


# Projet Info LDD2 – TD 2

Renaud Vilmar -- vilmar@lsv.fr

**Objectifs du TP :** Début de manipulation des graphes.

 Ce TD est noté, il est donc à finir, et l'état du projet à rendre, avant la séance du TD 3 (soit par mail, soit via eCampus).

Le TD précédent devrait être fini avant de poursuivre avec celui-ci. On s'était arrêté sur une fonction qui permettait d'ajouter un noeud dans le graphe. Maintenant, on va s'arranger pour pouvoir en retirer.

Au passage, comme les getters et setters sont en place, on va en général éviter d'accéder aux attributs d'une instance d'une classe directement. Si on veut connaître l'id d'un noeud `n`, par exemple, on va utiliser `n.get_id()` plutôt que `n.id`. C'est moins concis, mais plus robuste.

▮ **Exercice 1 :** Implémenter dans `node` les méthodes :

- `remove_parent_once` et `remove_child_once`
- `remove_parent_id` et `remove_child_id`

Dans les deux premières, on retire 1 occurrence (1 multiplicité) de l'id donné en paramètre. Dans les deux dernières, on retire *toutes* les occurrences de l'id. Dans les deux cas, lorsque la multiplicité tombe à 0, retirer la clé du dictionnaire (pour gagner en espace). ▮

▮ **Exercice 2 :** Implémenter dans `open_digraph` les méthodes :

1. `remove_edge(self, src, tgt)` (où `src` et `tgt` sont des ids)
2. `remove_parallel_edges(self, src, tgt)` (où `src` et `tgt` sont des ids)
3. `remove_node_by_id`

Petite aide : `x = d.pop(k)` permet de retirer la clé `k` du dictionnaire `d`, et de stocker la valeur associée dans `x`.

La première fonction doit retirer 1 arête seulement, la deuxième toutes les arêtes de `src` vers `tgt`. La dernière fonction devrait retirer les arêtes associées au noeud comme il faut.

Rajouter trois méthodes `remove_edges`, `remove_parallel_edges` et `remove_nodes_by_id` qui généralisent les deux précédentes pour traiter directement plusieurs arêtes/noeuds. On peut également préférer faire du 2-en-1 en utilisant `*args`.

Note : on attend comme argument de `remove_edges` et `remove_parallel_edges` quelque chose comme une liste de paires `(src,tgt)`. ▮

▮ **Exercice 3 :** Pour s'assurer qu'on n'obtient pas n'importe quoi après avoir manipulé un graphe, il peut être intéressant d'implémenter une méthode `is_well_formed` qui vérifie qu'un graphe est toujours "bien formé". Pour être bien formé, il doit vérifier les propriétés suivantes :

- chaque noeud d'**inputs** et d'**outputs** doit être dans le graphe (i.e. son **id** comme clé dans **nodes**)
- chaque noeud input doit avoir un unique fils (de multiplicité 1) et pas de parent
- chaque noeud output doit avoir un unique parent (de multiplicité 1) et pas de fils
- chaque clé de **nodes** pointe vers un noeud d'**id** la clé
- si **j** a pour fils **i** avec multiplicité **m**, alors **i** doit avoir pour parent **j** avec multiplicité **m**, et vice-versa

」

「 **Exercice 4 :** Définir la méthode **add\_input\_node** qui crée un nouveau noeud qu'on placera en input, et qui pointe vers le noeud dont a donné l'**id** en paramètre. (On peut laisser le label du nouveau noeud vide). Définir de façon similaire la méthode **add\_output\_node**.

Il y a une précaution à prendre pour s'assurer que le graphe reste bien-formé. La trouver et l'implémenter (renvoyer une erreur si la condition n'est pas respectée).

」

「 **Exercice 5 :** Effectuer quelques tests pour vérifier que :

- **is\_well\_formed** accepte les bons graphes, et rejette ceux qui sont mal formés
- rajouter ou retirer un noeud laisse un graphe bien-formé
- ajouter ou retirer une arête laisse un graphe bien-formé (si elle ne concerne pas des noeuds inputs/outputs)
- ajouter une entrée/sortie via les méthodes de l'exo précédent laisse un graphe bien-formé

」