Projet DIA

**Compte rendu : Ultimate Tic-Tac-Toe**

*Sarah AIT SAID – Sasha ROMMELFANGEN – Aymeric CHESTERIKOFF – Diana PEREIRA*

*TD J*

*Le* ***tic-tac-toe****, aussi appelé «****morpion »*** *est un* [*jeu*](https://fr.wikipedia.org/wiki/Jeu) *de réflexion se pratiquant à deux joueurs, tour par tour, dont le but est de créer le premier un alignement. L’****ultimate tic-tac-toe*** *est une variation combinatoire du jeu de morpion.*

**Objectif du code :**

L’objectif principal de notre code est la création d'une version jouable du jeu de Tic-Tac-Toe utilisant l'algorithme MiniMax avec élagage alpha-bêta pour la prise de décision de l'intelligence artificielle. Pour atteindre cet objectif, nous avons défini les étapes suivantes :

1. **Création du jeu de Tic-Tac-Toe :**

* Permettre aux joueurs humains de jouer contre l'intelligence artificielle dans une grille de Tic-Tac-Toe standard.
* Prendre en charge les mouvements des joueurs, la vérification des conditions de victoire et la gestion des matchs nuls.
* Afficher l'état actuel de la grille dans une interface utilisateur conviviale et fournir des indications sur les mouvements possibles.

1. **Mise en œuvre de l'algorithme MiniMax avec élagage alpha-bêta :**

* Permettre à l'intelligence artificielle de prendre des décisions éclairées sur le prochain mouvement à jouer.
* Explorer les différentes possibilités de mouvement dans l'arbre de jeu jusqu'à une certaine profondeur.
* Attribuer une valeur à chaque état de jeu en utilisant l'algorithme MiniMax.
* Utiliser l'élagage alpha-bêta pour réduire le nombre de branches explorées en éliminant les mouvements non pertinents.
* Permettre à l'intelligence artificielle de choisir le mouvement qui maximise sa valeur de gain potentielle ou minimise la valeur de gain du joueur adverse.

1. **Implémentation de la récursivité et évaluation des états de jeu :**

* Utiliser un algorithme récursif pour explorer les différents mouvements possibles jusqu'à une certaine profondeur.
* Attribuer une valeur aux états de jeu terminaux, tels que la victoire, la défaite ou le match nul.
* Évaluer les états de jeu non terminaux en utilisant la récursion et les valeurs attribuées aux états terminaux.

En résumé, notre objectif était de créer une version jouable du Tic-Tac-Toe avec une intelligence artificielle capable de prendre des décisions stratégiques en utilisant l'algorithme MiniMax avec élagage alpha-bêta. Nous avons réalisé cela en implémentant les fonctionnalités de jeu, l'algorithme de prise de décision et en évaluant les différents états de jeu.

**Structure et choix de code :**

Le code est structuré de manière à effectuer le jeu du morpion avec une intelligence artificielle utilisant l'algorithme MiniMax avec élagage alpha-bêta pour déterminer le meilleur coup à jouer. Voici une description de la structure du code :

Nous avons commencé tout d’abord par la définition de la constante `TIME\_LIMIT` définie dans le but de limiter le temps d'exécution de l'algorithme afin que l’intelligence artificielle ne mette pas trop de temps à choisir le coup à jouer.

Ensuite nous avons défini les fonctions utilitaires suivantes :

