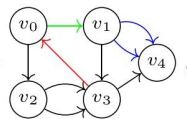


Projet Info LDD2 – TD 3

Objectifs du TP : Traductions graphe \leftrightarrow matrice d'adjacence ; génération aléatoire de matrice / graphe ;

On rappelle que des commentaires et des doc sont attendus pour (à peu près) toutes les fonctions et méthodes du projet.

Une représentation possible d'un (multi)graphe (dirigé) $G = (V, E)$ est via sa matrice d'adjacence A_G . Si on considère que $V = \{v_0, \dots, v_{n-1}\}$, on représente une arête (v_i, v_j) de G par un 1 à la i ème ligne de la j ème colonne de la matrice A_G . (On peut noter cet élément A_{ij} .) Dans un multigraphe, s'il y a k arêtes de v_i à v_j , on représente ça par un k dans la matrice.

Exemple : si $G =$ , alors $A_G =$ $\begin{pmatrix} 0 & \textcolor{green}{1} & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \textcolor{blue}{2} \\ 0 & 0 & 0 & 2 & 0 \\ \textcolor{red}{1} & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ (les cou-

leurs sont simplement là pour indiquer où se trouvent certaines arêtes).

Cette représentation n'est pas la plus efficace : on a besoin d'un espace de taille $O(n^2)$ pour le représenter, alors que notre représentation utilise un espace de taille $O(n \times \Delta)$ avec Δ le degré max du graphe.

Elle a par contre beaucoup de bonnes propriétés mathématiques. On propose donc dans la suite de traduire une représentation en l'autre.

Une représentation naturelle des matrices en info (si on n'utilise pas de librairies tierces), utilise des listes de listes. Dans notre cas, des `int list list`. Par exemple, la matrice A_G ci-dessus peut être représentée en Python par :

```
A = [ [0, 1, 1, 0, 0],  
      [0, 0, 0, 1, 2],  
      [0, 0, 0, 2, 0],  
      [1, 0, 0, 0, 1],  
      [0, 0, 0, 0, 0] ]
```

On peut ensuite accéder à la i ème ligne par `A[i]` et l'élément d'indice (i, j) par `A[i][j]`. Dans la suite du TD, quand on dit matrice, il faut comprendre liste de liste (d'int).

On va commencer en douceur en générant une liste de nombres aléatoires, ce qui nous servira pour définir aléatoirement des entrées et sorties de nos graphes ouverts.

▮ **Exercice 1 :** Définir une fonction `random_int_list(n, bound)` qui génère une liste de taille `n` contenant des entiers (aléatoires) entre 0 et `bound`. On peut

se servir de la fonction `randrange` ou `random` de la librairie `random` (en sachant que `int(x)` permet d'arrondir le float `x` à l'entier inférieur). ┘

On va ensuite définir quelques fonctions pour générer aléatoirement des matrices (carrées) d'entiers. Chaque exercice jusqu'au 6 (sauf le 3) demande de créer une nouvelle fonction qui génère une matrice de la forme voulue. On peut préférer (à choisir) de les agréger en une seule, avec des paramètres additionnels dont on donne une valeur par défaut. On aurait par exemple à la fin :

```
def random_matrix(n, bound, null_diag=False,
                  symmetric=False, oriented=False, triangular=False):
    <function body>
```

┌ **Exercice 2 :** Définir une fonction `random_int_matrix(n, bound)` qui génère une matrice $n \times n$ avec ses éléments des `int` tirés aléatoirement entre 0 et `bound`. ┘

Les matrices obtenues avec la version précédente peuvent avoir des éléments non nuls sur la diagonale. Vue comme une matrice d'adjacence, cela correspondrait à un graphe où l'on peut avoir des arêtes d'un nœud vers lui-même. Cette notion de graphe est parfaitement valide, mais on peut tout de même vouloir des graphes qui n'ont pas ces "boucles".

┌ **Exercice 3 :** Rajouter un paramètre `null_diag` à la fonction précédente, qui spécifie si on veut que la diagonale de la matrice soit nulle, et faire en sorte qu'il prenne une valeur par défaut. E.g. ┘

```
def random_int_matrix(n, bound, null_diag=True):
    # à faire
```

La matrice d'adjacence d'un graphe non-dirigé doit être symétrique (i.e. $A_{ij} = A_{ji}$).

