec $k > 2$
- Temps exponentiel (dès que n devient « grand » le calcul prend un temps excessif.) : k^n avec $k > 1$ (21019728s soit 6,171019717 millénaires!)

6.3 O.

C'est la notation utilisée dans ce rapport et sa définition est la suivante : Etant donnée une fonction $f(n)$, $O(f(n))$ est l'ensemble des fonctions $g(n)$ telles qu'il existe une constante positive réelle c et un entier non négatif N tel que pour tout $n > N$, $g(n) < c f(n)$. La notation $O(f)$ décrit le comportement asymptotique d'une fonction, c'est à dire pour des grandes valeurs. En d'autres mots, g est $O(f)$ lorsque, pour des valeurs suffisamment grandes de n ($n > N$), le rapport $g(n)/f(n)$ reste toujours borné par c .

Exemples :

- $n^2 + 10n \in O(n^2)$ pour $n \leq 1$ en effet : $n^2 + 10n \leq n^2 + 10n^2 = 11n^2$ donc $c=11$ et $N=1$
- $n^2 + 10n \in O(n^2)$ pour $n \geq 10$ en effet : $n^2 + 10n \leq n^2 + n \times n = 2n^2$ donc $c=2$ et $N=10$
- $n \in O(n^2)$ pour $n \leq 1$ en effet : $n \leq n^2$ donc $c=1$ et $N=1$

Petite remarque : La notation O décrit une borne supérieure. On peut donc l'utiliser pour obtenir une borne supérieure sur le temps d'exécution dans le pire des cas. Evidemment, ce faisant, on obtient aussi une borne supérieure pour des données quelconques.

6.4 Complexité des fonctions du code

La fonction affiche() : Elle parcourt la grille et affiche ses cases donc elle est de complexité $O(i \times j)$ tel que i est le nombre de colonnes et j est le nombre de lignes

La fonction couphumain : $O(1)$, complexité linéaire puisque les coups de l'IA sont aléatoires et les coups de l'humain sont prévisibles.

La fonction estGagnant() : 2 boucles superposées qui valent $O(n)$ chacune donc la fonction est de complexité $O(n)$ tel que n est la taille de la grille

La fonction gravite() : Elle n'a ni boucles ni appels récursifs donc elle est à complexité linéaire $O(1)$

La fonction possible() : 2 boucles imbriquées qui valent $O(n)$ chacune donc la fonction est de complexité $O(n^2)$ tel que n est la taille de la grille

La récursive du mini-max :

Un algorithme optimal :

- Basé sur le théorème du Minimax de von Neumann
- Mais assez coûteux en temps et en espace
- Des améliorations et approximations existent
- 2 joueurs : «Max» et «Min»
 - Jouent à tour de rôle
 - Somme nulle
- État initial : grille vide
- Etat final : une ligne, une colonne, ou une diagonale, remplie par un même joueur
- Action : marquer une case vide
- Evaluation : +1 si «Max» gagne ; -1 si «Min» gagne ; 0 si égalité

Sa complexité : Si b est le nombre de coups possibles par situation et m la profondeur maximale de l'arbre, minimax a une complexité :

- en temps de $O(b \times m)$
- en espace $O(b \times m)$

Comme évoqué précédemment, cet algorithme réalise une exploration en profondeur d'abord complète de l'arbre de jeu (ou jusqu'à la profondeur maximale). Ainsi si la profondeur maximale de l'arbre est de m et qu'il y a b coups légaux à chaque point, alors la complexité en temps est $O(b \times m)$. Effectivement, dans sa forme la plus simple, l'algorithme explorera tous les noeuds pour déterminer le meilleur des coups.

Concernant la complexité en espace, dans le cas où toutes les actions sont générées en une fois, elle est de $O(b \times m)$. En effet tous les états précédents doivent être conservés tant qu'ils n'ont pas été explorés. Dans le cas où les actions sont générées les une après les autres, la complexité est de $O(m)$, seuls les états explorés sont conservés.

La fonction explore() : C'est la fonction qui crée l'arbre fictif des coups possibles elle explore donc chaque possibilité qui est représentée par un nœud fictif d'où son nom «explore»

