NOR Logic Synthesis Problem

Constraint Programming
Lab Report

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

UPC - Facultat d'Informàtica de Barcelona — May 6, 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

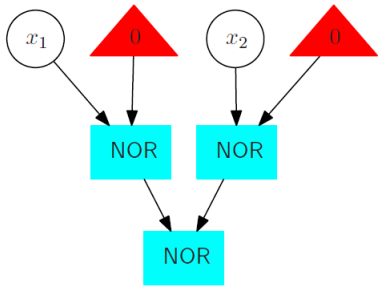


Figure 1: Truth table of y = AND(x1, x2) and NOR-circuit implementing it.

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 constraint programming. The CP solver used is **Gecode** [1].

1 Solution

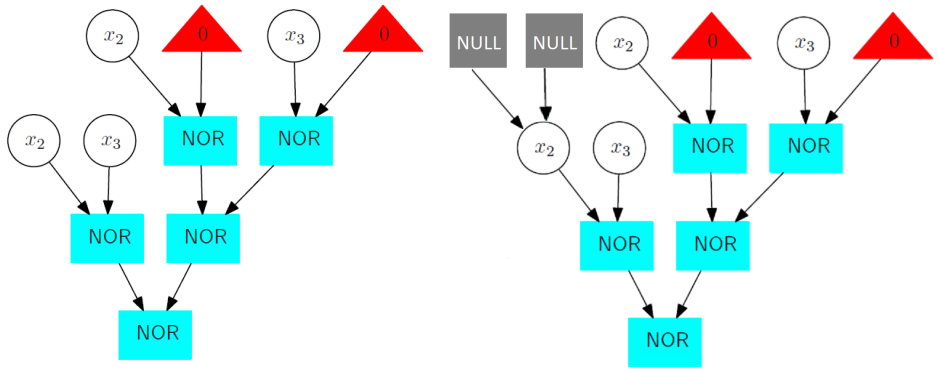
1.1 Model of the problem

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 constraint programming is representing the

[1] https://www.gecode.org/

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.

tree 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. With this in mind, we introduced a new type of node - a NULL node, which will fill out all the gaps in the tree, making it again simple to write to an array. The same circuit filled with NULL nodes can be seen in figure 2b (node x3 is still missing two NULL children nodes because the lack of space on the figure).

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

- **-2** - a NULL node
- **-1** - a NOR gate
- **0** - a 0 signal
- ***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
```

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 Gecode) variables will be used:

- **n_signals** - integer, number of input signals
- **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

To be able to impose constraints to corectly solve the problem, two Gecode arrays will be used:

- **node_codes** - IntVarArray, an array representation of the circuit represented as a full binary tree
- **node_outputs** - BoolVarArray, an array representation of the node outputs

1.3 Constraints

Finally, after defining the model and the variables we can define the constraints to solve our problem.

First, we impose constraints on the relationships between nodes. Going through each node, we want to define constraints that say:

$$node = NOR \implies left\_child \neq NULL \wedge right\_child \neq NULL.$$
$$node \neq NOR \implies left\_child = NULL \wedge right\_child = NULL.$$

This will ensure that the inputs of the NOR gate can only be other NOR gates, 0 signals or input signals, while the inputs of any other gate (0 signal, input signal, NULL) can only be null. So, for each node, the indexes of the left and right children are calculated and following constraints issued.

```
1  rel(*this, node_codes[i] == -1 && node_codes[left] != -2
2                  || node_codes[i] != -1 && node_codes[left] == -2);
3
4
5  rel(*this, node_codes[i] == -1 && node_codes[right] != -2
6                  || node_codes[i] != -1 && node_codes[right] == -2);
```

Listing 1: Constraints

The constraints for the left and right child are separated because beforehand we have to check if any of them are out of bounds of the array. If either of the child indexes are out of bounds of the array, we forbid that the current node is a NOR gate (as the NOR gate has to have inputs and can not be a leaf of the tree).

```
1 rel(*this, node_codes[i] != -1);
```

Listing 2: Forbid NOR

Above mentioned constraints ensure the proper relationship between the nodes of the circuit. The last thing that needs to be implemented is to ensure that the circuit implements the truth table given as input.

For this we use the **node_outputs** boolean 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 impose constraints on the **node_outputs** array so that the node outputs follow the truth table assignments.

The element at the *i-th* row and the *j-th* column is fetched as shown in listing 3.

```
1  BoolVar output_at(int i, int j) {
2       return node_outputs[i*length+j];
3   }
```

Listing 3: Fetching output at i,j

We will go trough each row of the matrix, indexed by *i*, and each node output, indexed by *j* and impose the constraints shown in listing 4.

```
1
2 // left <- index of left child, right <- index of right child
3
4 // NOR gates have NOR output
5 rel(*this, node_codes[j] != -1 ||
6 output_at(i, j) == (!(output_at(i,left) || output_at(i,right))));
7
8 // Zero signal has output 0
9 rel(*this, node_codes[j] != 0 || output_at(i,j) == 0);
10
11 //  Input signals follow the truth table
12 //  table_row is the i-th row of the truth table
13 for(int k = 0; k < n_signals ; k++){
14       rel(*this, node_codes[j] != k+1
15                  || output_at(i, j) == table_row[k]);
16 }
17
18 // NULL has no output
19 rel(*this, node_codes[j] != -2 || output_at(i, j) == NULL);
```

Listing 4: Node output constraints

1.4 Branching

After defining the variables and constraints, we branch out on both node_codes and node_outputs arrays.

```
1 branch(*this, node_codes, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
2 branch(*this, node_outputs, BOOL_VAR_NONE(), BOOL_VAL_MIN());
```

Listing 5: Branching policies

2 Usage

2.1 Compilation

Instructions for compiling and linking the code were found in the gecode manual [2], section 2.3.1. (Windows OS).

Assuming that we use Visual Studio Command Prompt, the two commands shown in 6 were used to compile and run the code (assuming the **CODE_ROOT** is set to the folder containing the source file **nlsp.cpp**). The backslash at the end of a line means that the line actually continues on the next line.

```
1 cl /DNDEBUG /EHsc /MD /0x /wd4355 -I"$GECODEDIR\\include" \
2 -c -Fo"$CODE_ROOT\\nor.obj" -Tp"$CODE_ROOT\\nlsp.cpp"
3
4 cl /DNDEBUG /EHsc /MD /0x /wd4355 -I"$GECODEDIR\\include" \
5 -Fe"$CODE_ROOT\\nlsp.exe" "$CODE_ROOT\\nlsp.obj" \
6  /link /LIBPATH:"$GECODEDIR\\lib"
```

Listing 6: Compiling and linking commands

2.2 Running

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

[2] https://www.gecode.org/doc-latest/MPG.pdf

The program was ran with the script shown in listing 7, 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 7: Code running script

A script written to run the checker on all the output files is shown in listing 8.

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

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

Listing 8: Output checking script

3 Results

With the timeout of 60 seconds, 221/332 instances were successfuly solved in my machine and the results are in directory **out**. By increasing the timeout to 400 seconds, 6 more instances were solved (outputs are in the directory **out_late**).

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.