* **Index (x, y) :** calcule l'index correspondant aux coordonnées (x, y) d'une case dans la grille 9x9. Elle est utilisée pour convertir les coordonnées d'une case en un index unique.
* **Box(x, y) :** renvoie le numéro du sous-morpion auquel appartient la case avec les coordonnées (x, y). Chaque grille 9x9 est divisée en neuf sous-morpions de taille 3x3. Cette fonction permet de déterminer dans quel sous-morpion se trouve une case spécifique.
* **Next(i) :** renvoie l'index du prochain sous-morpion à jouer, en fonction de l'index actuel i. Elle est utilisée pour déterminer quel sous-morpion sera le prochain à être joué, en se basant sur l'index de celui qui vient d'être joué.
* **Indices(b) :** renvoie la liste des indices des cases dans le sous-morpion b. Elle est utilisée pour obtenir les indices des cases qui appartiennent à un sous-morpion spécifique.
* **Board(state) :** affiche l'état actuel de la grille de jeu. Elle prend en compte l'état du jeu représenté par la variable *state* et l'affiche sous forme de grille pour que les joueurs puissent voir l'état actuel de la partie.
* **Result (state, move, player) :** renvoie l'état de la grille après avoir effectué le mouvement du joueur. Elle est utilisée pour mettre à jour l'état du jeu après qu'un joueur ait effectué un mouvement.
* **Check\_Small\_Box(box\_str) :** vérifie si l'un des joueurs a gagné le sous-morpion. Elle prend en compte l'état d'un sous-morpion représenté par la variable *box\_str* et vérifie s'il y a une combinaison gagnante.
* **Terminal\_Test(state) :** vérifie si l'un des joueurs a gagné un sous-morpion ou si la grille de jeu est pleine. Elle est utilisée pour déterminer si la partie est terminée.
* **Actions(last\_move) :** renvoie les indices des cases dans le sous-morpion suivant à jouer, en fonction du dernier mouvement effectué. Elle est utilisée pour obtenir les mouvements valides dans le sous-morpion suivant, en fonction du dernier mouvement joué.
* **Successeurs(state, player, last\_move) :** génère une liste de tous les états possibles après le prochain coup pour le joueur. Elle est utilisée pour générer tous les coups disponibles afin d'explorer les différentes options dans l'algorithme *MiniMax*.
* **AfficheSuccessors(state, player, last\_move) :** affiche les successeurs possibles de l'état actuel du jeu pour le joueur. Elle est utilisée pour afficher les différents mouvements possibles par un joueur.
* **Opponent(p) :** renvoie le joueur adverse.
* **Evaluate\_Small\_Box(box\_str, player) :** évalue l'état d'un sous-morpion en attribuant un score en fonction des combinaisons possibles de victoire.
* **Evaluate(state, last\_move, player) :** évalue l'état global du jeu en attribuant un score en fonction de l'état de chaque sous-morpion. Elle combine les évaluations des sous-morpions pour fournir une évaluation globale de la position du joueur dans la partie.
* **MiniMax(state, last\_move, player, depth, s\_time) :** implémente l'algorithme MiniMax avec élagage alpha-bêta pour déterminer le meilleur coup à jouer.
* **Tour\_Max(state, last\_move, player, depth, s\_time, alpha, beta) :** fonction auxiliaire pour l'algorithme MiniMax, représentant le tour du joueur MAX. Elle explore les positions possibles pour le joueur MAX et met à jour les valeurs d'alpha et de beta pour effectuer l'élagage alpha-bêta.
* **Tour\_Min(state, last\_move, player, depth, s\_time, alpha, beta) :** fonction auxiliaire pour l'algorithme MiniMax, représentant le tour du joueur MIN. Elle explore les positions possibles pour le joueur MIN et met à jour les valeur alpha et beta pour effectuer l'élagage alpha-bêta.
* **Sortie\_Valide(state, move)** : vérifie si un mouvement est valide dans l'état actuel du jeu. Elle prend en compte l'état actuel du jeu (**state**) ainsi que le mouvement à vérifier (**move**), représenté par un tuple (ligne, colonne).
* **Prendre\_Entree(state, bot\_move)**: permet à l'utilisateur de saisir son mouvement. Elle prend en compte l'état actuel du jeu (**state**) ainsi que le dernier mouvement effectué par l'IA (**bot\_move**). Elle restreint les mouvements possibles si un sous-morpion a déjà été remporté.
* **Demander\_Choix\_Joueur()**: demande à l'utilisateur s'il veut commencer le jeu. Elle renvoie **True** si l'utilisateur choisit de commencer ou **False** sinon.
* **Play(state="." \* 81, depth=20)**: gère le déroulement du jeu. Elle prend en compte l'état initial du jeu (**state**) et la profondeur de recherche pour l'algorithme MiniMax (**depth**). Elle utilise les autres fonctions pour afficher le plateau de jeu, demander les mouvements des joueurs, effectuer les mouvements, vérifier l'état du jeu et déterminer le gagnant.

Nous avons également fait le choix de représentation de la grille. Nous avons tout d’abord essayé de faire une représentation avec des matrices mais cela compliquait l’implémentation de l’algorithme. Nous avons finalement représenté la grille dans notre code par une liste linéaire, où chaque élément de la liste correspond à l’état d’une case qui contient des caractères vides représentant des cases non remplies. Cette représentation permet de simplifier la manipulation de l’état du jeu et d’accéder facilement à chaque case en utilisant simplement des indices.

Pour évaluer le jeu, nous avons fait en sorte que l’IA évalue l’état global du jeu en premier lieu en combinant les évaluations des sous-morpions et ensuite elle utilise cela pour attribuer un score global qui reflète la position du joueur dans la partie.

Nous avons également fait le choix de choisir le sous morpion dans lequel on va jouer lorsque le sous-morpion imposé initialement par la règle du jeu est déjà gagné ou perdu.

**Difficultés rencontrées :**

Nous avons rencontré quelques difficultés telles que :

* Faire en sorte que l’IA joue dans le bon sous-morpion selon le coup du joueur ;
* Déterminer un gagnant sur une échelle 9x9 en plus du gagnant à l’échelle 3x3 ;
* S’assurer que l’IA ne joue pas aléatoirement ce qui implique de rajouter une heuristique ;
* Le temps de prise de décision de l’IA : elle devient moins performante à la fin du jeu, et quand la partie dure trop longtemps (lorsque qu’on force l’IA a jouer dans un sous morpion déjà gagné ou perdu notamment), l’IA met plus de temps à prendre une décision, ce qui implique une crainte sur la possibilité de dépasser le temps imparti limiter à 10 secondes.

Nous avons pu les résoudre par l’utilisation de l’algorithme avec élagage alpha-bêta qui a donc permis à l’IA de rechercher de manière efficace les meilleurs coups possibles, en explorant l’arbre des possibilités et en évitant de considérer les branches les moins prometteuses. Cela permet à l’IA de prendre des décisions éclairées sur les coups à jouer et d’adapter sa stratégie en fonction de l’état actuel du jeu. L’IA cherche ainsi à maximiser son score et à minimiser celui de l’adversaire.

**Conclusion :**

En conclusion, nous avons réussi à mettre en œuvre les fonctionnalités du jeu comme nous le souhaitions. Cela nous a donc permis d’aboutir à la création d’une version jouable du morpion accompagnée de l’intelligence artificielle compétitive imposée. Grace à ce projet, nous avons donc acquis la capacité de mettre en pratique l’algorithme MiniMax et de voir une application concrète de ce dernier. Cette réalisation nous a donc non seulement permis de renforcer nos compétences en programmation mais également d’optimiser notre aptitude à résoudre des problèmes complexes.