Second projet de l'année qui va vous permettre d'utiliser à nouveau la programmation orientée objet et les arbres ainsi que leur parcours.
Ce projet sera à réaliser par groupe de trois (quatre pour un groupe).
Vous aurez la possibilité de choisir par groupe un des trois projets ci-dessous :

1 Proposition 1 : Morpion sur Pygame + IA

L'an passé vous aviez programmé (pour certains) un jeu de morpion jouable sur la console `Pyzo`.
Nous allons vous rendre disponible la version du corrigé que nous vous avions proposé en jeu à deux.

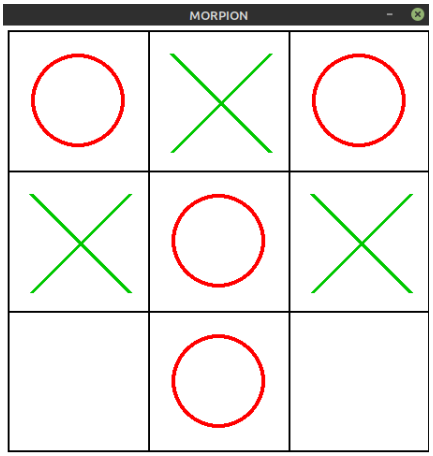
1.1 Morpion sur Pygame avec POO

Vous allez devoir convertir cette version du corrigé un une version en POO.
Vous pourrez faire vos choix sur les classes qui interviendront (éventuellement une classe Joueur en plus ?) mais vous devrez avoir au moins deux classes :

- la classe `Grille` qui gère l'affichage de la grille et l'évolution de la partie ;
- la classe `Jeu` qui gère l'évolution de l'affichage sur `Pygame`.

Votre jeu devra intégrer :

- un affichage graphique grâce à la bibliothèque Pygame.
  Vous pourrez chercher à obtenir un écran semblable à celui-ci :



- un mode partie à deux.
- un mode solo, avec deux niveaux de jeu (avec coup aléatoire et coup intelligent). Pour ce point là nous allons vous donner quelque détails supplémentaires dans la partie 1.2
- un compteur de points (si le ou les joueur(s) font plusieurs parties).
- un écran de sélection du mode de jeu.

Votre jeu devra gérer correctement les fins de parties.

1.2 Développement d'un mode solo avec niveau de difficulté

Comme précisé dans la partie 1.1 votre jeu devra comprendre différents niveau de jeu.

- un niveau facile avec une IA qui joue un coup aléatoire ;
- un niveau moyen avec une IA qui utilisera l'algorithme Minimax avec une profondeur 1 (ou 2) ;
- un niveau difficile avec une IA qui utilisera l'algorithme Minimax avec une profondeur 3 (ou 4).

1.3 Bonus : Compléments

Vous pourrez ensuite améliorer votre jeu en intégrant des écrans d'accueil, de fin de partie, de relance de partie, une musique de fond...

2 Proposition 2 : Puissance 4 sur Pygame + IA

L'an passé vous aviez programmé (pour certains) un jeu de puissance 4 jouable sur la console `Pyzo`.
Nous allons vous rendre disponible la version du corrigé que nous vous avions proposé en jeu à deux.

2.1 Puissance 4 sur Pygame avec POO

Vous allez devoir convertir cette version du corrigé un une version en POO.
Vous pourrez faire vos choix sur les classes qui interviendront (éventuellement une classe Joueur en plus ?) mais vous devrez avoir au moins deux classes :

- la classe `Grille` qui gère l'affichage de la grille et l'évolution de la partie ;
- la classe `Jeu` qui gère l'évolution de l'affichage sur `Pygame`.

Votre jeu devra intégrer :

- un affichage graphique sur Pygame ;
- un mode partie à deux ;
- un mode solo, avec deux niveaux de jeu (avec coup aléatoire et coup intelligent), pour ce point là nous allons vous donner quelque détails supplémentaires dans la partie 2.2 ;
- un compteur de points (si le ou les joueur(s) font plusieurs parties) ;
- un écran de sélection du mode de jeu.

