Projet Info LDD2 – TD 1

Renaud Vilmart -- vilmart@lsv.fr

Objectifs du TP : Mise en place, implémentation de graphes, début de manipulation.

Pour la séance suivante, il vous sera demandé de former un binôme ou trinôme, qui sera votre groupe de travail durant ce projet. Vous pouvez décider de cela tout de suite, auquel cas vous pouvez réaliser ce TD sur une seule machine par groupe.

Exercice 1 :

1. Créer 2 dossiers : `modules/` et `tests/`
2. Créer les fichiers :
   - `open_digraph.py` dans `modules/`
   - `open_digraph_test.py` dans `tests/`
   - `worksheet.py` à la racine
3. Tester l'exécution : faire un "hello world" dans la worksheet :

```
print("hello world")
```

Exécuter le fichier :

```
python3 worksheet.py
```

Exercice 2 : Dans `open_digraph`, définir les deux classes :

```
class node:

    def __init__(self, identity, label, parents, children):
        '''
        identity: int; its unique id in the graph
        label: string;
        parents: int->int dict; maps a parent node's id to its multiplicity
        children: int->int dict; maps a child node's id to its multiplicity
        '''
        self.id = identity
        self.label = label
        self.parents = parents
        self.children = children

class open_digraph: # for open directed graph

    def __init__(self, inputs, outputs, nodes):
        '''
        inputs: int list; the ids of the input nodes
```

```
        outputs: int list; the ids of the output nodes
        nodes: node iter;
        '''
        self.inputs = inputs
        self.outputs = outputs
        self.nodes = {node.id:node for node in nodes} # self.nodes: <int,node> dict
```

Exercice 3 :

1. Dans `open_digraph_test.py`, importer la librairie `unittest` et le "module" `open_digraph`

```
import sys
import os
root = os.path.normpath(os.path.join(__file__, './../..'))
sys.path.append(root)# allows us to fetch files from the project root
import unittest
from modules.open_digraph import *
```

2. Créer une classe pour le test des méthodes `__init__` de nos deux classes :

```
class InitTest(unittest.TestCase):

    def test_init_node(self):
        n0 = node(0, 'i', {}, {1:1})
        self.assertEqual(n0.id, 0)
        self.assertEqual(n0.label, 'i')
        self.assertEqual(n0.parents, {})
        self.assertEqual(n0.children, {1:1})
        self.assertIsInstance(n0, node)


if __name__ == '__main__': # the following code is called only when
    unittest.main()        # precisely this file is run
```

On peut lancer le test en exécutant directement, à partir du dossier `tests`, le fichier `open_digraph_test.py`.
On peut également chercher et exécuter tous les tests dans un dossier :

```
python -m unittest discover <dir> "*_test.py"
```

3. Définir une deuxième méthode de test pour l'`__init__` de `open_digraph`.

Ces tests servent à structurer le projet, à déterminer où se trouvent les problèmes si on casse quelque chose en changeant du code. Ils permettent également de montrer aux collaborateurs du projet comment s'utilisent les fonctions/méthodes implémentées (c'est le cas ci-dessus, même si ces premiers tests

n'ont pas l'air de vérifier grand chose). Ils sont complémentaires aux commentaires et à la doc.

On pourra faire une nouvelle classe pour tester les méthodes de `node`, et une autre pour tester les méthodes de `open_digraph` lorsque demandé. On pourra alors utiliser `setUp` pour factoriser les affectations. Par exemple :

```
class NodeTest(unittest.testcase):

    def setUp(self):
        self.n0 = node(0, 'a', [], [1])

    def test_get_id(self):
        self.assertEqual(self.n0.get_id(), 0)

    def test_get_label(self):
        self.assertEqual(self.n0.get_label(), 'a')
```

Exercice 4 :

1. Dans `worksheet.py`, importer les classes :

   ```
   from modules.open_digraph import *
   ```

   créer un petit graphe quelconque, et tenter de le printer. Que se passe-t-il ?
2. Pour avoir quelque chose de plus lisible, on peut définir la méthode `__str__`, qui doit retourner la chaîne de caractères utilisée pour printer l'instance. Créer cette méthode pour les deux classes.
3. Pour afficher un élément inductivement (par exemple dans une liste), on utilise `__repr__`. A nouveau, écrire cette méthode pour les deux classes.