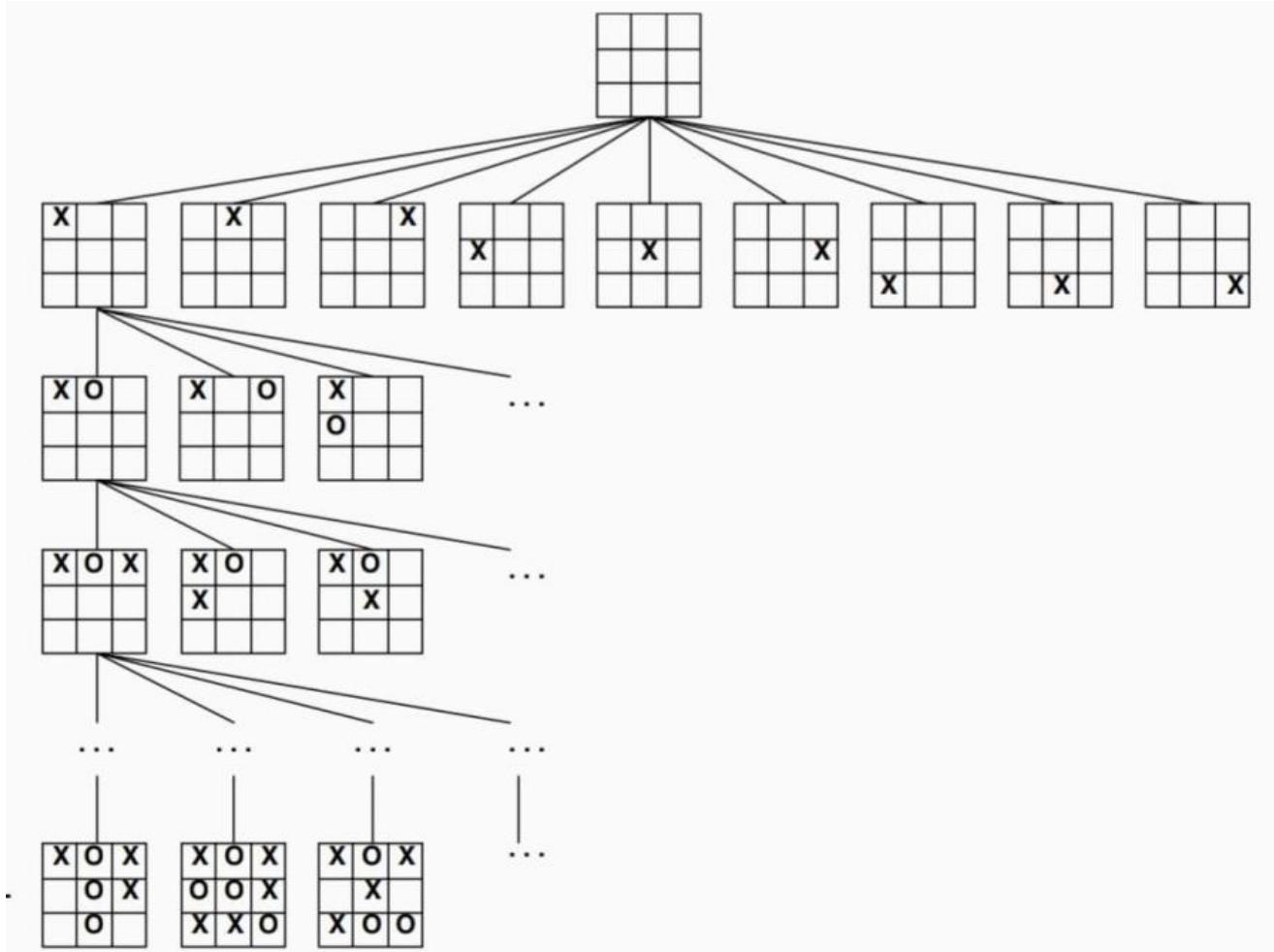


FIGURE 6.1 – Arbre Des Coups Possibles

La fonction se présente sous la forme :

```

1 void func(m) {
2     for(int i=0 ; i<m ; i++) {
3         for(int j=0 ; j<n ; j++){
4             // instructions
5         }
6         func(m-1);
7     }
8 }
```

FIGURE 6.2 – Arbre Des Coups Possibles

- Avec 2 boucles imbriquées avec un appel récursif dans la la boucle principale
- Sa complexité dépend du nombre de nœuds'

Pour déterminer précisément la complexité, puisqu'il s'agit d'un algorithme récursif, il faut établir l'équation de récurrence. Ici, la ligne « // instructions » est effectuée $m \times n$ fois, et « func(m - 1) » est appelé m fois. Donc la complexité $C(m)$ vérifie : $C(m) = mn + mC(m-1)$.

On peut "dérouler" la récurrence à la main, en remplaçant $C(m-1)$ par sa valeur $(m-1)n + (m-1)C(m-2)$:

$$C(m) = m \times n + m \times ((m-1) \times n + (m-1) \times C(m-2)) = m \times n + m \times (m-1) \times n + m \times (m-1) \times C(m-2).$$

En continuant ainsi, on trouve :

$$C(m) = n \times (m + m(m-1) + m(m-1)(m-2) + \dots + m(m-1)\dots1) = n \times m!(1/(m-1)! + 1/(m-2)! + \dots + 1/1!).$$

Ce qu'il reste à faire, c'est comprendre combien vaut la somme $(1/(m-1)! + \dots + 1/1!)$.

Au pire, on peut toujours la borner par m (puisque chaque terme est < 1) mais ça donnerait $C(m) = nm!m$. En fait, cette somme est bornée par $e = \exp(1) = 2.78\dots$. La raison mathématique est que $\exp(x) = x/0! + x/1! + x/2! + x/3! + \dots$ (jusqu'à l'infini). Donc la somme qui nous intéresse est le début de la somme pour $\exp(1)$ (en remplaçant x par 1), et donc elle est $\leq e$.

Autrement dit, le nombre total de fois que la ligne « // instructions » est effectuée est $\leq e \times n \times m! = O(n \times m!)$

\Rightarrow de 1 et 2 on déduit donc qu'elle est de complexité $O(i \times j!)$

tel que i est la taille de la grille j est la dimension de la grille

Chapitre 7

Interface Graphique

7.1 Présentation de l'environnement PyGame

7.1.1 Intérêt d'utilisation de la bibliothèque PyGame

PyGame est une bibliothèque libre multi-plateforme qui facilite le développement de jeux vidéo en temps réel avec le langage de programmation Python, langage utilisé dans le cadre de ce projet. Cette bibliothèque permet principalement de développer une interface graphique par l'intermédiaire du langage Python.



