LDD2 Info Project: Session 9

Aims of the tutorial: Circuit synthesis using a propositional formula.

In this Session, we will construct a boolean circuit from a propositional formula, given in the form of a chain of characters.To make life easier, we'll assume that the formula is given in completely parenthesised form, and in infixed notation. For example:

"((x0) & ((x1) & (x2))) | ((x1) & (~ (x2))) ".

On the one hand, this avoids having to set up the rules of calculation priority, and it also simplifies *parsing* of the formula.

The first objective is to create the formula tree, which we will represent by a bool_circ with the nodes containing the logical connectors or the variable names as labels (as these variable names are not valid labels, we will remove them afterwards). The algorithm for doing this is given below.

Algorithm: Propositional formula to tree

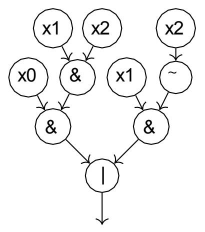
```
function PARSE PARENTHESES                                Note: s is a string

    g ← $\overset{\circ}{\downarrow}$ (i.e. the boolean circuit has a node connected to an output)
    current_node ← the id of the top node
    s2 ← ‘ ’
    for all char in s do
        if char = ‘(’ then
            add s2 to the label of current_node
            create a parent of current_node and make it current_node
            s2 ← ‘ ’
        else if char = ‘)’ then
            add s2 to the label of current_node
            change current_node so that it becomes its child
            s2 ← ‘ ’
        else
            add char to the end of s2
        end if
    end for
    return g
end function
```

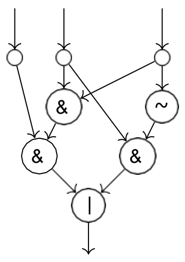
Exercise 1:
Implement this bool_circ method.

For instance, this is what we should obtain from the above formula:

“((x0) & ((x1) & (x2))) | ((x1) & (~ (x2))) ” ·········>



Technically, this is not a Boolean circuit, but we'll take care of that in what follows. As a reminder, here's what we're trying to achieve:

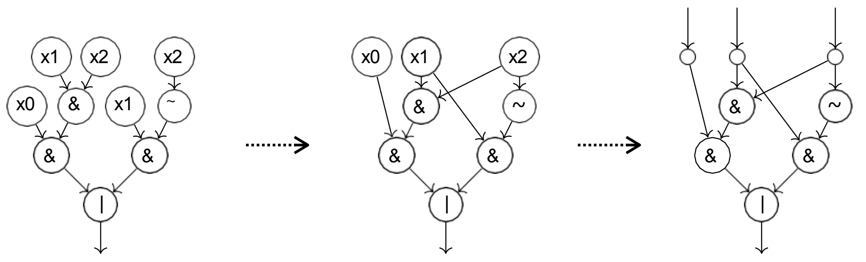


Note that we have a single-output copy operator. This is not surprising, it simply corresponds to information that has been copied *0 times*.

We now need to group the different occurrences of variables into a single copy node, linked to an input. To do this, a useful tool is to merge two nodes.

Exercise 2: Implement an open_digraph method that merges two nodes whose ids are given as parameters. You can choose one of the two labels as the default, or ask the user for the label to give to the merging of the two nodes.

Exercise 3:
Modify the algorithm in Exercise 1 to group the variables together and obtain a true Boolean circuit. The variable names must be removed to obtain a well-formed circuit. In addition to the circuit, return a list of variable names in the same order as the inputs: the ith input to the circuit must correspond to the ith variable in this list. For example, with the circuit above, we should have :



, [x0,x1,x2].

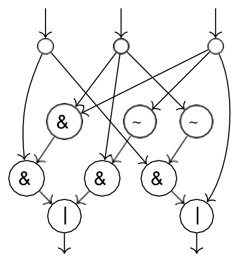
In a Boolean circuit, there can be several outputs, each corresponding to a Boolean formula.

Exercise 4:
Modify the algorithm again so that this time it takes *args: a sequence of strings of characters as parameters. Each string is supposed to be a propositional formula, as before. In the end, one should obtain a single boolean circuit that implements the sequence (i.e. each output corresponds to a formula).

Tip: we're not interested here in optimising the circuit, we can simply build the trees of the different formulas side to side and then do the operation that links the variables. For example, the algorithm ran on:

“((x0) & ((x1) & (x2))) | ((x1) & (~ (x2))) ", “ ((x0) & (~ (x1))) | (x2)"

can give :



You may notice that we haven't merged the consecutive associative gates (in the example above, we have two AND gates that we could merge). What happens if we run the algorithm on “((x0) & (x1) & (x2)) | ((x1) & (~ (x2))”?

Exercise 5 (Bonus):
Modify the algorithm one last time to deal with problems (if any) that arise when removing unnecessary brackets, as in the previous example.