NOR Logic Synthesis Problem

Linear Programming
Lab Report

Mirna Baksa `mirna.baksa@est.fib.upc.edu`

UPC - Facultat d'Informàtica de Barcelona — June 3, 2019

Problem Introduction

The problem at hand is to synthesize a logical circuit built up of NOR gates when given a truth table specifying the wanted behaviour of the circuit (i.e. NLSP - NOR Logic Synthesis Problem). The problem is simplified with the following assumptions:

- Only NOR gates with 2 inputs and 1 output can be used.
- The output of a NOR gate can only be the input of a single gate.
- In addition to the input signals, constant 0 signals can also be used as inputs of NOR gates.

An example is shown in figure 1 where the AND(x1,x2) function implemented as a NOR-circuit.

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

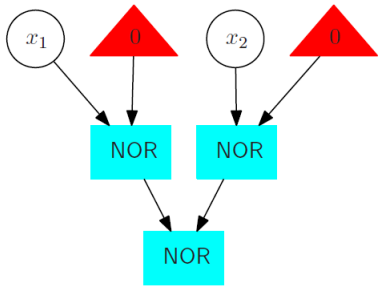


Figure 1: Truth table of y = AND(x1, x2) and NOR-circuit implementing it.

The objective is to find a NOR-circuit satisfying the specification that minimizes depth (maximum distance between any of the inputs and the output) or in case of tie in depth, size (number of NOR gates in the circuit).

The purpose of this project is to model and solve the NLSP with three problem solving techniques considered in the Combinatorial Problem Solving course - constraint programming, linear programming and propositional satisfiability.

The technology used in this report is linear programming. The LP solver used is **Cplex** [1].

1 Solution

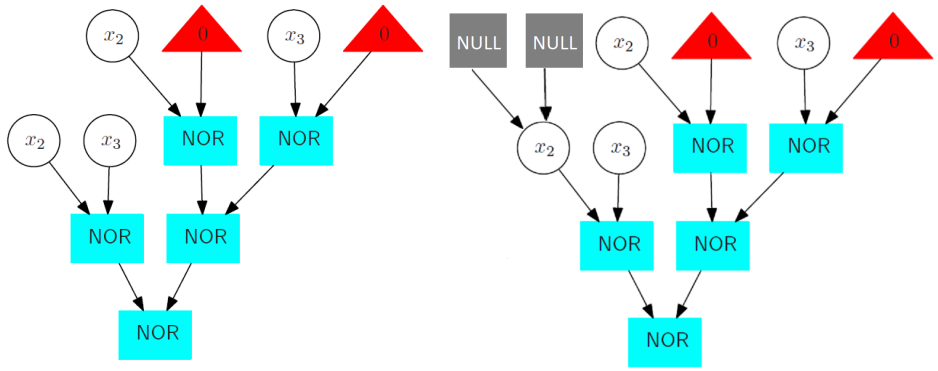
1.1 Model of the problem

The idea of the solution is to model the logical circuit as a binary tree. Looking back at figure 1, it is easy to imagine that the circuit in this figure is an "upside-down" (flipped by 180 degrees) binary tree with the output NOR gate being the root of the tree. By modelling the circuit as a binary tree we can easily perform many operations on it - the most useful one in the context of solving this problem is representing the tree

[1] https://www.ibm.com/analytics/cplex-optimizer

x_1	x_2	x_3	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(a) Truth table of y = AND(x1, x2) and NOR-circuit implementing it.



(b) Same circuit with added NULL nodes.

Figure 2: NOR - circuit example.

as a simple 1D array of integers. This will later allow us to easily apply constraints to get the solutions to our problem.

To define the rules of our binary tree, we have to define the types of nodes allowed in the tree and their representation. From the problem definition it is easy to see that we need three types of nodes - NOR gates, 0 signals and input signals.

A problem that evolves in this type of modelling is the structure of the binary tree - if the binary tree is not full (in a full binary tree all the nodes have two children except for the leaves), such as in figure 2a, we will have a much harder time representing the tree as an array and performing operations on it.

In the last lab [2], this was solved by introducing a new type of node - a NULL node, which was used to fill out all the gaps in the tree as can be seen in figure 2b (node x3 is still missing two NULL children nodes because the lack of space on the figure).