Votre jeu devra gérer correctement les fins de parties.

2.2 Développement d'un mode solo avec niveau de difficulté

Comme précisé dans la partie 2.1 votre jeu devra comprendre différents niveau de jeu.

- un niveau facile avec une IA qui joue un coup aléatoire ;
- un niveau moyen avec une IA qui utilisera l'algorithme Minimax avec une profondeur 1 (ou 2) ;
- un niveau difficile avec une IA qui utilisera l'algorithme Minimax avec une profondeur 3 (ou 4 ou plus).

2.3 Bonus : Compléments

Vous pourrez ensuite améliorer votre jeu en intégrant des écrans d'accueil, de fin de partie, de relance de partie, une musique de fond...

3 Proposition 3 : Jeu de Nim sur Pygame + IA

L'an passé vous n'aviez pas programmé le jeu de Nim. Nous n'allons donc pas vous rendre disponible la version du corrigé que nous vous avions proposé en jeu à deux.

3.1 Jeu de Nim sur Pygame avec POO

Vous devrez donc coder entièrement le jeu de Nim en réfléchissant au découpage en fonctions de ce jeu.
Vous pourrez faire vos choix sur les classes qui interviendront (éventuellement une classe Joueur en plus ?) mais vous devez avoir au moins deux classes :

- la classe `Grille` qui gère l'affichage de la grille et l'évolution de la partie ;
- la classe `Jeu` qui gère l'évolution de l'affichage sur `Pygame`.

Votre jeu devra intégrer :

- un affichage graphique sur Pygame ;
- un mode partie à deux ;
- un mode solo, avec deux niveaux de jeu (avec coup aléatoire et coup intelligent), pour ce point là nous allons vous donner quelque détails supplémentaires dans la partie 3.2 ;
- un compteur de points (si le ou les joueur(s) font plusieurs parties).
- un écran de sélection du mode de jeu.

Votre jeu devra gérer correctement les fins de parties.

3.2 Développement d'un mode solo avec niveau de difficulté

Comme précisé dans la partie 3.1 votre jeu devra comprendre différents niveau de jeu :

- un niveau facile avec une IA qui joue un coup aléatoire ;
- un niveau moyen avec une IA qui utilisera l'algorithme Minimax avec une profondeur 1 (ou 2) ;
- un niveau difficile avec une IA qui utilisera l'algorithme Minimax avec une profondeur 3 (ou 4).

3.3 Bonus : Compléments

Vous pourrez ensuite améliorer votre jeu en intégrant des écrans d'accueil, de fin de partie, de relance de partie, une musique de fond...

4 L'algorithme et le principe de MiniMax :

Les informations qui suivent proviennent pour leur grande majorité des page web située ici ou ici.

4.1 Le principe

L'application de l'algorithme MiniMax s'avère utile lorsque l'on s'intéresse aux jeux asynchrones opposant deux joueurs, où chacun joue à son tour, à information complète, où chacun sait tout de la situation de jeu, à chaque instant.
Dans la suite, on note J_1 et J_2 les deux joueurs et l'on se place dans la situation c'est à J_1 de jouer.
Le déroulement de la partie peut être vu comme un arbre :

- la racine correspond à l'état actuel du jeu ;
- les noeuds à profondeur paire correspondent aux noeuds où J_1 doit jouer, à profondeur impaire les noeuds où c'est à J_2 ;

