

1. En utilisant les méthodes `legal_moves()` et `is_game_over()`, proposer une méthode permettant d'explorer toutes les parties possibles au Morpion (lorsque X commence). Combien y-a-t-il de parties ? Combien votre arbre de recherche a-t-il de noeuds ? Combien de temps faut-il pour tout explorer ?
2. Sans utiliser à ce stade d'horizon maximal à votre recherche arborescente, recherchez s'il existe une stratégie gagnante au Morpion (pour ce faire, vous n'aurez à prendre en compte que l'information "Gagné" / "Perdu" / "Egalité obtenue à chaque fin de partie.")
3. Améliorez votre recherche de stratégie gagnante en effectuant votre première recherche "intelligente", qui sera capable d'élaguer des parties de l'arbre de recherche tout en conservant l'admissibilité de votre recherche. Comparez les temps et le nombre de noeuds nécessaires à la recherche.

3 Du Morpion aux échecs

Pour jouer aux échecs, nous utiliserons une librairie Python : `python-chess`, disponible à l'adresse <https://pypi.org/project/python-chess/>. Installez la librairie avec `pip` ou téléchargez l'archive et installez les fichiers dans votre répertoire de travail.

Copiez également le fichier `starter-chess.py` disponible depuis le site du cours. Il vous montre comment dérouler une partie aléatoire sur le jeu d'échec.

1. Faites une recherche exhaustive de toutes les parties d'échec, mais en limitant la profondeur de la recherche par un paramètre de la recherche. Jusqu'à quelle profondeur pouvez vous aller en moins de 30s ? Combien de noeuds votre recherche explore-t-elle à profondeur 1, 2, 3, ... ?
2. Nous allons devoir fixer un horizon à nos recherches. Pour cela, il nous faut définir une heuristique pour le plateau d'échec. Codez l'heuristique proposée par Claude Shannon en 1950 et vue en cours. Vous n'utiliserez que la partie de l'heuristique donnant un poids aux pièces de jeu. Ajoutez un moyen d'exprimer qu'il est préférable d'avancer ses pions pour les mener éventuellement à la Reine (**aide** : pour parcourir le plateau de jeu, vous pourrez utiliser la méthode `board.pieces_map()` tout en récupérant le caractère symbolisant la pièce grâce à la méthode `symbol()` offerte par la librairie).
3. Codez un Minimax sur les échecs. Attention, les algorithmes vus en cours ne se préoccupent pas de renvoyer le coup associé au meilleur choix : ils ne renvoient que la valeur minimax de l'arbre. Il faudra donc coder une version spéciale de `MaxMin(...)` pour le niveau 1.
4. Codez un match Joueur Aléatoire contre Minimax niveau 3. Puis Minimax niveau 1 contre Minimax niveau 3. Si vous remarquez que vos joueurs joeunt en boucle, ajoutez un mécanisme qui permet de choisir le coup au hasard parmi les ex-aequo, si besoin.

4 L'alpha et l'oméga de $\alpha - \beta$

1. Faites évoluer votre code Minimax en $\alpha - \beta$. Comparez les temps de recherche et le nombre de noeuds explorés sur plusieurs plateaux de jeu entre les deux méthodes, à profondeur égale. Faites un match Minimax contre $\alpha - \beta$ à profondeur égales.
2. Encapsulez votre $\alpha - \beta$ dans un Iterative Deepening pour garantir que votre recherche ne dépassera jamais 10s de calcul.
3. Améliorez votre heuristique en prenant en compte les informations sur les prises fournies par la librairie.
4. Fabriquez une bibliothèque d'ouverture pour le premier coup Blancs et le premier coup Noirs.
5. Programmez un joueur humain (qui demande au clavier le coup à jouer). Battez votre IA (ou pas).