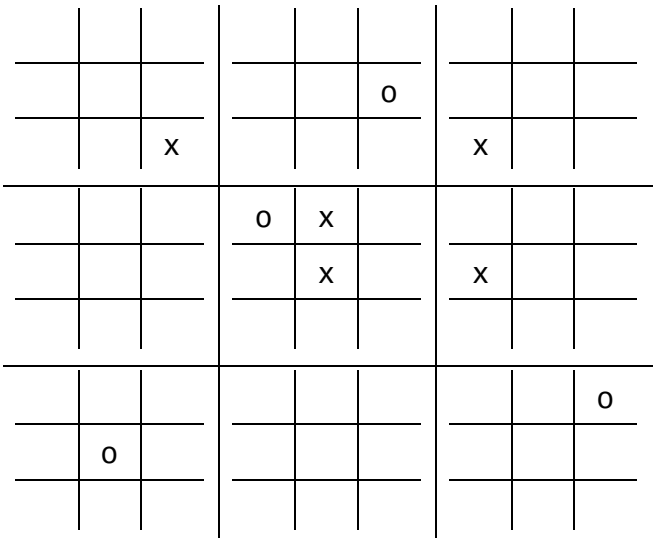
TP1 – IFT2015 – H17 – Major – Ultimate tic-tac-toe

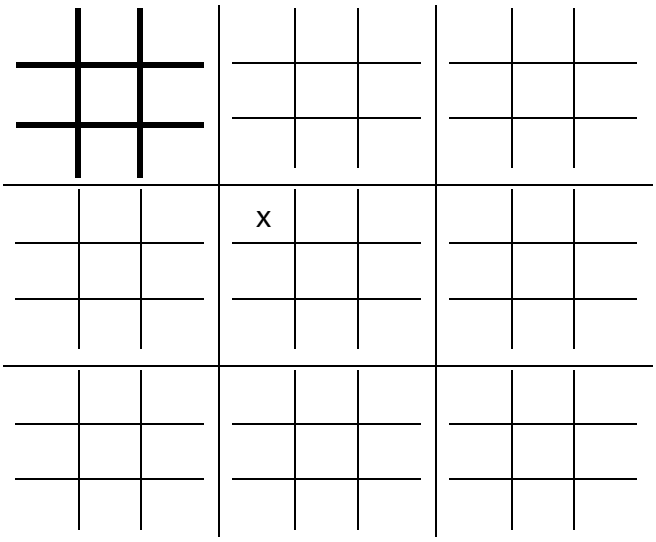
Pour ce TP, vous allez programmer un joueur de Ultimate tic-tac-toe (UTTT). Cela peut vous sembler simple à première vue, mais détrompez-vous! Cette variante du classique est plus complexe qu'elle ne semble l'être.

Ultimate tic-tac-toe

UTTT est en fait un jeu de tic-tac-toe dont chacune des 9 cases est elle-même un jeu de tic-tac-toe. Par exemple, ceci est une configuration possible du jeu:



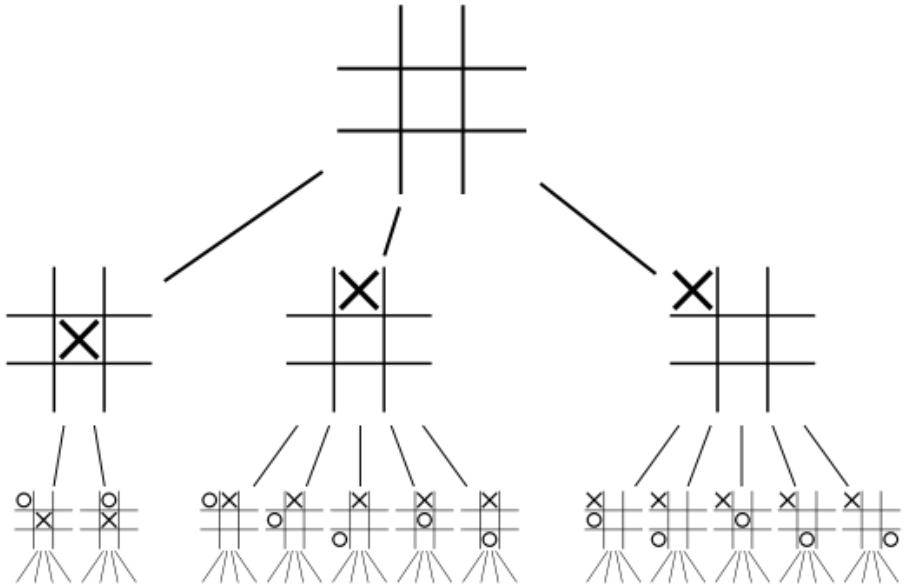
Le but du jeu est de gagner la partie externe. Chaque sous-partie gagnée compte comme un symbole du gagnant dans la partie externe, et le but est de former une ligne droite ou diagonale de sous-parties gagnées, comme au tic-tac-toe standard. Jusqu'ici, rien d'intéressant: pour des joueurs expérimentés, on s'attendrait à ce que toutes les sous-parties soient nulles et donc qu'il n'y ait jamais de gagnant. C'est pourquoi on introduit la restriction suivante: la case dans laquelle un joueur joue dans une sous-partie détermine la sous-partie dans laquelle son adversaire devra jouer au prochain tour. Par exemple, dans la figure suivant, "x" a commencé en jouant dans la sous-partie du centre, dans le coin supérieur gauche. Ceci force "o" à jouer dans la sous-partie du coin supérieur gauche (en gras):