In this lab, according to the suggestion of the professor, the NULL node was removed and instead the gaps in the tree were filled with 0 signals. This will hopefully speed up and simplify the optimization.

To further define our circuit/tree structure, we must define the notation for each node. We will follow the following integer notation:

- **-1** - a NOR gate
- **0** - a 0 signal / NULL node
- ***i* = *1, ..., n*** - the i-th input signal, n being the number of input signals

The way the tree will be represented as an array is following: assuming that the tree is full, the children of the node ***i*** are at indexes $2 * i + 1$ (left child) and $2 * i + 2$ (right child).

One last problem we have to think about is the depth of the circuit. The depth will dictate the length of the array representing the binary tree which has to be fixed before running the solver. This was solved as follows in algorithm 1.

Algorithm 1: `Depth regulation`

```
depth ← 1
while no solution found do
    define the model;
    run search engine;
    if no solution found then
        depth ← depth + 1;
        continue;
    end
end
```

[2] https://github.com/mirnabaksa/Combinatorial-Problem-Solving/tree/master/Constraint%20Programming%20-%20Lab1

1.2 Variables

After the definition of the model itself, we are ready to define variables that will be used in the solving process.

A few regular (i.e. non Cplex) variables will be used:

- **n** - integer, number of input signals
- **num_types** - integer, number of node types, equal to n+2 (n input signals, 0 signal and NOR)
- **depth** - integer, current maximum depth of the binary tree
- **length** - integer, equal to $2^{depth} - 1$, the total number of nodes in the full binary tree of the given depth
- **truth_table** - a vector of booleans representing the truth table given as input
- **node_types** - array of integers, id-s of nodes - [-1, 0, ..., n]
- **M constants** - 4 constants used in the linear equations

To solve the problem, a few Cplex arrays will be used:

- **node_codes** - IloIntVarArray, an array representation of the circuit represented as a full binary tree
- **node_outputs** - IloBoolVarArray, an array representation of the node outputs
- **is_NOR** - IloBoolVarArray of size *length*, indicators for NOR gates: if the i-th node in *node_codes* is a NOR gate, the i-th member of the *is_NOR* array will be 1, 0 otherwise
- **output_indicators** - IloBoolVarArray, output indicators - explained in section 1.3

1.3 Model

After defining the variables, it is time to populate our model. I will not include the code snippets, but only the equations because I feel that the Cplex code is pretty clear once you have the equations.

First, we will define relationships between node types. The node_codes array variables are all in the interval [-1, n], whereas the is_NOR is in [0, 1].

$$-1 \leq node\_codes_i \leq n$$
$$0 \leq is\_NOR_i \leq 1$$

Next, we will make sure that the inputs of the NOR gate can only be other NOR gates, 0 signals or input signals (i.e. all gates), while the inputs of any other gate (0 signal, input signal, NULL) can only be null.

$$-1 + (1 - is\_NOR_i) \leq node\_codes_i \leq -1 + (n+1) * (1 - is\_NOR_i)$$
$$-is\_NOR_i \leq child \leq n * is\_NOR_i$$

If the node is NOR the equations turns into:

$$-1 \leq node\_codes_i \leq -1$$
$$-1 \leq child \leq n$$

If the node is not NOR the equations turns into:

$$0 \leq node\_codes_i \leq n$$
$$0 \leq child \leq 0$$

In these equations, the *child* represents the left or right child of the node that we are looking at, where the left is at index $2 * i + 1$ and the right is at index $2 * i + 2$. If any of these indexes are bigger than the node_codes array size, we forbid the node to be a NOR gate:

$$node\_codes_i \neq -1$$
$$is\_NOR_i = 0$$

What is left to do is make sure that the nodes follow the truth table assignment.

For this we use the node_outputs array. This is an array representation of a matrix of dimension (size of truth table x number of nodes in the tree (length variable)) - for each row of the truth table we will add statements for the node_outputs array so that the node outputs follow the truth table.