⌟

Exercice 5 : Implémenter dans la classe `open_digraph` la *méthode de classe* `empty` (avec l'annotation `@classmethod`) qui renvoie un graphe vide. ⌟

Exercice 6 (test requis) **:** Implémenter les méthodes `copy` pour les deux classes, qui permettent de copier des instances de celles-ci. Bien vérifier en test que modifier une copie d'une instance de modifie pas l'instance originelle (on peut utiliser par exemple `assertIsNot(x.copy(),x)`). ⌟

Exercice 7 : Implémenter des "getters" :

— Pour `node` :
`get_id`, `get_label`, `get_parent_ids`, `get_children_ids`

— Pour `open_digraph` :
`get_input_ids`, `get_output_ids`,
`get_id_node_map` (renvoie un dictionnaire `id:node`),
`get_nodes` (renvoie une liste de tous les noeuds),
`get_node_ids`, `get_node_by_id`,
`get_nodes_by_ids` (renvoie une liste de noeuds à partir d'une liste d'ids)

Demander au chargé de TD si le nom des méthodes n'est pas clair.
On peut remplacer si on veut `get_node_by_id` par `__getitem__`, qui sera appelé lorsqu'on fera `G[_id]` par exemple. ⌟

⌜ **Exercice 8 :** Implémenter des "setters" :
- Pour `node` :
  `set_id`, `set_label`, `set_parent_ids`, `set_children_ids`, `add_child_id`, `add_parent_id`,
- Pour `open_digraph` :
  `set_input_ids`, `set_output_ids`, `add_input_id`, `add_output_id`

⌟

⌜ **Exercice 9 :** Dans la worksheet, ou bien en interactif, importer le module `inspect` et le module `open_digraph` sur lequel on travaille. Printer la liste des méthodes de nos deux classes (en utilisant `dir`).

Utiliser les fonctions du module `inspect` pour afficher le code source d'une des méthodes, sa doc, et le fichier dans lequel il se trouve. ⌟

Dans ce premier TD, jusqu'ici, on n'aura pas vraiment fait d'algorithmique, mais on a préparé le terrain pour la suite. On doit maintenant avoir un environnement fonctionnel, avec quelques fichiers qui intéragissent proprement, et on sait communiquer et interagir avec la structure de données qu'on a mise en place.

On va maintenant commencer quelque chose de plus intéressant : ajouter des noeuds dans un graphe.

⌜ **Exercice 10 :** Pour pouvoir rajouter un noeud au graphe, il faut s'assurer de lui assigner un `id` non utilisé. Définir une méthode `new_id` qui renvoie un `id` non utilisé dans le graphe.
Remarque : il y a plein de moyens de faire ça, on peut également rajouter un attribut à la classe `open_digraph` pour s'aider. ⌟

⌜ **Exercice 11 :** Ecrire une méthode `add_edge(self, src, tgt)` qui rajoute une arête du noeud d'`id tgt` au noeud d'`id src`. Ecrire une seconde méthode `add_edges(self, edges)` où `edges` est une liste de paires d'ids de noeuds, et qui rajoute une arête entre chacune de ces paires. ⌟

⌜ **Exercice 12 :** Ecrire une méthode `add_node(self, label='', parents=None, children=None)` qui rajoute un noeud (avec `label`) au graphe (en utilisant un nouvel `id`), et le lie avec les noeuds d'ids `parents` et `children` (avec leurs multiplicités respectives). Si les valeurs par défaut de `parents` et/ou `children` sont `None`, leur attribuer un dictionnaire vide. Renvoyer l'id du nouveau noeud. ⌟

Un noeud peut ne pas avoir de parent, sans pour autant être un input, idem pour les outputs. On fera des méthodes spécifiquement pour ajouter des inputs/outputs.

⌜ **Exercice 13** (À finir pour la prochaine séance) **:**

1. créer un binôme/trinôme
2. avoir accès à un repo git privé par groupe (sur GitHub/Gitlab, créer un repo privé, puis inviter des collaborateurs)
3. avoir cloné le repo :

```
git clone <url du repo>
```

⌟

Pour la suite, on ne va donc garder qu'un seul projet par groupe, et qui sera sur le repo distant.

 Lorsque vous avez fini de travailler (voire à intervalles plus réguliers), ne pas oublier de propager vos changements avec :

```
git add . # or add <files>
git commit -m "<message du commit>"
git pull
git push
```