FIGURE 7.1 – Logo PyGame

Il est intéressant de se demander pourquoi avoir choisi une telle bibliothèque pour le développement d'une interface graphique.

Celle-ci possède plusieurs points forts qui peuvent être énumérés ci-dessous :

- Une portabilité sur divers systèmes d'exploitation
- Facilité d'utilisation avec la multitude de fonctionnalités offerte par une telle librairie
- Construction sur la bibliothèque logicielle libre Simple DirectMedia Layer (SDL) qui supporte plusieurs OS
- Gestion de l'affichage vidéo, de périphériques communs avec le clavier et la souris, de l'audio numérique

7.1.2 Exemples d'utilisation

Pour se familiariser avec les fonctionnalités proposées par la bibliothèque PyGame, il peut être intéressant de pouvoir découvrir ce qu'il en est à travers différents exemples d'utilisation qui seront présentés à la suite.

Le programme 1 permet de construire une fenêtre d'affichage donnée en spécifiant ses dimensions (largeur, hauteur) exprimées en pixels. Après exécution du programme, une fenêtre d'affichage noire se lance et change de couleur après appui sur une touche de clavier par l'utilisateur, la couleur de la fenêtre d'affichage bascule du noir vers le blanc et vice-versa. Le code source et les résultats obtenus sont illustrés sur la page suivante.

```

1 import pygame
2
3
4
5 noir = (0,0,0)      ## Code couleur RVB associé au Noir
6 blanc = (255,255,255) ## Code couleur RVB associé au Blanc
7
8 pygame.init()
9
10 ## Création d'une fenêtre de dimension 640 * 480 pixels
11
12 fenetre = pygame.display.set_mode((640,480))
13
14 execution = True
15
16 while execution:
17     for event in pygame.event.get():
18         if event.type == pygame.QUIT:    ## Evenement : Fermer la fenêtre d'affichage après execution du programme
19             execution = False
20         if event.type == pygame.KEYDOWN: ## Evenement : Appuyer sur une touche du clavier
21             noir,blanc = blanc,noir      ## Permutation des couleurs après l'événement pygame.KEYDOWN
22
23             ## La fenêtre d'affichage change de couleur dès lorsque l'on appuie sur une touche du clavier
24             ## en passant du noir au blanc
25
26             fenetre.fill(noir)          ## Fenêtre d'affichage noire après execution de ce programme
27             pygame.display.flip()

```

FIGURE 7.2 – Code source du programme 1

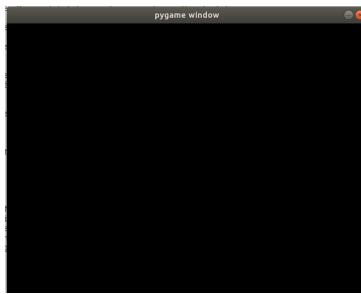


FIGURE 7.3 – Fenêtre d'affichage avant l'appui sur une touche de clavier



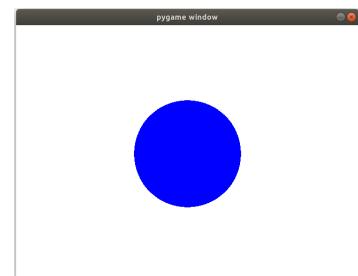
FIGURE 7.4 – Fenêtre d'affichage après l'appui sur une touche de clavier

Intéressons nous à présent à la construction de figures géométriques, la bibliothèque PyGame compte une multitude de fonctions permettant de réaliser des figures géométriques à partir d'une fenêtre d'affichage. L'exemple ci-dessous concerne la construction d'un cercle avec un rayon spécifique, le code source correspondant se base en partie de celui du premier programme.

```

1 import pygame
2
3
4
5 blanc = (255,255,255)      ## Code couleur RVB associé au Blanc
6 bleu = (0,0,255)            ## Code couleur RVB associé au Bleu
7
8 pygame.init()
9
10 ## Création d'une fenêtre de dimension 640 * 480 pixels
11
12 fenetre = pygame.display.set_mode((640,480))
13
14 execution = True
15
16 while execution:
17     for event in pygame.event.get():
18         if event.type == pygame.QUIT:    ## Evenement : Fermer la fenêtre d'affichage après execution du programme
19             execution = False
20         fenetre.fill(blanc)           ## Mise à jour de la fenêtre d'affichage
21
22         pygame.draw.circle(fenetre,bleu,(320,240),100) ## Dessin d'une figure géométrique (Cercle de rayon 100 px)
23                                         ## Position géométrique du cercle dessiné en pixels (320 px ; 240 px)
24
25         pygame.display.flip()          ## Mise à jour de la fenêtre d'affichage

