TD09 : Jeux d'accessibilité à deux joueurs (V3)

- Référence : ce TD est inspiré de la source que l'on trouvera ici
- Code du notebook associé : **224e-1319023**

1 Cadre général

1.1 Jeux d'accessibilité à deux joueurs

Dans le cadre du programme, on s'intéresse à des jeux à deux joueurs, dans lesquels chaque joueur joue à tour de rôle, avec une connaissance complète de l'état du jeu (jeux *à information complète*), avec pour but d'atteindre un état du jeu qui permet de le déclarer gagnant (jeux d'*accessibilité*).
Note : Les jeux considérés sont « sans mémoire », dans le sens où l'historique des coups précédents ne sera pas pris en compte, et se déroulent en temps « infini » (chaque joueur dispose du temps qu'il souhaite pour jouer un coup).

1.2 Modélisation du jeu

On modélise le jeu par une *arène*, qui est un graphe orienté et fini, $G = (V, E)$, dans lequel les sommets du graphe représentent des états du jeu (ensemble V), et les arêtes (ensemble E) représentent les coups possibles (faisant passer d'un état du jeu à un autre).
Cependant, un état du jeu, pour être pleinement descriptif doit indiquer quel est le joueur qui est autorisé à jouer. Typiquement, un état du jeu consistera en la description du plateau de jeu et la connaissance du joueur qui a la main.
Ainsi, si on numérote 1 et 2 les deux joueurs, l'ensemble des sommets, peut être partitionné en deux sous-ensembles, V_1 et V_2, ensembles des états du jeu où le joueur 1, respectivement le joueur 2, a la main (on dira que les états de V_i sont des états *contrôlés* par le joueur i).
On entoure usuellement d'un cercle les états contrôlés par le joueur 1 et d'un carré les états contrôlés par le joueur 2.
Dans la mesure où l'on traite de jeux d'accessibilité, on doit aussi spécifier deux sous-ensembles, Ω_1 et Ω_2, correspondant, respectivement, aux états du jeu dans lequel le joueur 1 ou le joueur 2 est gagnant.
On appelle Ω_i l'objectif du joueur i.
Éventuellement, on peut distinguer un état Ω_3 correspondant aux états du jeu donnant une partie nulle.

Lorsque les deux joueurs jouent à tour de rôle, les ensembles V_1 et V_2 sont conditionnés par la donnée du joueur qui commence la partie.
Il conviendra donc d'indiquer quel est le joueur qui jouera en premier (on conviendra ici qu'il s'agira systématiquement du joueur 1), ainsi que l'état initial du jeu.
Si les joueurs jouent à tour de rôle, les arêtes, représentant les coups jouables, *i.e.* les transitions possibles entre états du jeu, seront données par des couples (s, s') dans lesquels, nécessairement, $s \in V_1$ et $s' \in V_2$, ou bien $s \in V_2$ et $s' \in V_1$. De ce fait, l'arène sera un graphe dit *biparti*, dans lequel toute arête a son origine dans V_i et son extrémité dans V_j avec $(i, j) \in \{1,2\}$ et $i \neq j$.

1.3 Déroulement d'une partie

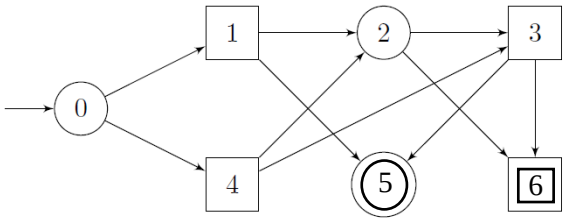
Un état de départ pour le jeu étant donné, incluant la connaissance du joueur contrôlant cet état initial (le joueur 1 ici, arbitrairement), une partie sera modélisée par une succession de coups (éventuellement infinie), retraçant les coups joués depuis le début de la partie, jusqu'à son terme s'il existe : il s'agit donc d'un chemin dans le graphe orienté qu'est l'arène.