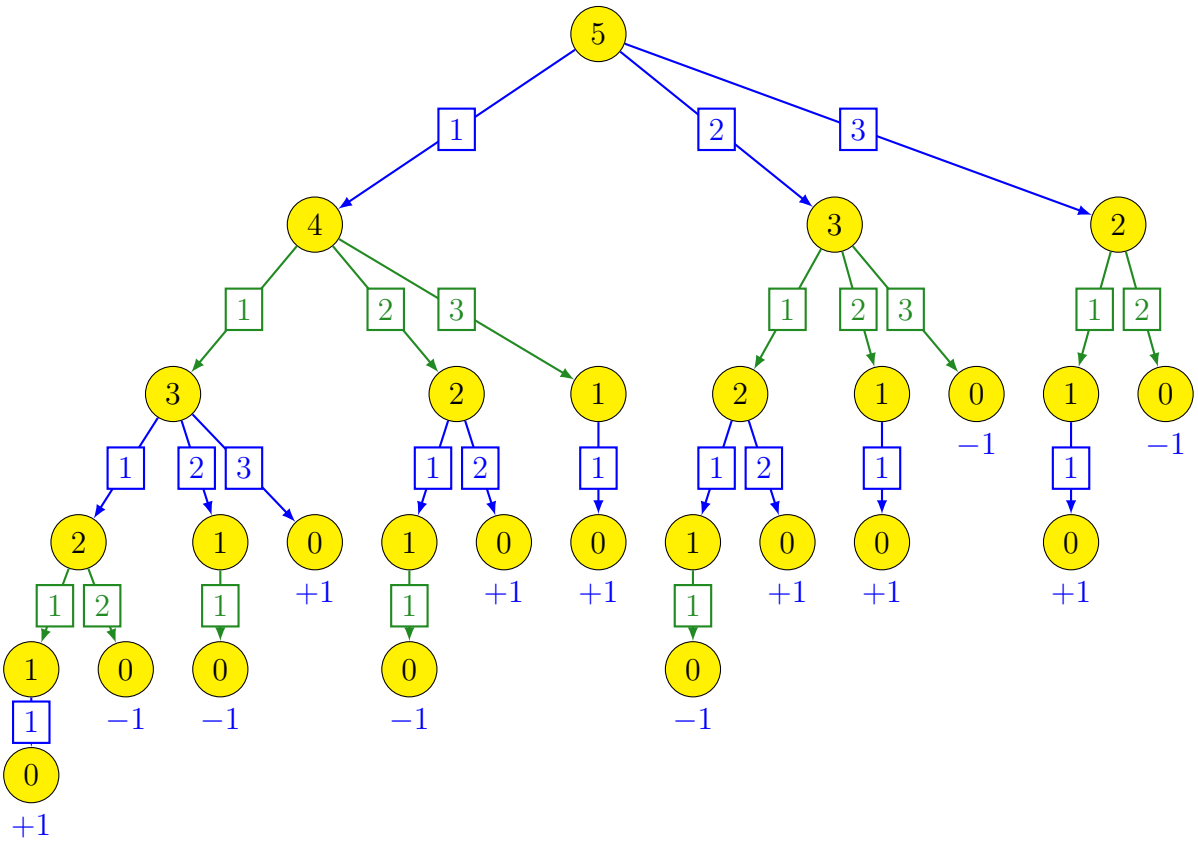
— chaque arête correspond à un coup joué et lie un état où ce coup est autorisé à l'état résultant de l'application du coup ;
— au niveau des feuilles, nous trouvons les fins de partie : les états gagnants ou perdants pour J_1, ou encore les états bloqués, matches nuls ;
— un chemin allant de la racine à une feuille décrit une partie possible.

On prendra la convention suivante pour démarrer :
— une situation gagnante pour J_1 vaut $+1$;
— une situation perdante pour J_1 vaut -1 ;
— une situation nulle vaut 0.