Lorsqu'une sous-partie est gagnée (ou remplie et nulle), on ne peut plus y jouer. Si un joueur envoie son adversaire vers une telle sous-partie, l'adversaire peut jouer sur n'importe quelle case. C'est toujours "x" qui commence et il peut aussi jouer sur n'importe quelle case à son premier tour. Consultez la page Wikipédia pour de plus amples renseignements: https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe

Votre joueur d'UTTT

Pour le tic-tac-toe standard, il est possible de construire l'arbre de jeu complet, dans lequel chaque noeud est une configuration du jeu. Les enfants d'un noeud sont toutes les configurations accessibles à partir de cette configuration, en un coup (ici on simplifie les configurations équivalentes par symétrie ou rotation):



https://en.wikipedia.org/wiki/Tic-tac-toe - /media/File:Tic-tac-toe-game-tree.svg

Un calcul naïf donne en effet 9! = 362880 déroulements possibles, ce qui est aisément calculable par ordinateur. Par contre pour UTTT, même en considérant que les joueurs auront toujours 9 possibilités (donc jamais un joueur n'enverra son adversaire sur une sous-partie complétée), on obtient $(9!)^9$ ou environ 10^{50} possibilités...

Comment explorer cet espace? Utiliser une heuristique! Pour ce genre de problème, une heuristique populaire est la recherche arborescente Monte-Carlo (ou Monte Carlo tree-search en anglais, voir page Wikipédia: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search) : on bâtit l'arbre de jeu complet jusqu'à une certaine profondeur, et ensuite on effectue un échantillonnage aléatoire des résultats possibles à partir de ces configurations. Ceci permet d'approximer la probabilité de succès associée à chaque coup possible et de choisir le coup qui maximise cette probabilité. Toutefois vous ne serez pas restreints dans le choix de votre stratégie heuristique, en autant que votre programme puisse construire l'arbre de jeu complet pour une faible profondeur. Si vous manquez d'inspiration, une stratégie simple est décrite dans l'**Appendice**.

Pour l'évaluation, votre joueur se fera présenter des configurations à quelques coups de la fin et devra prendre la (ou une des) décision optimale. Vous devrez aussi être en mesure de représenter l'état actuel d'une partie selon un standard défini plus loin. Finalement, votre programme devra produire l'arbre de jeu complet pour une configuration et une profondeur données.

En extra, nous vous proposons un tournoi amical entre ceux qui voudront bien participer. Il aura lieu durant la deuxième moitié d'un de vos cours du mercredi, à déterminer.

Représentation

Pour pouvoir explorer beaucoup de configurations en utilisant peu de mémoire, nous voulons une représentation compacte. Nous utiliserons donc un entier pour représenter une configuration du jeu, avec 2 bits par case:

```
00 :          case vide
01 :          x
10 :          o
11 :          pas utilisé
```

62 bits État de la grille 7 bits Dernier coup joué

Comme le jeu contient 81 cases, cette représentation donne un entier de 162 bits. Heureusement, les entiers n'ont pas de limite de taille en Python. En plus de spécifier l'état de chaque case, une configuration doit donner le dernier coup joué, qui détermine la sous-partie dans laquelle le prochain coup sera joué. L'indice de la case sera donc encodé sur 7 bits et sera ajouté en préfixe aux 162 bits, pour donner un entier de 169 bits donc les 7 bits les plus significatifs spécifient la case du dernier coup. Il est important de s'entendre sur une numérotation des cases. La voici:

```
 0   1   2 |  9  10  11 | 18  19  20
 3   4   5 | 12  13  14 | 21  22  23
 6   7   8 | 15  16  17 | 24  25  26
-----------------------------------
27  28  29 | 36  37  38 | 45  46  47
30  31  32 | 39  40  41 | 48  49  50
33  34  35 | 42  43  44 | 51  52  53
-----------------------------------
54  55  56 | 63  64  65 | 72  73  74
57  58  59 | 66  67  68 | 75  76  77
60  61  62 | 69  70  71 | 78  79  80
```

En général, pour accéder à ces cases à partir de l'entier, on peut utiliser le décalage de bits à droite. Par exemple, pour accéder à la valeur à une case *i*:

```
valeur = entier >> ((80-i)<<1) & 3
```

Ou pour obtenir l'indice de la case du dernier coup joué:

```
last = entier >> 162 & 127
```

Votre programme doit jouer un coup à la fois et recevra l'entier représentant la configuration actuelle en argument de ligne de commande, en représentation décimale. Il devra produire en sortie l'entier de la configuration résultante après son coup, aussi en représentation décimale. Cette configuration sera vérifiée afin qu'elle soit conforme et qu'elle représente bien un coup possible.