1.4 Stratégies

On appelle *stratégie* pour le joueur i, une fonction qui, à chaque état contrôlé par le joueur i, associe une transition possible depuis cet état (*i.e.* le coup que jouera le joueur i).
Une stratégie sera dite *gagnante* pour le joueur i, si l'application de cette fonction assure la victoire au joueur i, quelle que soit la stratégie du joueur adverse.

2 Étude d'un exemple

On considère le jeu décrit par l'arène suivante



où $G = (V, E)$ avec $V = \{(0,1)\,;(1,2);(2,1);(3,2);(4,2);(5,1);(6,2)\}$, un sommet de V étant décrit par un couple (i, j) donnant son numéro, i, et le numéro du joueur qui contrôle cet état, J, et avec $E = \{((0,1),(1,2)),((0,1),(4,2)),((1,2),(2,1)),\dots,((4,2),(3,2))\}$.
L'état initial (signalé par une flèche) est l'état 0, contrôlé par le joueur 1, qui commence donc toute partie.
Les états 5 et 6 correspondent, respectivement aux objectifs des joueurs 1 et 2, qui gagnent s'ils les atteignent au cours d'une partie. On les a mis en évidence en les entourant avec un trait double.
Avec les notations précédentes, les objectifs des deux joueurs sont donc $\Omega_1 = \{(5,1)\}$ et $\Omega_2 = \{(6,2)\}$.

Q1. Afin d'implémenter le jeu décrit ci-dessus, définir une liste des numéros de sommets, `LS`, et un dictionnaire, `dC`, dont les clés sont les numéros de sommets, avec pour valeur associée, le numéro du joueur contrôlant cet état, ainsi qu'un dictionnaire des listes d'adjacences de chaque sommet, `dAdj`.
Définir ensuite deux listes, `LObj1` et `LObj2`, donnant les états-objectifs des deux joueurs.
On définira enfin le jeu avec toutes ses caractéristiques, de la façon suivante :

```
jeu = {'etats': LS, 'init': 0, 'controles': dC, 'transitions': dAdj,
       'objectifs1': LObj1, 'objectifs2': LObj2}
```

où la clé `'init'` est associée à l'état initial du jeu.

Une stratégie pour le joueur 1, correspond à la donnée d'une fonction associant à chaque état contrôlé par le joueur 1, un état suivant possible.
On peut modéliser une telle fonction par un dictionnaire `strategie1`, dont les clés sont les états contrôlés par le joueur 1 et les valeurs associées l'état suivant que choisit d'atteindre le joueur 1, à partir de chacun d'eux.
Par exemple, le dictionnaire suivant est une stratégie possible pour le joueur 1 :

```
strategie1 = {0: 1, 2: 3}
```

Q2. Combien existe-t-il de stratégies pour le joueur 1 ?

Q3. Il existe une stratégie gagnante pour le joueur 2. Laquelle ? La décrire à l'aide d'un dictionnaire, `strategie2`.

Q4. Écrire une fonction `execute` qui prend en paramètre une description du jeu sous la forme donnée en **Q2**, et deux stratégies, une pour chaque joueur, et fait avancer la partie en suivant les instructions données par les deux stratégies, et s'arrête quand un objectif a été atteint et renvoie la liste des états du jeu. Dans le cas où une stratégie propose une transition non autorisée, on lèvera une exception.

Q5. Tester la fonction `execute` avec le jeu décrit en **Q1** et la stratégie gagnante pour le joueur 2 et diverses stratégies pour le joueur 1, afin de vérifier que l'on arrive toujours dans l'état gagnant pour le joueur 2.

Attracteurs

Afin de déterminer si une stratégie gagnante existe pour l'un des deux joueurs, on peut calculer des ensembles d'*attracteurs* pour chacun des joueurs.
Pour déterminer les stratégies gagnantes, s'il en existe, pour le joueur *j*, on définit une suite $\left(Attr_i^j\right)_{i\geq 0}$, où $Attr_i^j$ est l'ensemble des états du jeu à partir desquels le joueur *j* dispose d'une stratégie qui le fait gagner en *i* étapes ou moins (chacune des *i* étapes correspondant à un coup joué par l'un ou l'autre joueur).
On construit cette suite par récurrence.