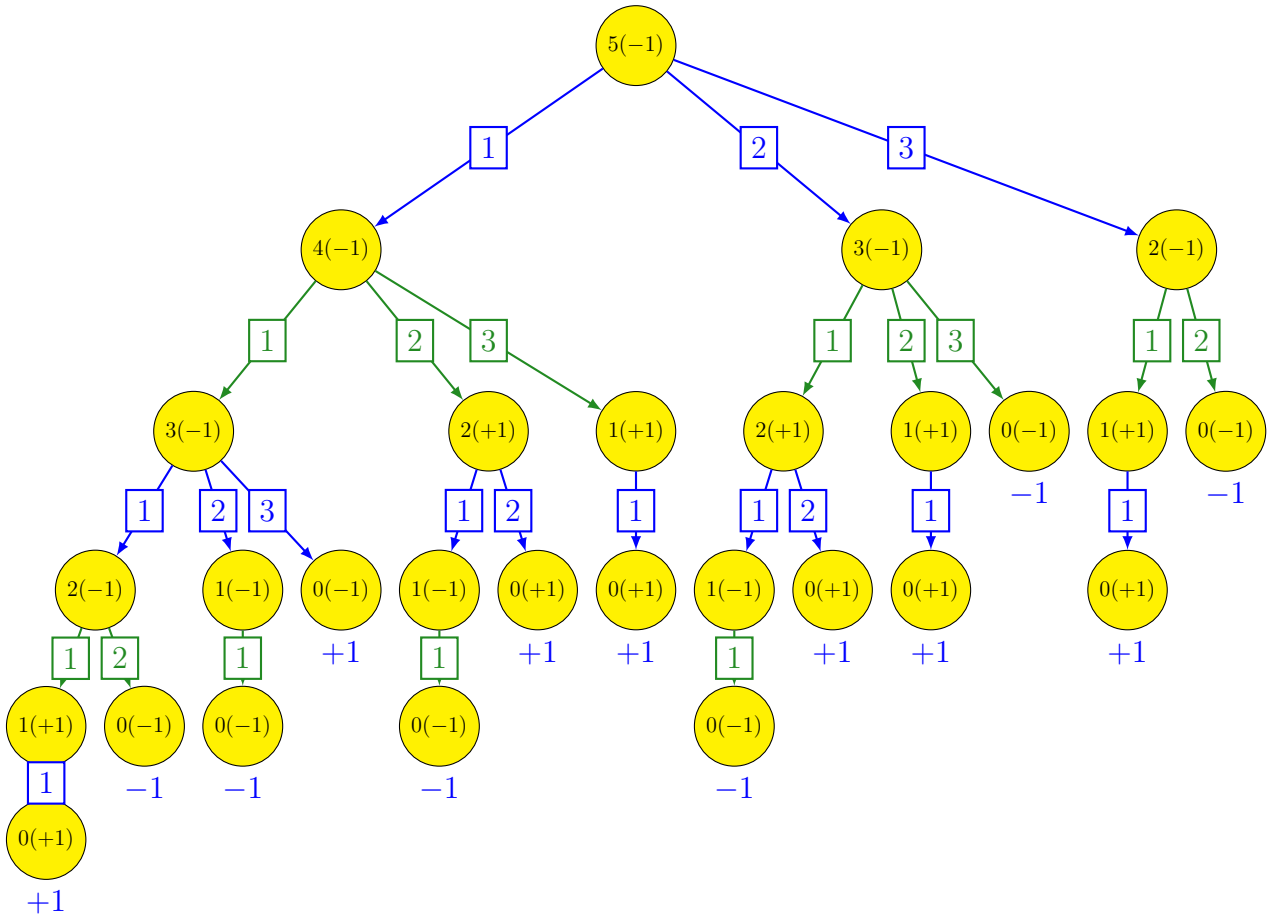
L'idée de l'algorithme est de développer complètement l'arbre de jeu, de noter chaque feuille avec sa valeur, puis de faire remonter ces valeurs avec l'hypothèse que chaque joueur choisit le meilleur coup pour lui. Cela signifie en pratique que :
— J_1 choisit le coup amenant à l'état de plus grande valeur ;
— J_2 choisit le coup amenant à l'état de plus petite valeur.

