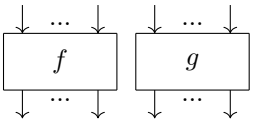
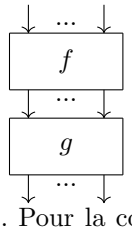
Projet Info LDD2 – TD 6

Renaud Vilmart -- vilmart@lsv.fr

Objectifs du TD : Compositions & connectivité ; organisation des fichiers.

En considérant les circuits booléens comme des processus, on a deux façons de les composer :

1) en parallèle : 

2) en séquence : 

.

Il n'y a aucune restriction pour la composition parallèle. Pour la composition séquentielle, il faut que le nombre d'entrées du deuxième circuit coïncide avec le nombre de sorties du premier (de la même façon que dans $f \circ g$, le résultat de g doit être un argument viable pour f i.e. l'image de g doit être dans le domaine de f). On va dans un premier temps implémenter ces deux compositions.

En pratique, pour faire ces compositions dans notre implémentation des graphes, il va d'abord falloir gérer les indices, pour s'assurer qu'il n'y ait pas de "chevauchement". Par exemple si les deux graphes ont un noeud d'indice 0, il va falloir modifier toutes les occurrences de 0 dans l'un des deux graphes. Une façon facile de procéder est de chercher M l'indice max d'un des deux graphes, et m l'indice min de l'autre, et de "translater" tous les indices du deuxième graphe de $M - m + 1$.

⌜ **Exercice 1 :** Implémenter les méthodes `min_id` et `max_id` qui renvoient respectivement l'indice min et l'indice max des noeuds du graphe. ⌟

⌜ **Exercice 2** (test requis) **:** Implémenter dans `open_digraph` une méthode `shift_indices (self,n)`, qui va ajouter `n` à tous les indices du graphe.

⌟

⌜ **Exercice 3 :** Dans `open_digraph`, implémenter une méthode `iparallel (self,g)` qui ajoute `g` à `self` (qui modifie donc `self`). `g` ne doit pas être modifié.

Implémenter une deuxième méthode `parallel` qui renvoie la composition parallèle des deux graphes, mais sans les modifier. ⌟

⌜ **Exercice 4 :** Implémenter une méthode `icompose(self, g)` qui fait la composition séquentielle de `self` et `g` (les entrées de `self` devront être reliées aux sorties de `g`). Lancer une exception dans le cas où les nombres d'entrées (de `self`) et de sorties (de `g`) ne coïncident pas. `g` ne doit pas être modifié.

Implémenter une deuxième méthode `compose` qui renvoie la composée des deux graphes, sans les modifier. ⌟

Une fonction qui en un sens fait l'inverse d'une composition parallèle sépare un circuit en ses composantes connexes. En terme de processus, deux composantes connexes d'un circuit booléen sont deux sous-programmes qui n'interagissent pas et peuvent donc tourner en parallèle.

⌜ **Exercice 5 :** Implémenter une méthode `connected_components` d'`open_digraph` qui renvoie le nombre de composantes connexes, et un dictionnaire qui associe à chaque `id` de noeuds du graphe un `int` qui correspond à une composante connexe.

Par exemple, si `g` a 4 composantes connexes, on peut les numéroter de 0 à 3. Alors pour chaque noeud `n`, on aura dans le dictionnaire la paire `n.id : k` si `n` est dans la composante connexe numérotée `k`. ⌟

⌜ **Exercice 6 :** Implémenter une méthode de `open_digraph` qui renvoie une liste d'`open_digraph`s, chacun correspondant à une composante connexe du graphe de départ. ⌟

⌜ **Exercice 7** (test requis) **:** Modifier `iparallel` et `parallel` pour prendre en entrée une liste de graphes. ⌟

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 8 :** Arranger son code en utilisant des mixins. ⌟