We will go trough each row of the matrix, indexed by i, and each node output, indexed by j and add following statements:

$$node\_outputs_{i,0} = truth\_table_i \quad \textbf{[1]}$$

$$is\_NOR_j + OI_{j,0} + OI_{j,1} + ... + OI_{j,n} = 1 \quad \textbf{[2]}$$

$$-1 * is\_NOR_j + 0 * OI_{j,0} + 1 * OI_{j,1} + ... + n * OI_{j,n} \leq node\_codes_j \leq \\ -1 * is\_NOR_j + 0 * OI_{j,0} + 1 * OI_{j,1} + ... + n * OI_{j,n} \quad \textbf{[3]}$$

$$nor\_output * is\_NOR_j + 0 * OI_{j,0} + x_1 * OI_{j,1} + ... + x_n * OI_{j,n} \leq node\_outputs_{i,j} \leq \\ nor\_output * is\_NOR_j + 0 * OI_{j,0} + x_1 * OI_{j,1} + ... + x_n * OI_{j,n} \quad \textbf{[4]}$$

Where nor_output is defined by:

$$x1 \leftarrow \text{output of left child (at } 2 * i + 1)$$
$$x2 \leftarrow \text{output of right child (at } 2 * i + 2)$$
$$or \leq x1 + x2$$
$$or \geq x1$$
$$or \geq x2$$
$$0 \leq or \leq 1$$
$$nor = 1 - or$$
$$0 \leq nor \leq 1$$

$$nor - 1 * (1 - is\_NOR_j) \leq node\_outputs_{i,j} \leq nor + 1 * (1 - is\_NOR_j)$$

Statement [1] assures that the output of the root node is the same as the truth table requires it to be. OI is a shortened name for output_indicators array we have mentioned in section 1.2. This array will be of dimensionality $length * num\_types$. Each row is an indicator for one node in the node_codes array and each column is an indicator for a type of node (i.e. $-1, 0, ...n$). The indicators can obviously have values from $0, 1$.

Statement [2] requires the sum of indicators in each row (denoted by i) to be 1 and since the indicators are from $0, 1$ this imposes that only one indicator at a time is 1 - so the indicator will tell us which node type is our node.

Statement [3] will give us a node type.

Statement [4] will give us the right output. The outputs of nodes $0, ..., n$ are known ahead (0 for zero signal, x_i for the i-th variable, while the nor output is calculated.

To understand better, an example: if the node is a x_1, statements [3] and [4] turn into:

$$1 \leq node\_codes_j \leq 1$$
$$x_1 \leq node\_outputs_j \leq x_1$$

since all the indicators besides the one for x_1 are 0.

In the end, we need an objective:

minimize sum(is_NOR)

i.e. minimize the size of the circuit (the number of NOR gates).

2 Usage

2.1 Compilation

I used Microsoft Visual Studio 2015. to build and run my Cplex program on Windows. Instructions for setting up Cplex can be found in the footnote [3].

[3] https://www.ibm.com/developerworks/community/forums/ajax/download/77777777-0000-0000-0000-000014819134/5b219cf2-1a5d-4a38-b630-7626bcbc0b5a/attachment_14819134_Using_IBM_ILOG_CPLEX_Optimizers_with_Mic.htm

2.2 Running

The program reads the instances from *stdin* and writes the solution to *stdout*.

The program was ran with the script shown in listing 1, assuming that we are positioned in the **src** folder, and the input files are in folder **../instances**. The outputs are written to the folder **../out**.

```
#!/usr/bin/env bash

for file in ../instances/*
do
 timeout 60s ./nor.exe < $file > ../out/$(basename $file .inp).out
done
```

Listing 1: Code running script

A script written to run the checker on all the output files is shown in listing 2.

```
#!/usr/bin/env bash

for file in ../out/*.out
do
        echo $file
        ./checker.exe < $file
done
```

Listing 2: Output checking script

3 Results

With the timeout of 60 seconds, **297/332** instances were successfuly solved in my machine and the results are in directory **out**. This is an improvement of 76 instances comparing to the constraint programming solution where 221 instances were solved within the time limit.

On all the output files, the **checker** script was ran to verify if the circuit satisfies the specifications given. Seeing that the checker does not check if the circuit is optimal, the results were additionally verified using the *results.txt* file given, using the script **checker_optimal**, which can be found in the **src** folder.