```



Le troisième et dernier exemple d'utilisation va concerner la création de boutons. Cet exemple consiste à créer une suite de boutons ayant une forme géométrique donnée, après un clic sur un des boutons une animation est déclenchée (disparition du bouton / changement de couleur d'un bouton).

```

1 import pygame
2
3 pygame.init()
4 clock = pygame.time.Clock()
5
6 bleu_ciel = (0, 200, 255)
7 blanc = (255, 255, 255)
8 vert = (0, 255, 0)
9 rouge = (255, 0, 0)
10
11 ## Création d'une fenêtre de dimension 640 * 480 pixels
12 fenetre = pygame.display.set_mode((640,480))
13
14 execution = True
15 while execution:
16     background = pygame.Surface(fenetre.get_size())
17     background.fill(bleu_ciel)
18     # Ajout du fond dans la fenêtre d'affichage
19     fenetre.blit(background,(0,0))
20
21     # Dessin d'un rectangle de contour blanc
22     # (x = 75 ; y = 10) coordonnées de position du rectangle blanc par rapport au coin supérieur gauche de la fenêtre
23     # Largeur = 100 pixels Hauteur = 50 pixels
24     rectangle_blan = pygame.draw.rect(fenetre,blanc,[75,10,100,50])
25
26     # Dessin d'un rectangle vert
27     rectangle_vert = pygame.draw.rect(fenetre,vert,[250,10,100,50])
28
29     # Position du curseur par rapport à la fenêtre d'affichage
30     # Retourne 1 si le curseur est au-dessus d'un rectangle
31     mouse_xy = pygame.mouse.get_pos()
32     over_blan = rectangle_blan.collidepoint(mouse_xy)
33     over_vert = rectangle_vert.collidepoint(mouse_xy)
34
35     for event in pygame.event.get():
36         if event.type == pygame.QUIT: # si fermeture de la fenêtre avec un clic de la souris
37             execution = False
38         # Si clic gauche de la souris sur le rectangle vert, celui-ci devient de couleur rouge
39         elif event.type == pygame.MOUSEBUTTONDOWN and over_vert:
40             vert = rouge
41
42         # Si clic gauche de la souris sur le rectangle blanc, celui-ci disparait
43         elif event.type == pygame.MOUSEBUTTONDOWN and over_blan:
44             blanc = bleu_ciel
45
46     # Actualisation de l'affichage
47     pygame.display.flip()
48     # 10 images par seconde
49     clock.tick(10)
50
51

```

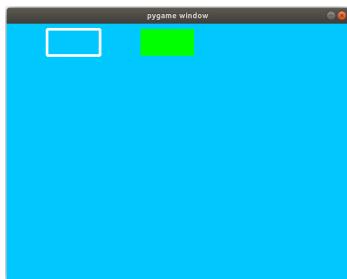


FIGURE 7.5 – Fenêtre d'affichage

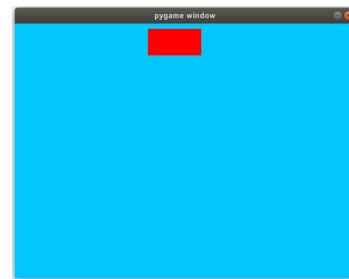


FIGURE 7.6 – Animations après clic

La bibliothèque PyGame propose davantage de fonctionnalités, de nombreux tutoriels sont présents afin de se familiariser avec cet environnement. La documentation de PyGame est consultable via le lien ci-dessous : <https://www.pygame.org/docs/>

7.1.3 Interface graphique proposée pour le Puissance 4

Il est à présent important de se focaliser sur le projet concerné, il sera question ici de présenter et d'expliquer le développement de l'interface graphique proposée pour le jeu du Puissance 4. Premièrement, il a été nécessaire de définir correctement l'environnement de jeu. Le code ci-dessous décrit la définition des paramètres géométriques sur l'environnement concerné. Il a été question de fixer une taille pour la grille de jeu adaptée pour une dimension 6 * 7, cette taille a été choisie de manière à ce que cela soit adaptée avec la dimension souhaitée. Il a fallu de plus, définir le rayon du cercle autrement dit l'emplacement nécessaire à l'insertion des jetons. Cela a été défini en tenant compte de la propriété géométrique d'une case (rectangle avec un cercle inscrit). Pour rendre un tel environnement jouable, il a fallu ajouter à cela des jetons de jeu (jaune et rouge), des pions utilisées pour des fonctions prochaines. La fenêtre de jeu doit être adaptée avec le dimensionnement de la grille, d'où la définition de largeur et de hauteur sur le code ci-dessous.

```

243 ##### INTERFACE GRAPHIQUE #####
244
245 taille_case = 100          ## Taille en pixels d'une case de la grille de jeu
246 rayon_cercle = int(taille_case/2 - 4) ## Rayon du cercle
247 cadre = (0, 0, 255)        ## Cadre bleu de la grille de jeu
248 blanc = (255, 255, 255)   ## Case initialement non remplie
249 jeton_rouge = (255, 0, 0)  ## Jeton utilisé par l'utilisateur
250 jeton_jaune = (255, 255, 0)## Jeton utilisé par l'IA
251 pion_joueur = 2           ## pion utilisé par l'utilisateur
252 pion_IA = 1               ## pion utilisé par l'IA
253
254 humain = 0                ## Variables 'humain' et 'IA' utilisées pour définir les tours de jeu
255 IA = 1
256
257
258 largeur = len(etat[0]) * taille_case      ## Dimensions de la grille de jeu
259 hauteur = (len(etat) + 1) * taille_case

```

FIGURE 7.7 – Spécifications Géométriques

Une fois avoir présenté l'environnement de jeu, il est maintenant important d'aborder les fonctions nécessaires à l'évolution du jeu.

