NOR Logic Synthesis Problem

Propositional Satisfiability
Lab Report

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

UPC - Facultat d'Informàtica de Barcelona — June 25, 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

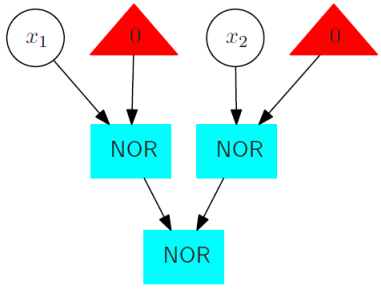


Figure 1: Truth table of y = AND(x1, x2) and NOR-circuit implementing it.

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 propositional satisfiability. The SAT solver used is **lingeling**.

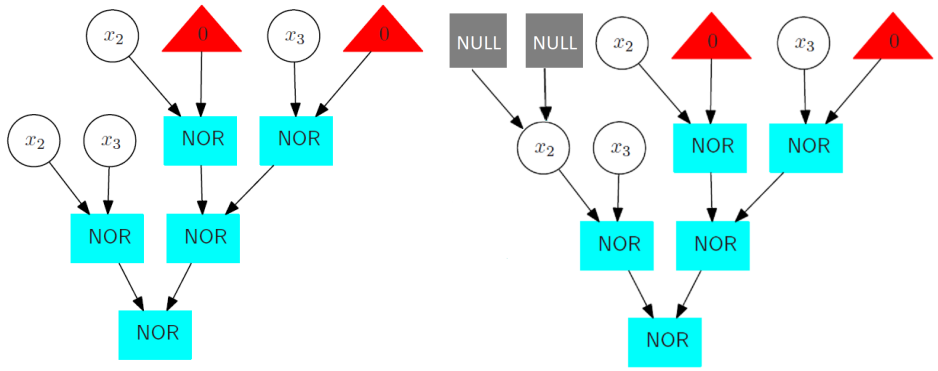
1 Solution

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

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.

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 [1], 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. In addition to the depth, we want to minimize the size of the circuit (number of NOR gates in the circuit).

These two problems were solved as follows in algorithm 1.

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

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
// Solution is found
while size > 0 do
    write size constraints;
    run search engine with added constraints;
    if no solution found then
        break;
    end
    size ← size − 1 ;
end
```

At first we run our search engine and if we do not get a solution we increase the depth and try again. When we do get a solution, we want to minimize the size. We write size constraints described in section 1.4 and run the search engine again. We decrease the current size and run the engine until no solution is found - and then we are sure that we have a minimal depth and size solution.

1.2 Variables

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

We will define some aliases to make the modelling easier:

```
#define V +
typedef string literal;
typedef string clause;
```

Next we define:

- **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]

To solve the problem, a few arrays will be used to implement our statements:

- **node_codes** - literal (string) array, an array representation of the circuit represented as a full binary tree
- **node_outputs** - literal(string) array, an array representation of the node outputs

1.3 Model

After defining the variables, it is time to populate our model.

In this section I will present statements that I used to solve the problem. The concrete implementation (dependent on lingeling syntax) can be found in the source repository or on GitHub [2].

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

Since SAT works with booleans, *node_codes* array has to be of size length * num_types. Each row will represent one variable, and each column will represent one node type. For example, if for some row *i* the first element is true, that will mean that that node is a NOR gate. If the second element is true, the node is a zero signal (or a NULL) and so on.

The constraints we have to impose are that exactly one variable can be true in each row (because one node can only be of one type). We do this by imposing the at-most-one and at-least-one constraints:

$$x_0 \vee x_1 \vee ... \vee x_n$$
$$\neg x_i \vee \neg x_j, \text{ for every i,j such that } i \neq j$$

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. Remember that the indicator for a NOR gate will be at index 0, for a zero signal at index 1 and so on.

$$\neg node\_type_{i,0} \vee node\_type_{child,0} \vee node\_type_{child,1} \vee node\_type_{child,2} ... \vee node\_type_{child,n+1}$$

$$\neg node\_type_{i,1} \vee node\_type_{child,1}$$
$$\neg node\_type_{i,2} \vee node\_type_{child,1}$$
$$...$$
$$\neg node\_type_{i,n} \vee node\_type_{child,1}$$

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:

$$\neg node\_type_{i,0}$$

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 through each row of the matrix, indexed by i, and each node output, indexed by j. For every type of node we impose:

If node type is NOR:

$$\neg node\_type_{j,type+1} \vee \neg node\_output_{i,left} \vee \neg node\_output_{i,j}$$
$$\neg node\_type_{j,type+1} \vee node\_output_{i,left} \vee node_output\_i, right \vee node\_output_{i,j}$$
$$\neg node\_type_{j,type+1} \vee \neg node\_output_{i,right} \vee \neg node\_output_{i,j}$$

If node type is 0:

$$\neg node\_type_{j,type+1} \vee \neg node\_output_{i,j}$$

If node type is an input signal:

If the input signal is true.

$$\neg node\_type_{j,type+1} \vee node\_output_{i,j}$$

If the input signal is false:

$$\neg node\_type_{j,type+1} \vee \neg node\_output_{i,j}$$

[2] https://github.com/mirnabaksa/Combinatorial-Problem-Solving

We also impose that the first element of the node_outputs array is equal to the truth table assignment:

If true:
$node\_output_{i,0}$

If false:
$node\_output_{i,0}$

1.4 Size constraints

The first iteration of limiting the size of the circuit (i.e. the number of NOR gates) used a naive encoding:

$$\text{For all } 1 \leq i_1 < i_2 < ... < i_k \leq n,$$
$$\neg x_{i,1} \vee \neg x_{i,2} \vee ... \vee \neg x_{i,k}$$

This is $\binom{n}{k}$ clauses and proved to be too slow!

To be able to limit the size of the circuit more efficient, I implemented a sorting network as described in slides for *Encodings into SAT* from the CPS course [3]. The implementation closely follows the structure proposed in the slides, so no further description will be given here.

2 Usage

2.1 Compilation

The code is compiled as a regular c++ file. In the code we run the system command:

```
tac tmp.rev | lingeling | grep -E -v \"^c\"
        | cut --delimiter=' ' --field=1 --complement > tmp.out
```

For this command to be ran correctly, the path to the lingeling solver has to be correct.

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] http://cs.upc.edu/ erodri/webpage/cps/cps.html

3 Results

With the timeout of 60 seconds, the naive approach (when imposing size constraints) solved *161/332* of all instances. Using the correct approach all instances (*332/332*) 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, and 35 instances comparing to the linear programming soltion where 297 instances were solved.

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.