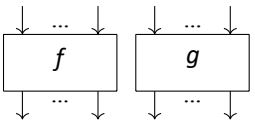
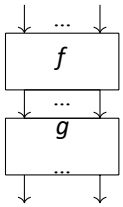
LDD2 Info Project: Session 6

Aims of this session: To learn composition and connectivity.

If we think of Boolean circuits as processes, there are two ways of composing them:

1) Parallel: 

2) Sequence: 

.

There are no restrictions for parallel composition. For composition in sequence, the number of inputs of the second circuit must coincide with the number of outputs of the first (in the same way as in $g \circ f$, the result of f must be a viable argument for g i.e. the image of f must be in the domain of g). We will first implement these two compositions.

To apply these compositions in our graph implementation, we first have to manage the indices and ensure that there is no overlap. For example, if both graphs have a node with index 0, we will have to modify all occurrences of 0 in one of the two graphs. An easy way of doing this is to find M, the max index of one of the two graphs, and m, the min index of the other, and to "translate" all the indices of the second graph by M - m + 1. The last two exercises in the previous tutorial should enable you to do this.

Exercise 1 (tests required):
In open_digraph, implement a method iparallel(self,g) which appends, in parallel, g to self (thereby modifying self). However, g must not be modified.

Implement a parallel method, which returns a new graph which is the parallel composition of the two parameter graphs, without modifying them.

Note that the neutral element for parallel composition is the empty graph. This is already part of the open_digraph methods.

Exercise 2 (tests required):
Implement, in open_digraph, a icompose(self, f) method, which performs the sequential composition of self and f (the inputs of self should be connected to the outputs of f). The method should raise an exception if the numbers of inputs from self and outputs from f do not match. f must not be modified.

Implement a compose method, returning a third graph, which is the composition of the other two, without modifying them.

Note that the neutral element for the sequential composition is the identity over n children (it should be defined for all n).

Exercise 3:
Implement a class method identity(cls, n), which creates an open_digraph representing the identity over n children.

A function that, in some sense, does the "opposite" of a parallel composition, is one that separates a circuit into its connected components. In process terms, two connected components of a Boolean circuit are two sub-programs that do not interact and can therefore run in parallel.

Exercise 4:
Implement a connected_components method (in open_ digraph), which returns the number of connected components of a graph, together with a dictionary that associates an int, corresponding to a connected component, with each node id in the graph. The idea is to sort out the nodes depending on which connected component they belong to. For example, say g has 4 connected components. These can be numbered from 0 to 3. Then, for each node n, the dictionary will contain the pair n.id: k, with k from 0 to 3, if n is in the connected component labeled by k.

Exercise 5:
In open_digraph, implement a method that takes a graph as an input and returns a list of open_digraphs, each corresponding to a connected component of the starting graph.