Pour $i = 0$, $Attr_0^j = \Omega_j$. En effet, le joueur j gagne en zéros étapes s'il se trouve déjà dans l'un des états réalisant son objectif.

Pour tout $i \geq 1$, si l'ensemble $Attr_{i-1}^j$ est connu, alors le joueur j est assuré de gagner en i étapes ou moins, si et seulement si :

- le jeu est dans un état élément de $Attr_{i-1}^j$;
- le jeu est dans un état e contrôlé par j et à partir duquel il existe une transition (un coup à jouer par j) vers un état e' élément de $Attr_{i-1}^j$;
- le jeu est dans un état contrôlé par l'adversaire, mais à partir duquel toutes les transitions possibles (tous les coups que l'adversaire peut jouer) amènent à un état e' élément de $Attr_{i-1}^j$.

Ce qui peut se traduire, en notant $\bar{j}$ l'opposant du joueur j, par la relation de récurrence suivante :

$$Attr_i^j = Attr_{i-1}^j \cup \left\{e \in V_j, \exists\, (e,e') \in E, e' \in Attr_i^j\right\} \cup \left\{e \in V_{\bar{j}}, \forall\, (e,e') \in E, e' \in Attr_i^j\right\}.$$

On définit ainsi une suite d'ensembles d'états, croissante pour l'inclusion, et, donc, nécessairement stationnaire, car l'ensemble des états est fini.

On pourra noter $Attr_\infty^j$ l'ensemble des états atteint au moment où la suite devient stationnaire.

Si l'on construit cette suite jusqu'à ce qu'elle soit stationnaire, **on peut conclure à l'existence d'une stratégie gagnante pour le joueur j, si l'état initial est élément de $Attr_\infty^j$.**

Dans ce qui suit, on appelle *successeur* d'un état e un état e' tel qu'il existe une transition possible de e à e'.

Q6. En supposant que les attracteurs sont définis sous la forme de listes Python d'états, définir une fonction, `appartientL`, prenant en argument un état `e` et un attracteur, `A`, et renvoyant un booléen indiquant si l'état `e` appartient à `A` ou non. Évaluer la complexité d'un appel à `appartientL` en fonction de `len(A)`.

Q7. Expliquer en quoi une représentation des attracteurs sous la forme de dictionnaires dont les clés sont les états appartenant à l'attracteur, avec pour valeur associée `None` (ou toute autre valeur arbitrairement choisie), permet d'améliorer de définir une fonction `appartientD`, testant si un état appartient à un attracteur, de meilleure complexité que la fonction `appartientL`.

Q8. Définir deux fonctions prenant toutes deux en argument la description d'un jeu, `jeu`, un état, `e`, un attracteur `A`, et une fonction booléenne, `appartient`, testant l'appartenance d'un état `e` à `A` :

- la première de ces fonctions, `exist_succ`, renverra `True` s'il existe au moins un successeur de l'état `e` qui soit dans `L` ;
- la seconde, `forall_succ`, renverra `True` si l'état `e` a au moins un successeur et que tous les successeurs possibles de l'état `e` sont éléments de `L`.

Q9. Définir une fonction `step`, prenant en argument la description d'un jeu, `jeu`, un numéro de joueur, `j`, et un attracteur, `A` (sous forme de dictionnaire), représentant un état de l'attracteur après i itérations, $Attr_i^j$, et ajoutant à l'attracteur `A`, les états qu'il convient, pour obtenir l'attracteur $Attr_{i+1}^j$. La fonction `step` renverra `True` si des états ont été ajoutés et `False` sinon.

Q10. Écrire une fonction `attracteur_infty`, prenant en argument la description d'un jeu, `jeu` et un numéro de joueur, `j`, et renvoyant la liste des états de l'attracteur $Attr_\infty^j$.

