

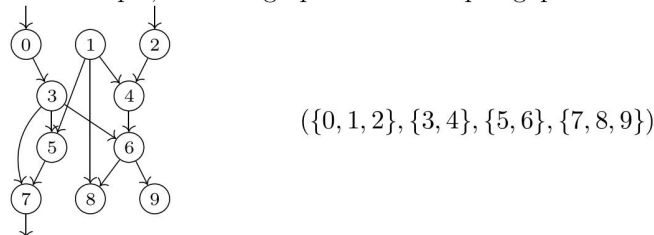
Projet Info LDD2 – TD 8

Objectifs du TD : Tri topologique, chemin le plus long, mixins.

La notion de chemin le plus long est un tout petit peu plus subtile que celle du chemin le plus court (vu au TD précédent). En particulier, dans les graphes cycliques, il faut faire attention au nombre d'occurrences des arêtes dans les chemins, sinon on pourrait utiliser les cycles pour augmenter indéfiniment leur taille. Même en faisant attention à cet aspect, le problème reste compliqué à résoudre efficacement (en fait il est NP-dur, i.e. on ne sait pas le résoudre plus efficacement qu'avec un algorithme exponentiel).

Heureusement, le problème devient simple dans les graphes acycliques. Une façon de procéder est de d'abord réaliser un *tri topologique*. Un tri topologique est un ordre (\prec) donné sur les noeuds du graphes tel que si (u, v) est une arête (orientée) du graphe, alors $u \prec v$. Une telle notion n'existe pas s'il y a des cycles, et un même graphe peut souvent donner plusieurs tris topologiques. On propose ici de calculer un tri topologique "compressé vers le haut". On peut représenter ce tri par une séquence d'ensembles $(\ell_i)_i$, tel que $i < j \implies (\forall u \in \ell_i, \forall v \in \ell_j, u \prec v)$, et tel que tout noeud de ℓ_{i+1} a au moins un parent dans ℓ_i . Les ensembles ℓ_i doivent partitionner les noeuds du graphe.

Par exemple, voici un graphe et le tri topologique obtenu :



(On n'a pas représenté les noeuds qui servent d'entrées ou de sorties. On les ignorera ici : on fera comme s'ils n'existaient pas.)

On peut faire cela assez simplement, en réutilisant des idées du test de cyclicité. Le premier ensemble est en fait composé des toutes les "co-feuilles" du graphe (i.e. les noeuds sans parents). Ensuite, si on ôte ces co-feuilles, l'ensemble suivant sera composé des co-feuilles du nouveau graphe, etc...

▮ **Exercice 1 :** Implémenter une méthode qui implémente ce tri topologique. On peut par ailleurs ce faisant détecter si le graphe est cyclique (ce qui arrive

s'il n'y a plus de co-feuilles mais que le graphe est non-vide). Renvoyer alors une erreur. ┘

On a choisi particulièrement ce tri topologique car il permet facilement d'obtenir une notion de profondeur des noeuds du graphe et du graphe lui-même. Si $u \in \ell_i$, sa profondeur est i . La profondeur du graphe est le maximum des profondeurs de ses sommets, c'est donc plus simplement le nombre des ensembles ℓ_i .

┐ **Exercice 2 :** Implémenter une méthode qui retourne la profondeur d'un noeud donné dans un graphe.
Implémenter ensuite une méthode qui calcule la profondeur d'un graphe. ┘

Le dernier exercice va consister à (enfin) calculer le plus long chemin d'un noeud vers un autre (en considérant qu'on est dans un graphe acyclique).

Attention : il ne suffit pas de faire la différence de profondeur entre les deux noeuds. Par exemple, dans le graphe ci-dessus, la longueur du chemin le plus long de 1 à 5 est bien 1 et non pas 2.

La méthode qu'on va utiliser utilise le tri topologique (et fonctionnerait aussi avec une petite variation pour le calcul du chemin le plus court).

Supposons qu'on veuille calculer le chemin de u vers v dans un graphe dont on connaît un tri topologique $(\ell_i)_i$. On cherche déjà ℓ_k tel que $u \in \ell_k$. Ensuite, pour tous les noeuds w dans ℓ_{k+1} , puis ℓ_{k+2} , puis ..., et tant que $w \neq v$, on va remplir $dist[w]$ et $prev[w]$ en fonction de leur valeur pour les parents de w . I.e. si aucun des parents de w n'est dans $dist$, ça veut dire que u n'est pas un ancêtre de w , donc on peut ne rien changer. Sinon, on choisit le parent de $dist$ maximum, on affecte cette valeur $+1$ à $dist[w]$, et on stocke dans $prev[w]$ le parent en question.

┐ **Exercice 3 :** Implémenter une méthode qui calcule le chemin et la distance maximaux d'un noeud u à un noeud v . ┘

Organisation des fichiers


À ce stade du projet le fichier `open_digraph.py` doit être plutôt bien rempli. On peut avoir envie de séparer les méthodes d'une classe dans différents fichiers si celle-ci devient longue. Pour ce faire, on peut utiliser des *mixins*. C'est un concept très proche de celui de l'héritage, à tel point que la syntaxe en Python est la même.

Supposons par exemple que je veuille mettre les méthodes qui ont trait aux compositions dans un fichier à part, disons dans `open_digraph_compositions_mx.py`. Il faut créer une classe `open_digraph_compositions_mx` par exemple dans laquelle on met les méthodes en question, et ensuite charger cette classe lors de la définition de `open_digraph`, i.e. il faut importer le fichier du mixin et définir :

```
class open_digraph(open_digraph_compositions_mx):  
    ...
```

On peut évidemment charger plusieurs mixins, séparés par des virgules. Si l'on en a beaucoup, on peut envisager de créer un sous-dossier de `modules` qui ne va contenir que les mixins de `open_digraph` par exemple.

「 **Exercice 4 :** Arranger son code en utilisant des mixins. 」

 Dans les mixins, on ne pourra pas faire appel à `open_digraph` littéralement. Si on en a besoin, on pourra utiliser `cls` dans le cas des méthodes de classe, ou bien `self.__class__` dans le cas des méthodes habituelles.