Voici deux exemples d'utilisation :

- On applique le principe du MiniMax Jeu de Nim à 5 allumettes (chaque joueur peut à chaque coup prendre 1, 2 ou 3 allumettes et celui qui prend la dernière allumette a perdu).
  On commence par développer l'arbre de jeu et on étiquette les feuilles selon qu'il s'agit d'une partie gagnante ou perdante :



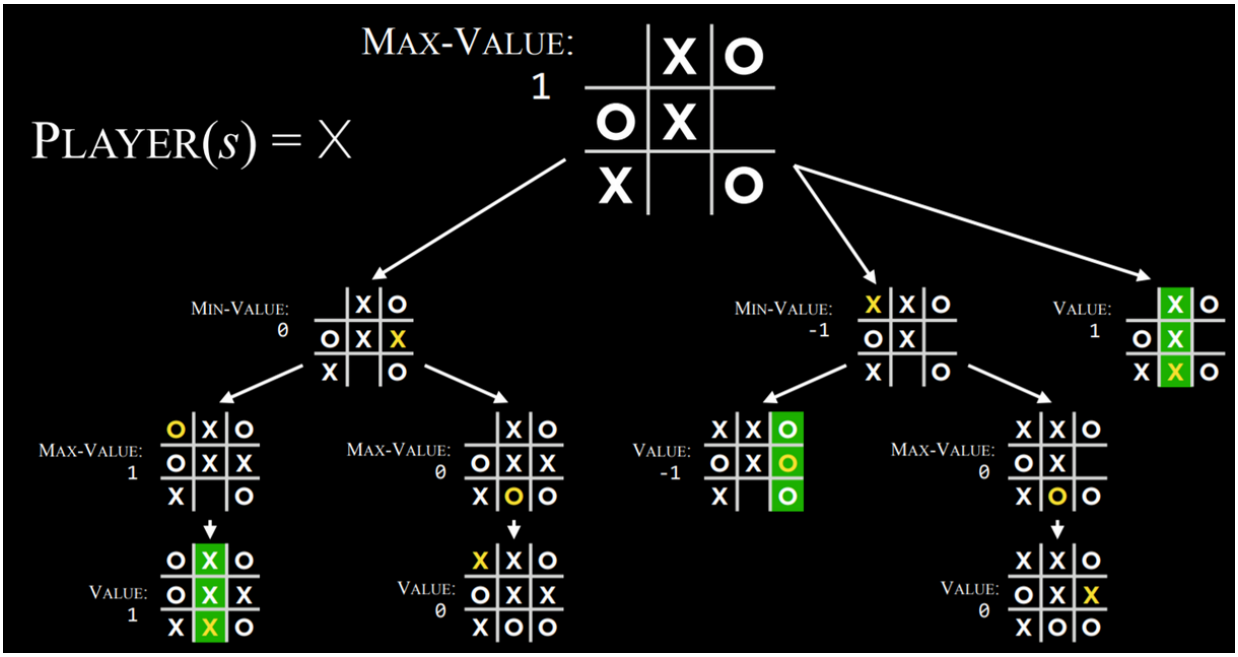
  Ensuite, on fait remonter les évaluations depuis les feuilles jusqu'à la racine :
  — en bleu les arêtes correspondant aux coups joués par J1 où l'on maximise en « remontant » ;
  — en vert les arêtes correspondant aux coups joués par J2 où l'on minimise en « remontant » ;



Si on détermine la valeur MiniMax pour la situation avec 5 allumettes sur la table et le tour du joueur J_1, on va trouver -1, donc le joueur J_1 va perdre la partie si le joueur J_2 joue de façon logique. Les valeurs associées à tous les noeuds J_2 successeurs de la racine sont égales à -1, donc quel que soit le coup appliqué par J_1, on va arriver dans une position gagnante pour le joueur J_2.
En effet, si J_1 retire 1 allumette, J_2 en retire à son tour 3, si J_1 en retire 2, J_2 en retire 2 aussi et si J_1 en retire 3, J_2 va en retirer 1, et dans toutes ces situations J_1 va trouver sur la table 1 allumette qu'il sera forcé de prendre, donc il perdra la partie.
On constate donc que la valeur MiniMax qui sera déterminée ne peut pas garantir la victoire pour le joueur J_1, mais le conduira toujours dans la meilleure situation possible pour la position donnée.

