LDD2 Info Project: Session 7

Aims of this session: to calculate path lengths.

A path in a Boolean circuit corresponds to the route taken by the information. Notions such as shortest or longest path, depth of a node, (smallest) common ancestor, (largest) common descendant, etc. all have a meaning in this domain.

The length of the shortest path from u to v represents the minimum time for the information (or part of this information) in u at a given moment to reach v. The length of the longest path, on the other hand, can tell us the point at which the information in u at an instant t ceases to influence that in v. I leave it to you to find a meaning for the notions of common ancestors and common descendants.

The notion of depth is particularly important: the depth of the circuit (which we will define more precisely in the following) represents, in a sense, its complexity. It's a metric we'd like to be able to minimise (another is simply the number of nodes in the graph).

An important algorithm that simplifies many of these calculations is Dijkstra's algorithm. A common variant of it calculates the tree of shortest paths from a given node.

We will often apply it to directed acyclic graphs, but not always. We therefore want a variant of this algorithm which also takes into account the orientation of the graph's edges. The algorithm adapted to our needs is given in pseudo-code below. There, `direction` defines the notion of neighbourhood. `None` means that we search both parents and children, -1 only in the parents, and 1 only in the children. So, applying the algorithm on u with `direction=-1` must browse only the parents of u.

Exercise 1 (Tests Required):
Implement Dijkstra's algorithm in open_digraph.

Tip: for finding the min, we can use min(*l*, *key=f*), which returns a $u \in l$ such that $f(u) \leq f(v)$ for all $v \in l$; i.e. which returns the (one) u that minimises f.

Now, if we are simply looking for the distance (and shortest path) between two given nodes, we can make the algorithm stop earlier.

Exercise 2:
Modify the algorithm by adding tgt=None as an argument, so that if the latter is specified, we return *dist* and *prev* directly as soon as we know the shortest path from src to tgt.
Use this method to implement shortest_path, which calculates the shortest path from u to v.

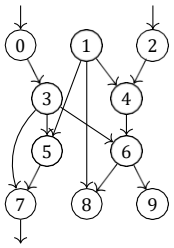
Dijkstra's algorithm

```
function Dijkstra(src, direction=None)
    Q ← [src]
    dist ← {src: 0}
    prev ← {}
    while Q != [] do
        u ← node id in Q with min dist[u]
        remove u from Q
        neighbours ← the neighbours of u subject to direction
        for all v ∈ neighbours do
            if v !∈ dist then
                add v to Q
            end if
            if v !∈ dist or dist[v] > dist[u] + 1 then
                dist[v] ← dist[u] + 1
                prev[v] ← u
            end if
        end for
    end while
    return dist, prev
end function
```

- src is the node from which we calculate the distances
- direction ∈ {None, -1, 1}

Exercise 3:
Implement a method which, given two nodes, returns a dictionary which associates each common ancestor of the two nodes with its distance from each of the two nodes. For example, in the graph given below, the algorithm applied to nodes 5 and 8 should return : {0 : (2, 3), 3 : (1, 2), 1 : (1, 1)}. (See illustration below.) A priori, one should use Dijkstra's algorithm.

Example:
In :



the method of exercise 3 applied to 5 and 8 should give {0 : (2, 3), 3 : (1, 2), 1 : (1, 1)}.