La première fonction *inserer jeton* prend comme paramètres :

- Un état du jeu qui est la grille
- i étant l'indice de ligne de la case utilisée pour l'insertion d'un jeton
- j étant l'indice de colonne de la case pour laquelle on veut insérer un jeton

Celle-ci effectue au cours du jeu les insertions de jeton pour l'humain et l'IA, on retrouvera cette fonction dans la boucle de jeu qui sera abordée plus tard.

La fonction suivante *emplacementValide* prend comme paramètres l'état de jeu et l'indice de la colonne de la case souhaitée pour effectuer l'insertion d'un jeton de jeu. L'emplacement est valide lorsqu'un jeton occupe la position la plus basse possible sur une colonne donnée. Cette fonction renvoie un booléen pour indiquer si l'insertion est permise ou non.

La troisième et dernière fonction de jeu *rangeeSuivante* prend les mêmes paramètres que la fonction précédente. En revanche, celle-ci renvoie l'indice le plus grand de la ligne d'un état de jeu pas encore complétée de jetons de jeu. Exemple, si la dernière rangée est complétée de jetons de jeu il suffira de rechercher l'indice le plus grand de la rangée non encore complétée, étant donné qu'un jeton à insérer doit occuper la position la plus basse possible sur une colonne précise de la grille de jeu.

```

264 def inserer_jeton(etat,i,j,pion): ## Fonction permettant l'insertion d'un jeton de joueur (Utilisateur / IA)
265     etat[i][j] = pion
266     ## etat : grille de jeu , i : abscisse du jeton inséré
267     ## j : ordonnée du jeton inséré , pion : 1 ou 2
268
269 def emplacementValide(etat,colonne):    ## Fonction vérifiant si une case de la grille de jeu est inoccupée avant l'insertion d'un jeton de joueur
270     if etat[len(etat)-1][colonne] == 0: ## Le jeton inséré doit occuper la position la plus basse sur une colonne (gravité)
271         return True
272     else:
273         return False
274
275
276 def rangeeSuivante(etat, colonne): ## Fonction recherchant une ligne non occupée de la grille de jeu
277     for ligne in range(len(etat)):
278         if etat[ligne][colonne] == 0:
279             return ligne

```

FIGURE 7.8 – Fonctions de jeu

La fonction suivante est l'une des fonctions principales pour le développement de l'interface graphique. Celle-ci va s'occuper de l'affichage de la grille de jeu, la grille de jeu se construit de manière itérative. En parcourant toute la grille qu'on peut assimiler à une matrice, les cases se construisent progressivement. Chaque case est représentée par un rectangle avec des dimensions choisies (largeur,hauteur) et des coordonnées de position (x;y) par rapport à la grille de jeu. Chaque rectangle contient un cercle ayant un rayon défini et aussi des coordonnées de position (x,y) par rapport à la grille de jeu.

Les dimensions proposées dans le code ci-dessous ont été choisies de manière à ce que cela s'adapte au mieux aux dimensions de la grille de jeu.

Elle se construit ainsi progressivement dès lorsque des insertions de jetons de jeu ont été effectuées, cette fonction fera l'objet de plusieurs appels car à chaque insertion de jetons il est nécessaire de mettre à jour l'affichage de la grille de jeu.

```

244 grille = pygame.display.set_mode((largeur, hauteur)) # Définition de la grille de jeu
245 def grilleApparition(etat):
246     for colonne in range(len(etat[0])):
247         for ligne in range(len(etat)):
248             ## Construction des emplacements pour l'insertion des jetons
249             pygame.draw.rect(grille, cadre, (colonne*taille_case, (ligne+1)*taille_case, taille_case, taille_case))
250             pygame.draw.circle(grille, blanc, (int(colonne*taille_case + taille_case/2), int((ligne+1)*taille_case + taille_case/2)), rayon_cercle)
251
252     for colonne in range(len(etat[0])):
253         for ligne in range(len(etat)):
254             if etat[ligne][colonne] == pion_joueur:
255                 pygame.draw.circle(grille, jeton_rouge, (int(colonne*taille_case + taille_case/2), hauteur-int((ligne)*taille_case + taille_case/2)), rayon_cercle)
256             # Si c'est au tour de l'humain d'insérer un jeton, création d'un jeton rouge à insérer dans la case souhaitée
257             elif etat[ligne][colonne] == pion_IA:
258                 pygame.draw.circle(grille, jeton_jaune, (int(colonne*taille_case + taille_case/2), hauteur-int((ligne)*taille_case + taille_case/2)), rayon_cercle)
259
260     # Mise à jour de la grille de jeu
261     pygame.display.update()

```

FIGURE 7.9 – Création de la grille de jeu

La dernière étape ayant permis la réalisation de l'interface graphique est la fonction jeu(). Cette fonction peut se décomposer en plusieurs phases :

- Gestion des tours de jeu
- Insertion d'un jeton de jeu dans le cas où c'est au tour de l'utilisateur de débuter la partie
- Insertion d'un jeton de jeu dans le cas où c'est au tour de l'IA de commencer
- Détermination de l'état final du jeu (Victoire de l'utilisateur / Victoire de l'IA / Match nul)