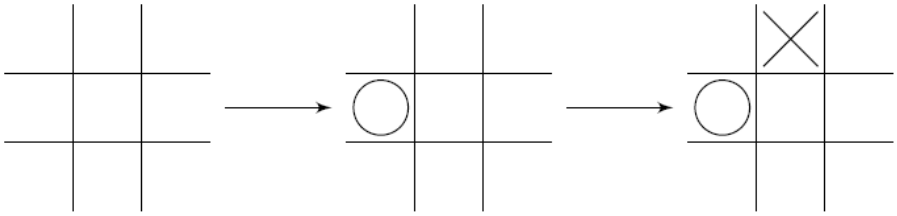
Q11. Tester la fonction `attracteur_infty`, sur le jeu donné en exemple et vérifier qu'il est confirmé qu'il existe une stratégie gagnante pour le joueur 2, mais pas pour le joueur 1.

Dans le cas où il existe une stratégie gagnante pour un joueur, on voudrait pouvoir déterminer explicitement l'une de ces stratégies.

Q12. Modifier les fonctions `exist_succ`, `step` et `attracteur_infty` en trois fonctions `exist_succ_strategie`, `step_strategie` et `attracteur_infty_strategie` afin de construire et renvoyer un dictionnaire des coups à jouer pour tous les états présents dans l'attracteur $Attr_\infty^j$ pour atteindre un état gagnant pour le joueur j.

On créera, dans la version modifiée de la fonction `attracteur_infty`, une variable locale `strategie`, sous la forme d'un dictionnaire associant aux états contrôlés par le joueur j, l'état suivant à choisir pour atteindre un état-objectif. Ce dictionnaire sera initialisé au dictionnaire vide et passé en argument de chaque appel à la fonction `exist_succ_strategie`, pour qu'il soit actualisé.

3 Étude du jeu de Morpion



On représente un état du plateau de jeu par un tableau à deux dimensions de taille 3×3, dont les éléments sont les caractères, espace, `'O'` ou `'X'`, figurant, respectivement une case vide, une case occupée par le joueur 1 ou le joueur 2.
Ces différents symboles seront stocké dans un dictionnaire `symboles = {0: ' ', 1 : 'O', 2 : 'X'}`, utilisé en tant que variable globale.

Q13. Écrire une fonction `plateau_vide()`, renvoyant le plateau dans son état initial, vide.

Q14. Écrire une fonction `jouer(plateau, i, j, k)`, qui renvoie une copie du plateau, dans laquelle on a ajouté le symbole du joueur k dans la cellule de coordonnées (i, j).

Q15. Écrire une fonction `lire(plateau, i, j)`, qui renvoie le numéro du joueur occupant la cellule de coordonnées (i, j), en fonction du symbole lu dans la case. La fonction renverra zéro si la cellule est inoccupée.

On rappelle qu'il est convenu que le joueur 1 est le joueur commençant la partie.

Q16. Écrire une fonction `controle`, prenant en argument une configuration (un état du plateau de jeu), et renvoyant le numéro du joueur qui doit jouer le coup suivant, et qui lève une exception si la configuration n'est pas correcte (la vérification se fera uniquement sur le nombre de cellules occupées par l'un ou l'autre joueur).

On rappelle qu'une configuration est gagnante pour le joueur k si le joueur k a réussi à aligner trois symboles, horizontalement, verticalement ou le long de l'une des diagonales du carré.

Q17. Écrire une fonction `gagnante(conf, k)`, renvoyant `True` si la configuration du plateau `conf` est gagnante pour le joueur k.

On souhaite construire la liste, `confs`, de toutes les configurations possibles, ainsi qu'une représentation du jeu ayant la même forme que précédemment, soit

```
jeu = {'etats': LS, 'init': 0, 'controles': dC, 'transitions': dAdj,
       'objectifs1': LObj1, 'objectifs2': LObj2}
```

Les états (les configurations) seront numérotés par leur position dans la liste `confs`.

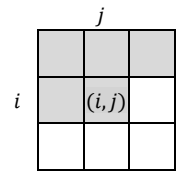
3.1 Première approche pour la construction d'une représentation du jeu

On cherche ici à construire d'abord la liste, `confs_all`, de toutes les configurations possibles du plateau dans lesquelles une case est soit occupée par l'un ou l'autre joueur, soit libre, sans contrôler ici si les nombre de cellules occupées par l'un ou l'autre joueur.
On éliminera ensuite de cette liste les configurations impossibles à atteindre dans un déroulement du jeu conforme aux règles.
On construira ensuite une représentation du jeu.

