Feuille 2 : clôture transitive de relations binaires

Note Comme dans la feuille 2, on gardera ouverts deux documents : `anneau02.py` et `relation.py`.

Dans le second document (`relation.py`), vous avez déjà mis les fonctions `composition()` et `union`.

L'objet d'étude de ce TP est une carte, représentée ("modélisée") par un graphe $G = (N, A)$, où

- les noeuds représentent des villes ($N = \{v_1, v_2 \dots v_n\}$)
- les arcs $A \subseteq N \times N$ représentent les chemins élémentaires entre 2 villes. A est donc une relation binaire sur N, et on peut la représenter par une matrice booléenne $G[1..n][1..n]$.

S'il existe un chemin élémentaire (v_i, v_j) entre deux villes différentes (un arc entre 2 nuds différents), il est supposé unique et à double sens. Il faut donc que A soit symétrique. On peut convenir de représenter G par une matrice carrée symétrique $G[1..n][1..n]$, ou par la demi-matrice inférieure (les cellules (i, j) telles que $i \leq j$).

D'autre part, on veut associer une longueur lg à chaque arc. On aura bien sûr $(\forall i, j \in (1..n))$ [$lg(v_i, v_j) = lg(v_j, v_i)$].

Il y a un problème de modélisation du cas où il n'y a pas d'arc entre u et v. Mettre 0 ne peut pas aller, si on veut après parler de distances. [1] On convient ici de coder cette absence par la longueur -1 et de l'interpréter comme l'infini ($+\infty$). G_p devient une matrice sur $\mathbb{N} \cup \{\infty\}$.

ATTENTION : pour obtenir une distance δ sur l'ensemble des villes, au sens rigoureux du terme, on ne peut pas se contenter de prendre pour $\delta(u, v)$ le minimum de la somme des longueurs des arcs constituant un chemin de u à v ($c = u_0 \dots u_k$ avec $(\forall r < k)$ [$u_k \ A \ u_{k+1}$]. En effet, rien n'interdit (pour l'instant) de considérer un graphe à 3 nuds u, v, w, avec les poids $lg(u, v) = 3, lg(u, w) = 1, lg(v, w) = 1$, et l'inégalité du triangle n'est pas respectée.

Anneau min-plus

Dans un premier temps, codons les fonctions de base de l'anneau. On utilise l'anneau min-plus sur l'ensemble $\mathbb{N} \cup \{\infty\}$, l'opération somme étant le minimum, d'élément neutre $+\infty$, et l'opération produit étant la somme classique, d'élément

1. Pour que $\delta : N \times N \to \mathbb{N}$ soit une distance, il faut que δ vérifie les 3 axiomes suivants :
   - identité : $(\forall u, v \in N)$ [$\delta(u, v) = 0 \iff u = v$].
   - symétrie : $(\forall u, v \in N)$ [$\delta(u, v) = \delta(v, u)$].
   - inégalité du triangle : $(\forall u, v, w \in N)$ [$\delta(u, w) \leq \delta(u, v) + \delta(v, w)$].

neutre 0. On constate qu'on a bien

$$(\forall x, y, z \in \mathbb{N} \cup \{\infty\}) \ [\ x + \min(y, z) = \min(x + y, x + z)$$

Ajouter dans `anneau02.py` :

```
def unit():                          def zero():
   return −1                            return 0

def produit(a,b):                    def somme(a,b):
   if (a == -1 or b == -1)              if (a == -1 and b == -1)
     return -1                            return -1
   else                                 else
     return a+b                           return min(a,b)
```

Pourrions-nous inverser le rôle des opérations somme et produit (en changeant également unit et zero) ?

Réutiliser la fonction `affiche` de la feuille 1 de TP, en la modifiant pour remplacer les nombres négatifs ("−1") par des espaces (" "). Elle prendra désormais en entrée une relation pondérée (liste de liste d'entiers), ne retournera toujours pas de sortie, et imprimera un tableau.

Vous pourrez partir de la fonction `affiche` que vous avez écrite pour le TP1, en n'oubliant pas de modifier le contenu des print().

L'entrée

```
[[−1, 1 , 1 , 1 ,−1],
 [ 6 ,−1, 6 ,−1,−1],
 [−1,−1, 4 ,−1, 0 ],
 [−1,−1,−1, 7 ,−1]]
```

devrait donner

$$\begin{bmatrix} & 1 & 1 & 1 & \\ 6 & & 6 & & \\ & & 4 & & 0 \\ & & & 7 & \end{bmatrix}$$

Faire la somme de deux éléments consiste à comparer deux chemins parallèles et à choisir le plus court. Le produit consiste à prendre un chemin en deux parties, et à calculer la distance totale.

Opérations sur des relations pondérées

Ordre sur les relations binaires On considère l'ordre suivant : R est plus petite que S si

- toute arête de R appartient aussi à S,
- pour tout arc de R, son coefficient est plus grand que celui de son équivalent dans S.

Par exemple, $R = [[-1, 0, 1][2, 7, 5]]$ est plus petite que $S = [[-1, 0, 0][2, 5, 2][-1, 1, 18]]$.