Les différentes phases sont décrites par les deux illustrations qui suivent.

```

266 def jeu(etat):
267     fin_partie = False
268     myfont = pygame.font.SysFont("comicsansms", 40) ## Police d'écriture utilisée pour l'affichage du gagnant de la partie / égalité
269     tour = random.randint(humain,IA)    ## Gestion des tours de jeu
270
271     while not fin_partie and len(posibles(etat)) != 0:
272         for event in pygame.event.get():
273             ## Si clic sur la croix de la fenêtre d'affichage, celle-ci se ferme
274             if event.type == pygame.QUIT:
275                 sys.exit()
276
277             if event.type == pygame.MOUSEMOTION: ## Mouvement du curseur
278                 pygame.draw.rect(grille, blanc, (0,0, largeur, taille_case)) ## Bande horizontale blanche au-dessus de la grille de jeu
279                 posx = pygame.mouse.get_pos()[0] ## Récupération des coordonnées de position (x;y) du curseur de la souris
280                 if tour == humain: ## Si c'est au tour de l'humain de jouer, celui-ci va devoir insérer un jeton rouge dans la case désirée
281                     pygame.draw.circle(grille, jeton_rouge, (posx, int(taille_case/2)), rayon_cercle)
282                 else: ## Jeton jaune dans la case où c'est à l'IA de commencer la partie
283                     pygame.draw.circle(grille, jeton_jaune, (posx, int(taille_case/2)), rayon_cercle)
284
285
286         pygame.display.update() ## Actualisation de la fenêtre d'affichage
287
288
289         if event.type == pygame.MOUSEBUTTONDOWN: ## Evenement : clic gauche de la souris sur un emplacement inoccupé de la grille de jeu
290             pygame.draw.rect(grille, blanc, (0,0, largeur,taille_case))
291
292             if tour == humain:
293                 posx = pygame.mouse.get_pos()[0] ## Position en abscisse du curseur par rapport à la fenêtre d'affichage
294                 col = posx // taille_case ## Position en ordonnée du curseur par rapport à la fenêtre d'affichage
295
296                 if emplacementValide(etat, col):
297                     ligne = rangeeSuivante(etat, col) ## Insertion du jeton de l'utilisateur dans le cas où la situation est favorable
298                     inserer_jeton(etat, ligne, col, 2)
299
300                     if estGagnant(etat,4,"H",[0,0]): ## Si l'utilisateur remporte la partie, un message s'affiche en haut de la grille
301                         label = myfont.render("Victoire de l'humain !", 1, jeton_rouge)
302                         grille.blit(label, (40,10))
303                         fin_partie = True
304
305             tour += 1 ## Incrémentation des tours pour permettre aux joueurs de jouer l'un après l'autre
306             tour = tour % 2
307
308
309         ## Mise à jour de la grille de jeu après plusieurs insertions de jetons
310         grilleApparition(etat)
311         affiche(etat) ## Affichage de la grille en format ASCII sur le Terminal de Commandes

```

FIGURE 7.10 – Partie 1 de la fonction jeu

```

312     if tour == IA and not fin_partie:
313         longueur = len(possibles(etat))
314         coup = possibles(etat)[longueur // 2] ## Jeton placé sur la 4e colonne de la grille
315         colonne = coup[1]
316
317         if emplacementValide(etat, colonne):
318             ligne = rangeeSuivante(etat, colonne)
319             inserer_jeton(etat, ligne, colonne, 1)
320
321             if estGagnant(etat, 4, "IA",[0,0]):
322                 ## Si l'IA remporte la partie, un message s'affiche en haut de la grille
323                 label = myfont.render("Victoire de l'IA !", 1, jeton_jaune)
324                 grille.blit(label, (40, 10))
325                 fin_partie = True
326
327
328
329         tour += 1
330         tour = tour % 2
331
332         grilleApparition(etat)
333         affiche(etat)
334
335
336         if(possibles(etat) == [] and not (estGagnant(etat, 4, "IA",[0,0]) == True) and not (estGagnant(etat, 4, "H",[0,0]) == True)):
337             ## Message affiché sur la bande horizontale blanche, dans le cas d'une saturation de la grille
338             label = myfont.render("MATCH NUL", 1, (0,0,255))
339             grille.blit(label, (40, 10))
340             pygame.time.wait(3000)
341
342         if fin_partie:
343             pygame.time.wait(3000)
344             ## La fenêtre d'affichage se ferme 3000 ms après que la partie soit terminée
345
346
347     print(jeu(etat))

