# TP 6 : Tic-Tac-Toe

On cherche à implémenter un jeu de tic-tac-toe https://fr.wikipedia.org/wiki/Tic-tac-toe avec un joueur pseudo-aléatoire et un joueur "intelligent". Récupérer le code à compléter sur https://www.lri.fr/~bagneres/index.php?section=teaching&page=2015_Et5_DA.

## 1 Plateau & jeu

Créer une classe qui correspond au jeu (`tic_tac_toe_t`). Elle fournira les services suivants :

1. accès (en lecture seule) au plateau, au nombre de lignes, au nombre de colonnes, à une case, au joueur courant (avec les fonctions membres suivantes : `board`, `nb_row`, `nb_col`, `operator ()(size_t const i, size_t const j)`, `get(size_t const i, size_t const j)`, `is_playing`)
2. accès (en lecture seule) à l'historique (`log_first_player`, `log_plays`)
3. jouer un coup (`play(size_t const i, size_t const j)`, `play(hopp::vector2<size_t> const & i_and_j)`)
4. lancer la partie avec deux joueurs (`run(player_t0 & player_0, player_t1 & player_1)`, `run(player_t0 && player_0, player_t1 && player_1)`) (ces deux fonctions appelleront la fonction membre `private .run_(player_t0 & player_0, player_t1 & player_1)`)
5. accès (en lecture seule) aux informations sur la fin du jeu (`is_ended`, `winner`)
6. génère les différents coups possibles pour le joueur courant (`moves_possible`)

L'attribution des `X` ou des `O` sera faite dans le constructeur et sera pseudo-aléatoire.

Le stockage des indices peut se faire dans le type de votre choix (`hopp::vector2<size_t>`, `std::pair<size_t, size_t>`, ...).
Permettre l'affichage du jeu avec l'opérateur `<<` entre un `std::ostream` et le jeu.

### 1.1 Expliquer/Justifier le(s) choix de votre conteneur (conteneur + type des éléments) pour stocker le plateau.

### 1.2 Quels sont les avantages et inconvénients d'utiliser la fonction membre `run(player_t0 & player_0, player_t1 & player_1)` contrairement à `run(player_t0 && player_0, player_t1 && player_1)` ?

http://en.cppreference.com/w/cpp/language/reference
http://en.cppreference.com/w/cpp/language/template_argument_deduction

## 2 Joueur pseudo-aléatoire

Créer un joueur pseudo-aléatoire (classe `player_random`). Cette classe a une fonction membre qui retourne le coup à jouer étant donné le jeu passé en paramètre.
Signature approximative : `/*TODO*/ play(tic_tac_toe_t const & tic_tac_toe)`

### 2.1 Que faire si aucun coup ne peut être jouer ?

## 3 Intelligence Artificielle

On souhaite créer un joueur "intelligent" "qui-ne-peut-pas-perdre" (classe `player_ai`). Cette classe s'utilise comme `player_random` (la fonction membre `play` retourne le coup à jouer).

### 3.1 Pourquoi ne peut-on pas créer une intelligence artificielle qui gagne tout le temps ?

Pour trouver un des coups le plus intéressant à jouer, il faut explorer toutes les possibilités et les évaluer.

On propose d'utiliser un arbre comme conteneur (voir la classe `tree<T>` proposée). Un nœud de l'arbre (donnée membre `data`) doit contenir un jeu (`tic_tac_toe_t`) et un entier (`int`) permettant l'évaluation du jeu pour un joueur.

Dans la classe `tree`, écrire une fonction membre `nb_children` qui retourne le nombre descendants (en utilisant `std::accumulate` (et une lambda) http://en.cppreference.com/w/cpp/algorithm/accumulate).

 

```
template <class T>
class tree
{
public:

    T data;

    std::vector<tree<T>> children;

public:

    tree(T const & data = T()) : data(data), children() { }

    size_t nb_children() const
    {
        return 0; // TODO
    }
};
```

### 3.2 Pourquoi les données membres `data` et `children` peuvent-elles être `public` ?

Dans la classe `player_ai` :

- Écrire une fonction membre `generate_games` qui génère tous les jeux possibles à partir du jeu passé en paramètre.
  Signature approximative : `std::vector<tree</*TODO*/>> generate_games(tic_tac_toe_t const & tic_tac_toe)`
- Écrire une fonction membre `generate_children` qui génère tous les jeux possibles depuis ce nœud de l'arbre. (Cette fonction sera probablement récursive.)
  Signature approximative : `void generate_children(tree</*TODO*/> & games)`

### 3.3 Au premier appel de la fonction `generate_games`, quel est le nombre de jeux stockés dans l'arbre ?

### 3.4 Quel est le nombre d'états que le plateau de jeu peut avoir ?

Écrire une fonction membre statique `compare_node` qui compare la note de deux nœuds (avec l'opérateur `<`).
Signature approximative : `static bool compare_node(tree</*TODO*/> const & a, tree</*TODO*/> const & b)`

Écrire une fonction membre `minimax` qui parcourt tous les nœuds de l'arbre pour les évaluer :

- Si le jeu est fini sans gagnant (égalité), la note est 0
- Si le jeu est fini et que le joueur a gagné, la note est 1
- Si le jeu est fini et que le joueur a perdu, la note est $-1$
- Si le jeu n'est pas fini et que c'est au joueur adverse de jouer, la note est la note minimale de tous les fils
- Si le jeu n'est pas fini et que c'est au tour du même joueur que dans la fonction membre `play`, la note est la note maximale de tous les fils

Pour plus d'information sur cet algorithme : https://fr.wikipedia.org/wiki/Algorithme_minimax
(Utiliser l'algorithme `std::minmax_element` http://en.cppreference.com/w/cpp/algorithm/minmax_element.)
Signature approximative : `void minimax(tree</*TODO*/> & games, /*TODO*/ const is_playing)`

Mesurer le temps mis par `generate_children` et `minimax` (avec `hopp::now::seconds()` par exemple).
(Les calculs pourraient être plus rapide en ne parcourant qu'une seule fois l'arbre (en fusionnant `generate_children` et `minimax`).)

### 3.5 Hors optimisation de l'implémentation actuelle, donner 2 autres solutions (algorithmiques) permettant d'obtenir la solution plus rapidement ? Comment les implémenter ?

Dans la fonction membre `.play()`, récupérer le meilleur jeu avec `std::max_element` http://en.cppreference.com/w/cpp/algorithm/max_element et renvoyer le coup à jouer à partir de l'historique.

### 3.6 Regarder la "qualité" des premiers coups, expliquer pourquoi l'IA fait de tels "choix" ? Comment faire quelque chose de plus "malin" ?