Vous devrez aussi inclure une méthode d'impression d'un état sur la sortie standard. Pour avoir tous vos points, cette impression devra être parfaitement conforme aux normes suivantes:

- Les symboles représentant les cases occupées sont les lettres x et o minusules (ASCII 120 et 111)
- Le symbole représentant les cases inoccupées est le point: . (ASCII 46)
- Chacune des 81 cases est composée de 3 caractères: Espace caractère Espace (ASCII 32), où caractère est un x, un o ou un .
- Les sous-parties sont séparées verticalement par des | (ASCII 124)
- Les sous-parties sont séparées horizontalement par des lignes complètes de 29 caractères - (ASCII 45)
- Le dernier coup est identifié avec une lettre majuscule X ou O (ASCII 88 et 79)

Voici un exemple de représentation d'un état d'une partie (le X majuscule se trouve à la case 11):

```
 .  o  o | .  o  X | .  x  o
 .  x  . | x  x  x | x  o  x
 o  .  x | o  .  x | x  .  o
-------------------------------
 .  x  . | o  o  . | o  x  o
 o  x  x | o  x  o | x  x  o
 .  x  . | o  x  o | .  o  o
-------------------------------
 x  o  . | o  .  x | x  .  x
 o  x  . | o  x  x | .  o  x
 x  .  x | .  o  . | o  o  o
```

Arbre de jeu

Votre programme devra construire l'arbre de jeu partiel, c'est-à-dire un arbre de jeu complet limité à un certain nombre de niveaux. Chaque noeud de l'arbre de jeu a comme enfants les configurations données par chaque coup possible. Vous devrez construire cet arbre partiel à partir d'une configuration et d'une profondeur donnés. Une profondeur de 1 veut dire la racine et une génération d'enfants.

Affichage de l'arbre

L'arbre est affiché par une **traversée en largeur**, c'est-à-dire niveau par niveau en commençant à la racine. Pour chaque noeud, l'entier représentant sa configuration est affiché en représentation décimale, suivi d'un espace. Chaque niveau correspond à une ligne. Par exemple, pour un arbre de profondeur 1, on pourrait avoir cet affichage:

```
459329034283597291728327479273734123420780266358036
330716890198477834926403213994701218254008155997460 319024877099830611580773735332970962541009996973332
```

L'ordre des enfants n'a pas d'importance.

Entrée

Votre programme doit comprendre un fichier `tp1_`*`matricule1_matricule2`*`.py` où vous remplacez *matricule* par votre ou vos matricules. Dans le mode par défaut, votre programme se fait passer une configuration sous forme d'entier en représentation décimale comme seul argument et retourne la configuration après avoir joué un coup, aussi sous forme d'entier en représentation décimale. Exemple d'appel et de sortie:

```
>python3 tp1_0000000.py 459329034283597291728327479273734123420780266358036
330716890198477834926403213994701218254008155997460
>
```

Notez bien que **rien d'autre** que l'entier n'est affiché en sortie.

En plus de ce mode "jeu", vous aurez à fournir un mode "arbre". Pour accéder à ce mode, on passera à votre programme l'argument `a` suivi de la profondeur de l'arbre de jeu à calculer, suivie elle-même d'une configuration sous forme d'entier en représentation décimale. Cette configuration passée en entrée correspond à la racine de l'arbre. Voici un exemple d'appel et de sortie:

```
>python3 tp1_0000000.py a 1 459329034283597291728327479273734123420780266358036
459329034283597291728327479273734123420780266358036
330716890198477834926403213994701218254008155997460 319024877099830611580773735332970962541009996973332
>
```

Finalement, dans le mode "affichage", on passera à votre programme l'argument `p` suivi d'une configuration sous forme d'entier en représentation décimale. Votre programme devra afficher l'état de la partie en sortie sans faire de coup, tel que défini plus haut. C'est en quelque sorte un mode "traduction pour un humain". Exemple:

```
>python3 tp1_0000000.py p 459329034283597291728327479273734123420780266358036
 o  x  . | x  .  . | x  .  .
 o  x  . | .  x  o | x  .  o
 o  .  o | x  o  o | .  x  .
-----------------------------
 .  o  o | o  o  o | x  x  x
 x  x  o | .  x  o | x  .  .
 x  o  o | o  .  o | x  .  .
-----------------------------
 .  x  . | o  .  o | x  o  .
 x  o  x | o  x  o | x  x  .
 o  x  o | .  .  x | X  x  .
>
```

Sortie

Votre sortie doit être exactement telle que décrite, incluant les espaces séparant les caractères. Votre programme sera évalué sur différentes configurations initiales et devra prendre des décisions optimales lorsque la configuration est assez près de la fin. Si la configuration est loin de la fin, votre sortie devra simplement correspondre à un coup permis par les règles.