En effet, les éléments (couples) de la première sont :

- (0,1) de valeur 0 et (1,0), de valeur 2, qui apparaissent aussi avec les même valeurs pour la seconde relation.
- (0,2) de valeur 1, (1,1) de valeur 7, et (1,2) de valeur 5, qui apparaissent aussi dans la seconde relation, et ceux de la première sont des entiers plus *petits* que ceux de la seconde.

[[ ]] est la plus petite relation, et $0^{N\times M}$ (tableau de taille $N \times M$ rempli par des 0) est la plus grande relation parmi celles de taille inférieure ou égale à $N \times M$.

Symétrie *Une relation binaire pondérée est symétrique si, pour toute arête de la relation, l'arête inverse appartient également à la relation et possède le même coefficient.*

$\Longrightarrow$ Écrire une fonction `testSymetrie(R)` qui prend en entrée une relation pondérée R et retourne un booléen indiquant si R est symétrique.

On se rappelle du TD précédent, où on parcourait les paires de sommets et où on testit la propriété sur chacun. Est-il nécessaire de tester toutes les arêtes ?

$\Longrightarrow$ Écrire une fonction `symetrie(R,somme)` qui prend en entrée une relation R et retourne la plus petite relation S symétrique et plus grande que R.

On rappelle que `Somme`(a, b) retourne ici le plus grand entier inférieur à a et à b.

Deux façons de faire sont possibles : comparer chaque arête avec son inverse, ou écrire une fonction `inverse(R)` qui prend en entrée une relation R et retourne R^{-1}, puis appeler `symetrie(R, somme) = union(R, inverse(R), somme)`.

$\Longrightarrow$ Facultatif : écrire une fonction `symetrie2(R)` qui prend en entrée une relation R et retourne la plus grande relation S symétrique plus petite que R. Vous pourrez utiliser max et ferez attention aux infinis.

Réflexivité *Une relation est réflexive s'il existe une boucle de poids* 0 *sur tout sommet.*

$\Longrightarrow$ Écrire une fonction `testReflexivite(R)` qui prend une relation en entrée et retourne un booléen vrai si la relation est réflexive et faux sinon. (Vous pouvez vous référer au TD).

$\Longrightarrow$ Écrire une fonction `reflexive(R)` qui prend une fonction en entrée et retourne la plus petite fonction réflexive qui la contient.

Alternativement, écrire une fonction `identite(n)` prenant en argument une taille d'ensemble, n, et retournant la relation identité sur $[1, n]$, puis appeler `reflexive(R, somme) = union(R,identite(len(R)),somme)`

$\Longrightarrow$ Tester les deux fonctions sur la relation vide de taille 3 ($[[-1,-1,-1],[-1,-1,-1],[-1,-1,-1]]$).

$\Longrightarrow$ Exécuter `testreflexive` sur `reflexive([[-1,-1,-1],[-1,-1,-1],[-1,-1,-1]])`.

Génération aléatoire Importer `random` et initialiser la graine (`random.seed()`).

On sait que `random.randint(a,b)` retourne un nombre aléatoire uniforme compris entre a et b, tous deux inclus.

$\Longrightarrow$ Écrire une fonction `mapgen(s,m=10,p=1,q=1)` qui prend en entrée quatre entiers :

le nombre de sommets, i.e. la taille de la relation
une borne sur les distance
deux entiers p et q codant le rationnel $\frac{p}{q}$, qui sera la proportion de distances non infinies ($\neq -1$).

et retourne une relation de l'ensemble $[0, s-1]$ dans lui-même, dont chaque élément a une probabilité $\frac{p}{q}$ d'être choisi uniformément dans $[0, m]$, et vaut ∞ sinon.

$\Longrightarrow$ Tester et afficher votre fonction.

Clôtures

La clôture réflexive et transitive d'une relation R est la plus petite relation qui contient R et est réflexive et transitive.

Puisque la relation pleine (toute paire de sommets est indexée par 0) contient toute les autres et est réflexive et transitive, et que l'intersection de deux relations réflexives et transitives l'est également, une telle clôture existe toujours et est bien définie.

Suite récurrente $\Longrightarrow$ Récupérer ou recoder les fonctions Union et composition du TP précédent.

$\Longrightarrow$ Vérifier la compatibilité avec l'anneau de ce TP.

$\Longrightarrow$ Quelle est la taille du plus long chemin simple (ne passant pas deux fois par un même sommet) possible dans un graphe ?

$\Longrightarrow$ Si vous avez un graphe G, et si vous ajoutez à G les fusions d'arêtes, sans retirer celles consommées, pour obtenir $G^{(2)} = G.G \cup G$, $G^{(3)} = G^{(2)}.G \cup G \cdots$, et si vous répétez l'opération autant que la taille du plus long chemin, reste-t-il des arêtes dont la fusion n'est pas déjà dans le graphe ?

Que peut on en déduire de $G^{(n)}$?

Existe-t-il un graphe transitif contenant G mais ne contenant pas $G^{(2)}$?
Même question avec $G^{(3)}$, etc., $G^{(n)}$?
Qu'en déduit-on pour $G^{(n)}$?

$\Longrightarrow$ Utiliser `union`, `reflexive`, `composition`, et la réponse aux questions précédentes pour écrire une fonction `transcloture(R,union, intersection, reflexive)` qui retourne la clôture réflexive et transitive de R.

$\Longrightarrow$ La tester sur des graphes générés aléatoirement avec les paramètres $10, 10, 1, 3$.