Question-Exercice :
Que ce passe-t-il s'il reste 3 allumettes sur la table et que c'est au joueur J_1 de jouer ? Construire l'arbre correspondant (comme précédemment) puis déterminer le coup que devra jouer J_1.

- Un deuxième exemple avec le morpion. (lien vers l'image ci-dessous)



4.2 Évolution à mettre en oeuvre :

En pratique, l'algorithme MiniMax n'est pas utilisé en l'état : il est rarement possible de développer l'ensemble de l'arbre de jeu, sauf si l'on est en fin de partie. Nous allons donc devoir limiter la profondeur d'exploration (on pourrait de plus envisager d'effectuer des coupes dans l'arbre de jeu).
Suivant la profondeur envisagée, le niveau de difficulté du jeu évolue (plus on descend profondément dans l'arbre plus l'IA sera « forte »).
Interrompre l'exploration de l'arbre avant d'avoir atteint les fins de partie implique de pouvoir évaluer l'état du jeu à un instant donné.
Pour cela, nous allez devoir (le point difficile) mettre en place une fonction d'évaluation du jeu pour une situation donnée.
Les résultats de l'algorithme Minimax sont donc liés à la qualité de la fonction d'évaluation du jeu et au nombre de coups d'avance que l'on autorise dans le développement de l'arbre.
La fonction d'évaluation devra être redéfinie pour chaque jeu, en y incluant toute la connaissance possible sur le jeu (sans que la fonction soit trop complexe, sinon le temps pour jouer un coup sera trop long).
La profondeur maximale est souvent liée à la machine utilisée mais également à l'avancement dans la partie (on peut souvent aller plus profond en fin de partie).
Remarque : Le niveau des joueurs artificiels d'échecs en fonction de cette profondeur limite (ici un coup signifie en fait un échange, l'action de J_1 et la réplique de J_2) donne les niveaux suivant :

- 2 coups : novice ;
- 4 coups : maître ;
- 6-7 coups : champion du monde.

4.3 Mise en oeuvre dans votre projet :

Les méthodes dont vous aurez besoin (la liste n'est pas exhaustive) :

— Une méthode `fonction_eval_score()` qui évalue le jeu. Plus le score est élevé et plus le joueur a intérêt à jouer le coup proposé. Exemples de mise en oeuvre (vous aurez à faire des choix...) :
  - Pour le puissance 4, par exemple, on examine chaque possibilité de réalisation d'un alignement victorieux, si il y a juste un jeton du joueur 1 on gagne 1 point, si il y juste deux jetons on gagne 3 points et si il y a en a 3 on gagne 10 points. Les points sont négatifs si les jetons sont pour le joueur 2. Si on trouve une victoire du joueur 1 on retourne un grand nombre, si le joueur 2 gagne on retourne un grand nombre négatif...
  - Pour le morpion, par exemple, on examine chaque possibilité d'alignement victorieux, si le joueur 1 a un pion (dans la ligne, colonne ou diagonale) on attribue 1 point, -1 si c'est le joueur 2. S'il y a 2 pions pour le joueur 1, on attribue 3 points ou -3 si c'est le joueur 2. Enfin si le joueur 1 a 3 pions alignés, on attribue 100 points ou -100 si c'est pour le joueur 2...
— Une méthode `donne_coup()` qui donne tous les coups possibles (par exemple sous forme d'un tableau pour le puissance 4, [3,4,6] signifie que l'on peut joueur aux colonnes 3 4 et 6.
— Une méthode `joue_coup(coup)` qui joue un coup (vous l'avez déjà pour le morpion et le puissance 4).
— Une méthode `annule_coup(coup)` qui annule le coup joué.
— Une méthode qui vérifie la victoire d'un joueur (vous l'avez déjà pour le morpion et le puissance 4).
— Une méthode qui vérifie le match nul (vous l'avez déjà pour le morpion et le puissance 4).
— ... toute méthode qui vous sera utile.