Q18. Justifier qu'il existe 19 683 configurations possibles du plateau de jeu, lorsque chaque case est soit vide, soit occupée par le joueur 1, soit occupée par le joueur 2. On accepte ici toutes les configurations même si par ailleurs elles ne sont pas atteignables dans un déroulement du jeu conforme aux règles.

Q19. Afin de générer la liste `confs_all`, on initialise cette liste à une liste réduite au plateau vide, puis pour chaque case en position (i, j) $(0 \leq i, j \leq 3)$: on parcourt la liste des configurations déjà construites, et on construit la liste `new_confs` des nouvelles configurations obtenues en faisant occuper la cellule (i, j) soit par le joueur 1, soit par le joueur 2.

Chaque configuration déjà construite, dans laquelle la case (i,j) est vide, en générera donc deux nouvelles, dans lesquelles la case (i,j) est occupée par l'un des deux joueurs, ce qui multipliera par trois le nombres de configurations à chaque fois qu'une nouvelle case (i,j) aura été remplie de toutes les façons possibles. À la fin de chaque itération, pour une valeur donnée de (i,j), on aura construit tous les états possibles pour les cases des lignes 0 à $i-1$ et les cases de la ligne i, en positions $(i,0),\dots,(0,j)$ (zone grisée ci-dessous :



Pour chaque configuration déjà construite, on en fera deux copies indépendantes à l'aide de la fonction `deepcopy` du module `copy`, et se seront sur ces copies que l'on fera occuper la case (i,j), soit par le joueur 1, soit par le joueur 2.
On utilisera la méthode `.extend` pour ajouter, après avoir traité une position (i,j), la liste des nouvelles configurations à la liste `confs_all`.
On pourra compléter le code suivant

```
from copy import deepcopy

symboles = {0: '_', 1: 'O', 2: 'X'}

confs_all = [...]
for i in range(3):
    for j in range(3):
        new_confs = []
        for conf in confs_all:
            for k in range(...):
                new_conf = ...
                jouer(...)
                new_confs.append(...)
        confs_all.extend(new_confs)
```

On vérifiera que l'on a bien construit une liste de 19 683 configurations.

Q20. Définir, en modifiant le code de la fonction `controle`, une fonction `verif1(conf)` renvoyant un booléen, `True` ou `False`, selon que le nombre de cases occupées par l'un ou l'autre joueur est compatible avec un déroulement du jeu conforme aux règles.

Q21. Définir, en modifiant le code de la fonction `gagnante`, une fonction `verif2(conf)` renvoyant un le booléen `True` seulement si la configuration contient aucun alignement gagnant pour le joueur qui la contrôle (ceci peut advenir dans un déroulement normal du jeu car la partie aurait déjà terminé).

Q22. Faire appel aux fonctions `verif1` et `verif2` pour construire à partir de la liste `confs_all`, la liste `confs1` de toutes les configurations compatibles avec un déroulement du jeu conforme aux règles.
On pourra alors construire la liste `LS1` des numéros d'états du jeu à l'aide de l'instruction suivante :

```
LS1 = [i for i in range(len(confs)]
```

Q23. En faisant appel aux fonctions `controle` et `gagnante`, construire, d'une part, le dictionnaires `dC1` donnant pour chacun des deux joueurs la liste des numéros d'états qu'il contrôle, et d'autre par les listes `LObj11` et `LObj12` des numéros des états gagnants pour l'un ou l'autre joueur.

Q24. Écrire une fonction `transition(conf1, conf2)` prenant en argument deux configurations et renvoyant un booléen, égal à `True`, si et seulement si il est possible de passer de la configuration `conf1` à la configuration `conf2` en un coup (on vérifiera que toutes les cases des deux configurations ont le même contenu, à l'exception d'une seule, qui est vide dans la configuration 1 et occupée dans la configuration 2 par le joueur contrôlant la configuration 1).

Q25. Utiliser la fonction `transition` pour construire le dictionnaire `dAdj1` associant à chaque numéro d'état la liste des numéros des états atteignables au depuis cet état dans un déroulement normal du jeu.
On pourra, pour accélérer la recherche, construire un dictionnaire ayant pour clés les nombres de cases occupées possibles, associés chacun à la liste des numéros des états comportant un tel nombre de cases occupées.

