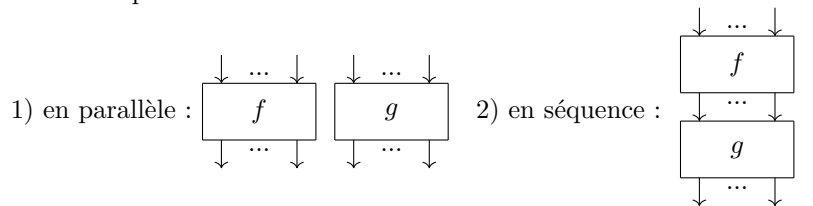


# Projet Info LDD2 – TD 6

Renaud Vilmart -- vilmart@lsv.fr

## Objectifs du TD : Compositions & connectivité.

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



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  $g \circ f$ , le résultat de  $f$  doit être un argument viable pour  $g$  i.e. l'image de  $f$  doit être dans le domaine de  $g$ ). 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  $M - m + 1$ . Les deux derniers exercices du TD précédent devraient permettre de réaliser cela.

▮ **Exercice 1** (tests requis) : 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 un nouveau graphe qui est la composition parallèle des deux graphes en paramètre, mais sans modifier ces derniers. ▮

L'élément neutre pour la composition parallèle est le graphe vide. Celui-ci fait déjà partie des méthodes de `open_digraph`.

▮ **Exercice 2** (tests requis) : Implémenter une méthode `icompose(self, f)` qui fait la composition séquentielle de `self` et `f` (les entrées de `self` devront être reliées aux sorties de `f`). Lancer une exception dans le cas où les nombres d'entrées (de `self`) et de sorties (de `f`) ne coïncident pas. `f` ne doit pas être modifié.

Implémenter une deuxième méthode `compose` qui renvoie un troisième graphe qui est la composée des deux autres, sans modifier ces derniers. ▮

L'élément neutre pour la composition séquentielle est l'identité sur  $n$  fils (elle

doit être définie pour tout  $n$ ).

「 **Exercice 3 :** Implémenter une méthode de classe `identity(cls, n)` qui crée un `open_digraph` représentant l'identité sur  $n$  fils. 」

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 4 :** 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 5 :** Implémenter une méthode de `open_digraph` qui renvoie une liste d'`open_digraphs`, chacun correspondant à une composante connexe du graphe de départ. 」