```

FIGURE 7.11 – Partie 2 de la fonction jeu

7.1.4 Tests effectués sur une grille traditionnelle 6 * 7

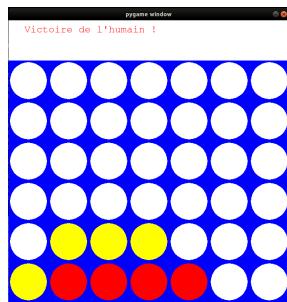


FIGURE 7.12 – Victoire de l'utilisateur

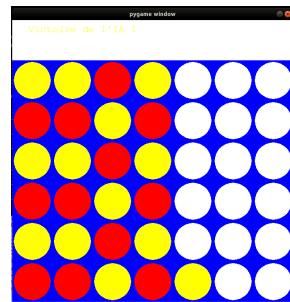


FIGURE 7.13 – Victoire de l'IA

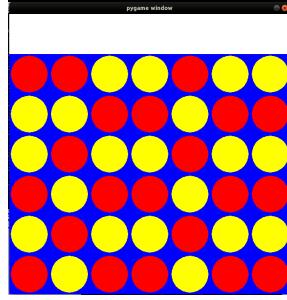


FIGURE 7.14 – Saturation de la grille

Chapitre 8

Bilan de notre projet

En conclusion, nous avons fini de développer une intelligence artificielle capable de jouer et gagner au jeu du puissance 4 sur des grilles de taille allant jusqu'à 6 x 6; face à un humain avec 3 pions gagnants. Nous sommes tous fiers du projet que nous vous avons présenté même si nous aurions aimé pouvoir continuer de le développer. Comme nous l'avons vu dans la partie 5, la prochaine étape de notre projet aurait été l'implémentation du Monte-Carlo dans le programme.

Pendant ce projet nous avons appris à travailler en groupe, à se partager les tâches, à communiquer, à compter sur le travail des uns et des autres et à se faire confiance. C'est une expérience que nous espérons tous de nouveau vivre l'année prochaine. Grâce à ce projet, nous avons appris des choses, nous avons découvert l'intelligence artificielle, les espaces de recherche avec les arbres fictifs, l'algorithme du mini-max, le fonctionnement de la méthode Monte-Carlo, l'implémentation d'une interface graphique mais aussi l'utilisation de Latex pour le rapport.

Enfin, nous tenons tous à remercier Monsieur Pompidor de nous avoir beaucoup aidés tout au long du semestre et pour avoir lu et corrigé ce rapport.

Chapitre 9

Bibliographie

Vandewiele Gilles. Creating the (nearly) perfect connect-four bot with limited move time and file size. Towardsdatascience. Medium. 28 Novembre 2017 [Consulté en Février 2020]. Disponible sur : <https://towardsdatascience.com/creating-the-perfect-connect-four-ai-bot-c165115557b0>

Philipp Muens. Game AIs with Minimax and Monte Carlo Tree Search. Towardsdatascience. Medium. 3 Avril 2019 [Consulté en Février 2020]. Disponible sur : <https://towardsdatascience.com/game-ais-with-minimax-and-monte-carlo-tree-search-af2a177361b0>

Aaron Wong. I created an AI that beats me at tic-tac-toe. Hackernoon. Sponsorium. 24 Février 2018 [Consulté en Février 2020]. Disponible sur : <https://hackernoon.com/i-created-an-ai-that-beats-me-at-tic-tac-toe-3ea6ba22cd71>

Mathieu Blanchette. Défi de codage. Youtube. 11 Décembre 2019 [Consulté en Février 2020]. Disponible sur : <https://www.youtube.com/watch?v=trKjYdBASyQ>

Contributeurs de Wikipédia. Algorithme minimax. Wikipédia, l'encyclopédie libre. 21 Janvier 2020 [Consulté en Avril 2020]. Disponible sur : https://fr.wikipedia.org/wiki/Algorithme_minimax

Chapitre 10

Annexe

Voici le code de notre programme dans son intégralité :

```
1 import sys
2
3 def affiche(etat):
4     i = 0
5     while i < len(etat) :
6         j = 0
7         while j < len(etat[i]) :
8             print(etat[i][j], end="")
9             j += 1
10            print()
11            i += 1
12        print()
13
14 def coupHumain(etat, typeJeu) :
15     dim = len(etat)
16     while True :
17         abs = input("Ligne (entre 0 et "+str(dim-1)+" ) ? ")
18         ord = input("Colonne (entre 0 et "+str(dim-1)+" ) ? ")
19         if abs.isdigit() and int(abs) >= 0 and int(abs) < dim and ord.isdigit() and int(
20             ord) >= 0 and int(ord) < dim and etat[int(abs)][int(ord)] == 0 :
21             if typeJeu == "p4" :
22                 if gravite(etat, int(abs), int(ord)) :
23                     etat[int(abs)][int(ord)] = 2
24                     return (int(abs), int(ord))
25                 else :
26                     print("Coup impossible ( gravit ) !")
27             else :
28                 etat[int(abs)][int(ord)] = 2
29             return (int(abs), int(ord))
30         print("Coup impossible ( position ) !")
31
32 def estGagnant(etat, nbPions, joueur, dernierCoup):
33     ligne = dernierCoup[0]
34     colonne = dernierCoup[1]
35     dim = len(etat)
36     pion = 1
37     if joueur == "H" : pion = 2
38     nbPionsAlignes = 0
39     j = 0
40     while j < dim :
```

```

41         if etat[ligne][j] == pion :
42             nbPionsAlignes += 1
43             if nbPionsAlignes == nbPions : return True
44         else : nbPionsAlignes = 0
45         j += 1
46     nbPionsAlignes = 0
47     i = 0
48     while i < dim :
49         if etat[i][colonne] == pion :
50             nbPionsAlignes += 1
51             if nbPionsAlignes == nbPions : return True
52         else : nbPionsAlignes = 0
53         i += 1
54
55     nbPionsAlignes = 0
56     lignetest = ligne
57     colonnetest = colonne
58     while lignetest > 0 and colonnetest > 0 :
59         lignetest -= 1
60         colonnetest -= 1
61     while lignetest < dim and colonnetest < dim :
62         if etat[lignetest][colonnetest] == pion :
63             nbPionsAlignes += 1
64             if nbPionsAlignes == nbPions : return True
65         else : nbPionsAlignes = 0
66         lignetest += 1
67         colonnetest += 1
68
69     nbPionsAlignes = 0
70     lignetest = ligne
71     colonnetest = colonne
72     while lignetest < dim-1 and colonnetest > 0 :
73         lignetest += 1
74         colonnetest -= 1
75     while lignetest >= 0 and colonnetest < dim :
76         if etat[lignetest][colonnetest] == pion :
77             nbPionsAlignes += 1
78             if nbPionsAlignes == nbPions : return True
79         else : nbPionsAlignes = 0
80         lignetest -= 1
81         colonnetest += 1
82
83
84     return False
85
86 def gravite(etat , i , j) :
87     if i == len(etat)-1 :
88         return True
89     if etat[i+1][j] != 0 :
90         return True
91     return False
92
93 def possibles(etat):
94     dim = len(etat)
95     possibles = []
96     i = 0
97     while i < dim :
98         j = 0
99         while j < len(etat[i]) :
100             if etat[i][j] == 0 :
101                 if typeJeu == "p4" :
102                     if gravite(etat , i , j) :
103                         possibles.append([i,j])
104                 else :