Q26. Construire enfin le dictionnaire `jeu1` représentant le jeu.

3.2 Seconde approche pour la construction d'une représentation du jeu

On se propose de construire cette fois une représentation du jeu, en explorant le graphe de tous les déroulements possibles du jeu par un parcours en largeur.
On utilisera une file de type FIFO pour y stocker chaque nouvelle configuration possible du jeu.
Une file sera créée à l'aide de la structure du module `deque` de la bibliothèque `collections`, de la façon suivante :

```
from collections import deque
file = deque()
```

L'ajout d'un élément `x` et le retrait d'un élément seront réalisés, respectivement, par les instructions `file.appendleft(x)` et `file.pop()`.
On posera les initialisations suivantes, traduisant que l'état initial est le plateau vide :

```
confs2 = [plateau_vide()] ## liste des configurations possibles
conf_init = 0 ## numéro de la configuration initiale
LS2 = [0] ## liste des numéros d'états (position de la configuration dans confs2)
dC2 = {0: 1} ## dictionnaire indiquant pour chaque état le joueur qui le contrôle
dAdj2 = {0: []} ## dictionnaire donnant les états atteignables depuis chaque état
LObj21, LObj22 = [], [] ## listes des états objectifs pour les joueurs 1 et 2
```

On initialisera une file contenant initialement l'entier zéro numéro de l'état initial, puis, tant que la file n'est pas vide, on répètera les opérations suivantes :

1. défiler un numéro d'état ;
2. charger la configuration correspondante, `conf`, et déterminer le joueur, `joueur`, qui la contrôle ;
3. pour chaque case vide de cette configuration, créer une copie profonde de cette configuration, `next_conf`, et y faire occuper cette case par le joueur `joueur`.
4. pour chaque configuration `next_conf` :
   - si elle est déjà présente dans la liste `confs2` de toutes les configurations possibles, récupérer son numéro d'état, sinon, l'ajouter à la liste `confs2` et lui affecter un numéro d'état, puis actualiser en conséquence la liste des numéros d'états, `LS2`, le dictionnaire `dC2`, et, s'il s'agit d'une configuration gagnante, la liste `LObj21` ou `LObj22` ;
   - actualiser le dictionnaire `dAdj2` pour traduire qu'il existe une transition de l'état correspondant à la configuration `conf` vers l'état correspondant à la configuration `next_conf`.
   - ajouter le numéro d'état de la configuration `next_conf`, uniquement si elle n'est pas gagnante.

Afin d'accélérer la recherche d'une configuration `next_conf` dans la liste `confs2` et de son numéro d'état, on pourra construire un dictionnaire dont les clés sont les configurations existantes, associées à leur numéro d'état. Comme les configurations sont de type `list`, mutable, il n'est pas possible de les utiliser comme clé d'un dictionnaire, aussi on pourra utiliser la fonction encodage suivante, afin d'obtenir une représentation des configurations de type `str`, non mutable, utilisable comme clé :

```
def encodage(conf):
    ch = ""
    for i in range(3):
        for j in range(3):
            ch = ch + conf[i][j]
    return ch
```

Q27. Appliquer l'algorithme et construire une représentation, `jeu2`, du jeu.

3.3 Recherche de stratégies gagnantes

Q28. À l'aide des fonctions `step_strategie` et `attracteur_infty_strategie`, calculer les attracteurs pour les deux joueurs, $Attr_\infty^1$ et $Attr_\infty^2$pour les deux joueurs.

Q29. Déterminer s'il existe une stratégie gagnante pour l'un des deux joueurs.

Q30. Utiliser les fonctions `step_strategie` et `attracteur_infty_strategie` pour déterminer cette stratégie gagnante, s'il en existe une.

Q31. S'il n'existe aucune stratégie gagnante, décrire l'usage qu'il est possible de faire des dictionnaires renvoyés par l'appel à la fonction `attracteur_infty_strategie` pour l'un et l'autre joueur.