LDD2 Info Project: Session 8

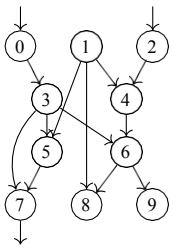
Aims of the tutorial: To learn topological sorting, longest paths, mixins.

The notion of the longest path is a tiny bit more subtle than that of the shortest path (seen in the previous session). In particular, for cyclic graphs we have to be careful about the number of occurrences of edges in paths, otherwise we could use cycles to increase their size indefinitely. Even with this in mind, the problem is difficult to solve efficiently (in fact it is NP-hard, i.e. the complexity of the best-known algorithm grows exponentially with the number of inputs).

Fortunately, the problem becomes simple in acyclic graphs. One way to proceed is to first perform a *topological sort*. A topological sort is a total order ($<$) given on the nodes of the graph, subjected to the constraint that, if (u, v) is an (oriented) edge of the graph, then $u < v$. Such a notion does not exist if there are cycles. For directed acyclic graphs, there is always at least one topological sort.

Here we propose to perform an "upwardly compressed" topological sort. This sort can be represented by a sequence of sets $(S_i)_i$, such that $i < j \Rightarrow (\forall u \in I_i , \forall v \in S_j , u < v)$, and such that every node of S_{i+1} has at least one parent in S_i. The sets S_i must partition the nodes of the graph.

For example, here is a graph and the topological sort obtained:



({0, 1, 2}, {3, 4}, {5, 6}, {7, 8, 9})

(The nodes that serve as inputs or outputs have not been represented. We'll ignore them here: we'll pretend they don't exist).

This can be done quite simply, using ideas from the cyclicity test. The first set is in fact made up of all the 'co-leaves' of the graph (i.e. nodes with no parents). Then,

if we remove these co-leaves, the next set will be composed of the co-leaves of the new graph, etc….

Exercise 1:
Construct a method that implements this topological sorting. By doing so, we can also detect whether the graph is cyclic (which happens if there are no more co-leaves but the graph is non-empty). In such case, return an error.

We have chosen this topological sort in particular because it makes it easy to obtain a notion of the depth of the nodes of the graph and of the graph itself. If $u \in S_i$, its depth is i. The depth of the graph is the maximum of the depth of its vertices, so it is more simply the number of sets S_i.

Exercise 2:
Implement a method that returns the depth of a given node in a graph.
Then implement a method that calculates the depth of a graph.

The following exercise consists of (finally) calculating the longest path from one node to another (assuming we are in an acyclic graph). Note that, in doing so, it is not enough to take the difference between the depths of the two nodes. For example, in the graph above, the length of the longest path from 1 to 5 is 1 and not 2.

The method we're going to use uses topological sorting (and would also work with a small variation for calculating the shortest path). Suppose we want to compute the path from u to v in a graph for which we know a topological sort $(S_i)_i$. We first look for S_k such that $u \in S_k$. Then, for all nodes w in S_{k+1}, then S_{k+2}, then ..., and as long as w is not v, we will fill $dist[w]$ and $prev[w]$ according to their values for w's parents. I.e. if *none* of the parents of w is in $dist$, it means that u is not a parent of w, so we can change nothing. Otherwise, we choose the maximal parent of $dist$, assign this value +1 to $dist[w]$, and store in $prev[w]$ the parent in question.

Exercise 3:
Implement a method that calculates the maximum path and distance from a node u to a node v.

File organisation
At this stage of the project, the open_digraph.py file should be quite full. You may want to split the methods of a class into different files if it becomes long. To do this, you can use *mixins*, as explained below.

Suppose I wanted to put all the methods regarding to compositions of graphs in a separate file, say in open_digraph_compositions_ mx.py. You need to create an open_digraph_compositions_mx class, for example, in which you put the methods in question, and then load this class when you define open_digraph, i.e. you need to import the mixin file and define :

```
class open_digraph(open_digraph_compositions_mx):
    ...
```

You can obviously load several mixins, separated by commas. If you have a lot of them, you might consider creating a sub-folder of modules containing only the open_digraph mixins, for example.

Exercise 4:
Arrange your code using mixins.

Note that, in mixins, you won't be able to use open_digraph literally. If you need to, you can use cls in the case of class methods, or self. __class__ in the case of the usual methods.