```

```

105             possibles.append([i,j])
106             j += 1
107             i += 1
108         return possibles
109
110     def min(scores) :
111         scoreMin = scores[0]
112         numCoupMin = 0
113         numCoup = 1
114         while numCoup < len(scores) :
115             if scores[numCoup] < scoreMin :
116                 scoreMin = scores[numCoup]
117                 numCoupMin = numCoup
118             numCoup += 1
119         return numCoupMin
120
121     def max(scores) :
122         scoreMax = scores[0]
123         numCoupMax = 0
124         numCoup = 1
125         while numCoup < len(scores) :
126             if scores[numCoup] > scoreMax :
127                 scoreMax = scores[numCoup]
128                 numCoupMax = numCoup
129             numCoup += 1
130         return numCoupMax
131
132     def explore(etat, nbPions, profondeur, joueur) :
133         coupsPossibles = possibles(etat)
134         nbCoupPossibles = len(coupsPossibles)
135         autreJoueur = "H"
136         if joueur == "H" : autreJoueur = "IA"
137         scores = []
138         numCoup = 0
139         for coup in coupsPossibles :
140             if joueur == "IA" : etat[coup[0]][coup[1]] = 1
141             else : etat[coup[0]][coup[1]] = 2
142             if estGagnant(etat, nbPions, joueur, coup) == True :
143                 if joueur == "IA" :
144                     scores.append(1)
145                     etat[coup[0]][coup[1]] = 0
146                     break
147                 else :
148                     scores.append(-1)
149                     etat[coup[0]][coup[1]] = 0
150                     break
151             else :
152                 if nbCoupPossibles == 1 :
153                     scores.append(0)
154                 else :
155                     scores.append(explore(etat, nbPions, profondeur+1, autreJoueur))
156                     etat[coup[0]][coup[1]] = 0
157                     numCoup += 1
158         if joueur == "H" :
159             numCoup = min(scores)
160             if profondeur == 0 :
161                 return coupsPossibles[numCoup]
162             return scores[numCoup]
163         if joueur == "IA" :
164             numCoup = max(scores)
165             if profondeur == 0 :
166                 return coupsPossibles[numCoup]
167             return scores[numCoup]
168

```

```

169 if len(sys.argv) != 4 :
170     print("Usage : ttt_ou_p4.py <ttt ou p4> <dimension> <nb pions aligner>")
171     exit()
172 if sys.argv[1] != "ttt" and sys.argv[1] != "p4" :
173     print("Le premier param tre doit tre ttt (tic-tact-toe) ou p4 (puissance4)")
174     exit()
175 if not sys.argv[2].isdigit() or not sys.argv[3].isdigit() :
176     print("La dimension et le nombre des pions doivent tre des entiers")
177     exit()
178
179 typeJeu = "ttt"
180 if sys.argv[1] == "p4" :
181     typeJeu = "p4"
182 dimension = int(sys.argv[2])
183 nbPions = int(sys.argv[3])
184 etat = [[] for i in range(0, dimension)]
185 for i in range(0, dimension) :
186     for j in range(0, dimension) :
187         etat[i].append(0)
188 print(typeJeu)
189 print(etat)
190
191 # Boucle de jeu
192 while True :
193
194     print("L'IA r fl chit ...")
195     meilleurCoup = explore(etat, nbPions, 0, "IA")
196     etat[meilleurCoup[0]][meilleurCoup[1]] = 1
197     affiche(etat)
198     if estGagnant(etat, nbPions, "IA", meilleurCoup) == True :
199         print("Victoire de l'IA")
200         exit()
201     if possibles(etat) == [] :
202         print("Grille pleine et aucun gagnant !")
203         exit()
204     coupH = coupHumain(etat, typeJeu)
205     affiche(etat)
206     if estGagnant(etat, nbPions, "H", coupH) == True:
207         print("Victoire de l'humain")
208         exit()

```