Structure du code

Avertissement : vous devez développer VOTRE PROPRE CODE et tout plagiat détecté entraînera automatiquement un échec.

Votre code doit être clair et bien documenté. Vous pouvez aller consulter PEP 8 qui est un guide de style pour Python à la page suivante : https://www.python.org/dev/peps/pep-0008/

Voici quelques idées de classes utiles :

- *Game* qui est une classe pour un jeu de tic-tac-toe standard
- *MetaGame* qui gère l'entier représentant le jeu, appelle Game pour tester les sous-parties et gère l'impression et la modification des configurations
- *Node* qui est un noeud dans l'arbre de jeu
- *GameTree* qui est l'arbre de jeu et qui sera responsable de l'impression de l'arbre

Pour les arguments en ligne de commande, ils se trouvent dans la liste `sys.argv` et comme il n'y a que trois possibilités vous ne devriez pas avoir besoin de modules externes pour les traiter.

Version de Python
Il est fortement recommandé d'utiliser Python 3.6.0 bien qu'il serait surprenant que des problèmes de compatibilité surviennent.

Équipes
Vous pouvez faire votre travail seul ou en équipes de **deux** personnes **maximum**. Vous pouvez discuter avec les autres équipes évidemment, mais tout code dupliqué entre deux équipes sera considéré comme un plagiat des deux côtés.

Remise
Vous avez jusqu'au **22 mars à 23h55** pour remettre votre travail sur studium. Remettez votre travail soit sous forme d'un seul fichier `tp1_matricule1_matricule2.py` ou sous la forme d'une archive `tp1_matricule1_matricule2.tar.gz` ou `tp1_matricule1_matricule2.zip` (**aucune** autre forme de compression ne sera tolérée) si vous avez plus d'un fichier. Dans ce cas, l'archive devra contenir entre autres le fichier `tp1_matricule1_matricule2.py` qui sera exécuté.

Correction négative
- 50% des points peuvent être perdus si votre code ne s'exécute pas ou ne produit pas de sortie
- Une pénalité de **20% par jour de retard** sera appliquée dès la première seconde de chaque période de 24h, en commençant à 23h56 le 22 mars

Correction positive
- 30% : clarté du code
- 10% : tests d'impression de configurations
- 30% : tests de génération d'arbres
- 30% : tests de jeu : vos coups doivent être optimaux lorsque possible et respecter les règles

Points boni
Des points boni seront accordés en fonction de la vitesse d'exécution (par rapport à la moyenne) de votre code pour la génération d'arbres:
Code rapide : +4%
Code très rapide: +7%
Code extrêmement rapide: +10%

Questions
Envoyez vos questions sur le travail à omailhot92@gmail.com.
Bon travail !!!

Appendice : recherche arborescente Monte Carlo

Voici une façon simple de faire la recherche arborescente Monte Carlo dans le cadre de ce TP. Voici les méthodes dont vous aurez besoin:

- MetaGame.winner() : donne le gagnant actuel de la partie, s'il y a lieu, et différencie une partie nulle terminée d'une partie non-terminée
- MetaGame.possibleMoves() : donne la liste des coups permis à partir de la configuration actuelle
- MetaGame.getInt(move) : retourne la représentation entière de l'état de la partie après que le joueur actuel ait joué à la position move
- Node.sample(n) : à partir de la configuration à ce noeud, effectuer n simulations de coups aléatoires jusqu'à la fin de partie et garder les statistiques du nombre de parties gagnées par x et gagnées par o (les autres étant nulles)

La stratégie : à partir d'une configuration, générer l'arbre de jeu complet pour une profondeur de 1, c'est-à-dire générer seulement les enfants résultant de chacun des coups possibles pour votre joueur. Si un noeud représente une fin de partie dans laquelle vous êtes gagnant, sélectionnez ce coup. Sinon, pour chaque noeud qui n'est pas une fin de partie, utilisez Node.sample(n) avec un n assez grand, disons 1000 ou plus. Ensuite, sélectionnez le coup qui donne la meilleure probabilité de gagner à votre joueur. Vous pouvez expérimenter avec des profondeurs plus grandes d'exploration complète avant la génération des probabilités, mais il y aura plus d'exceptions à traiter dans ce cas, par exemple un noeud interne qui est une fin de partie.