┌ **Exercice 4 :** Définir une fonction `random_symetric_int_matrix(n, bound, null_diag=True)` qui renvoie cette fois-ci une matrice symétrique (toujours avec la possibilité de rendre la diagonale nulle). ┘

On peut vouloir imposer la contrainte suivante sur un graphe : $(v_i, v_j) \in E \implies (v_j, v_i) \notin E$ (ces graphes sont parfois appelés *oriented graphs* en anglais, mais en français orienté=dirigé). Cela implique que les arêtes entre chaque paire de nœuds ne peuvent aller que dans un sens.

┌ **Exercice 5 :** Définir une fonction `random_oriented_int_matrix(n, bound, null_diag=True)` qui fasse ça. ┘

Enfin, on peut faire en sorte que la matrice soit une matrice d'adjacence d'un "DAG" : un graphe dirigé acyclique. Il suffit pour cela de générer une matrice

triangulaire (supérieure, par exemple, c'est à dire A telle que $i > j \implies A_{ij} = 0$).

▮ **Exercice 6 :** Définir une fonction `random_triangular_int_matrix(n, bound, null_diag=True)` qui fasse ça. ▮

On peut maintenant s'arranger pour pouvoir convertir une matrice (d'adjacence) en graphe, et à l'inverse d'extraire la matrice d'adjacence à partir d'un graphe.

▮ **Exercice 7 :** Définir une fonction `graph_from_adjacency_matrix` qui renvoie un multigraphe à partir d'une matrice. On laissera les attributs `inputs` et `outputs` du graphe vides. ▮

On peut maintenant se servir de tout ce qu'on a fait précédemment pour générer des graphes aléatoires, avec qui plus est une certaine marge de manœuvre.

▮ **Exercice 8 (tests requis) :** Définir une méthode pour les graphes :

```
@classmethod
def random(cls, n, bound, inputs=0, outputs=0, loop_free=False,
           DAG=False, oriented=False, undirected=False):
    <function body>
```

qui génère un graphe aléatoire, suivant les contraintes données par l'utilisateur (et gère les options conflictuelles).

Puisque beaucoup de ces options sont exclusives, on peut préférer donner un unique argument sous la forme d'une chaîne de caractères :

```
@classmethod
def random(n, bound, inputs=0, outputs=0, form="free"):
    """
    Doc
    Bien préciser ici les options possibles pour form !
    """
    if form=="free":
        ...
    elif form=="DAG":
        ...
    elif form=="oriented":
        ...
    elif form=="loop-free":
        ...
    elif form=="undirected":
        ...
    elif form=="loop-free undirected":
        ...
```

On sélectionne aléatoirement les entrées et sorties du graphe (`inputs` et `outputs` représentent le nombre voulu d'entrées et de sorties). ▮

On va maintenant faire le chemin inverse : extraire une matrice d'adjacence à partir d'un graphe. Pour simplifier la gestion des indices, on va se doter de la méthode suivante :

┌ **Exercice 9 :** Définir une méthode qui, lorsqu'appliquée à un graphe à n noeuds, renvoie un dictionnaire, associant à chaque `id` de noeud un unique entier $0 \leq i < n$. ┘

┌ **Exercice 10 :** Définir une méthode `adjacency_matrix(self)` qui donne une matrice d'adjacence du graphe (on ignore ici aussi `inputs` et `outputs`). Attention à la gestion des indices. ┘

┌ **Exercice 11 (Bonus) :** La fonction `random.randint` produit des `int` de façon uniforme. Ça vaut dire notamment qu'avec par exemple `random_int_matrix(n, 3)`, il y a entre chaque paire de noeuds autant de chance d'avoir 0 arêtes que d'en avoir 2 ou encore 3. Modifier la fonction `random_int_matrix` pour qu'elle accepte un argument supplémentaire optionnel, qui est une fonction génératrice de `floats` entre 0 et 1 donnée par l'utilisateur. On pourra ainsi utiliser par exemple :

```
random_int_matrix(10, 3, number_generator=(lambda : random.betavariate(1,5)))
```

où `betavariate` est une fonction génératrice de nombres selon une distribution bêta, fournie par la librairie `random`. D'autres distributions sont disponibles dans la librairie `random`. Adapter évidemment les autres fonctions génératrices de matrices, ainsi que la méthode `random` des graphes, pour qu'elles aussi utilisent cet argument supplémentaire. ┘