Combinatorial Problem Solving
Course Project - Constraint Programming

Sergio Mosquera Dopico
Universitat Politècnica de Catalunya
`sergio.mosquera@est.fib.upc.edu`

Tuesday 7th May, 2019

1 Understanding the Problem

I am required to solve the NLSP or *NOR Logical Synthesis Problem* using Constraint Programming. The idea of the problem is that, given an input with n lines, such that the first line indicates the number of inputs available and lines 2 to n represent a truth table result, we need to develop a program that creates a logical circuit only using NOR gates that outputs the same truth table as the one received in the input.

To achieve this task I will use the framework *Gecode*, which is an open source C++ toolkit for developing constraint-based systems and applications. The final representation of the system corresponds to a binary tree where the first element is the root that can be either an input, a zero value or a NOR gate. If the element is a NOR gate, it would have two leaves as input values, that can be any of the previous elements. The second goal of the project is to find a valid system with the smallest possible *depth* and in case of tie, get the one with the smallest *size*. Depth and size are important parameters in this problem, and represent the highest number of NOR gates to be traversed through the generated tree to go from the root to a leaf, and the number of NOR gates in the circuit, respectively.

The solution obtained from the program execution must follow the format specified by the problem statement. A graphical representation of a finished circuit would be the following:

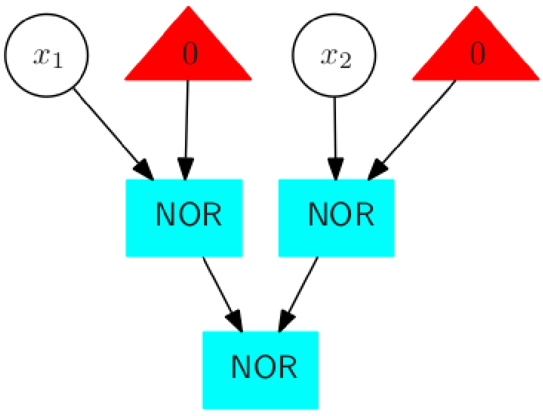

Figure 1: Tree representation of the solution

2 Problem Formalization

This project begins with a deep reading of the statement of the problem. Once I have a clear idea of the it, I started with its formalization. This is the first step, prior to the development of the final model for the problem. First of all, I need to define the variables used for this project along with their domain:

Variable	Definition
I	Array of Integer Variables.
VB	Vector of Boolean values.
num_inputs ($\mathbb{N}$)	Maximum number of input elements that can be used.
depth ($\mathbb{N}$)	Maximum number of NOR gates to be traversed from the root to a leaf.
max_nodes ($\mathbb{N}$)	Maximum number of nodes that the system will be able to have. This variable is used to relax the constraint of the depth. Initially, we specify a maximum depth that the system can achieve so that max_nodes correspond to $2^{depth+1} - 1$.
max_nors ($\mathbb{N}$)	Maximum number of nodes that the system will be able to have. This variable is used to relax the constraint of the depth. Initially we specify a maximum depth that the system can achieve so that max_nodes correspond to $2^{depth+1} - 1$.
original ($\mathbb{B}$) $b \in B$	The original truth table the program is searching a NLS circuit for.
result ($\mathbb{N}$) $i \in I$ (≥ -1)	Solution representation, each element of the array is an Integer Variable i Each element of the tree is represented by 4 values, being the first one the index of the element in the tree, the second one a code to indicate whether the element is a NOR gate (-1) or an input (> 0), and the third and forth the index of the inputs of the NOR gate (in case it is a NOR gate) or both zero.
inputs ($\mathbb{B}$) $b \in B$	Representation of the truth values that each of the inputs would have in a truth table. The length of the vector is $(num\_inputs + 1) \times 2^{num\_inputs}$. This vector also holds the truth values for a *zero input*, that is why we need to specify $num\_inputs + 1$

Table 1: Variable Definition

The above variables are the ones used to model the problem, that is done by means of constraints. Gecode itself provides several types of constraints (linear, relation, distinct, ...), but for us is enough using only relation and count constraints. The different constraints are presented in the next section along with an explanation for each of them.

3 Constraint Programming Model

Once we have the variables being used for modelling the program constraints, we can start thinking about the model. There is a distinction in the constraints that can be made, depending on the type of the variable (NOR gate or input) we are imposing a constraint on. Also there are more global constraints that I will define first, as they are the most simple and easier to identify. The last part of this section will consist in the main constraint of the problem, the one that ensures that the NOR operation performed over the system returns the same solution as the original one.

3.1 General Constraints

As each element for the tree is represented by a combination of four variables, being the first one the index of the element, it makes sense that the first element (the root), the one that will be responsible of returning the output of the execution, equals 1:

$$result_{[0]} = 1 \tag{1}$$

As I have explained in table 1 we are relaxing the depth constraint such that there is an initial maximum depth. This relaxation means that we have to include a new constraint to impose that the solutions can have at most $2^{depth} - 1$ NOR gates:

$$\#\{i \in \{0,..,|result|-1\} | result_i = -1\} \leq 2^{depth} - 1 \tag{2}$$

This constraint only applies when the size of the array is greater than one (there is at least one NOR gate in the circuit). As the definition of a NOR gate in our system requires two inputs, it makes sense that these inputs are also going to be defined in the final solution. This means that the index identifying every element used as an input in a NOR gate must appear just twice in the whole system:

$$\forall e \in result \quad \#\{i \in \{0,..,|result|-1\} | result_i = e\} = 2 \tag{3}$$

We need to impose the constraint that the tree has to be well-formed, that is that every element is made up of the four corresponding elements. This constraint is imposed ensuring that the size of the array is divisible by 4.

$$|result| \% 4 = 0 \tag{4}$$

As the representation of the system is a binary tree, we can easily realize that the number of nodes in the tree will be odd (even in the case there is just a root element):

$$(|result|/4) \% 2 = 0 \tag{5}$$

The index of every element needs to be positive. The index of the element can be identified as the first element of the four-tuple identifying a node, that is all the elements such that their index is divisible by 4.

$$\forall i \in |result| \quad result_i > 0 \quad s.t. \{i \% 4 = 0\} \tag{6}$$

Also I had to impose a constraint for the representation of the code identifying the type of the node in the tree, which needs to be greater than or equal -1.

$$\forall i \in |result| \quad result_i \geq -1 \quad s.t. \{(i-1) \% 4 = 0\} \tag{7}$$

3.2 Inputs constraints

The inputs are the two last elements of the four-tuple that identify a node. If the element identifier is different than -1, then it is an input, and that means that it is a leaf node (has no inputs) so the last two elements of this tuple need to be both 0.

$$\forall i \in |result| \quad result_i = 0 \quad s.t.\{(i-1)\ != -1\} \tag{8}$$

3.3 NOR gates constraints

The NOR gates receive two inputs to perform the NOR operation, this inputs are identified by their index (first element of the four-tuple) so if we find that the current element is a NOR gate, the next two elements need to be greater than 1, because index 1 corresponds to the root node and cannot be reused as an input.

$$\forall i \in |result| \quad result_{i+1} > 0 \text{ and } result_{i+2} > 0 \quad s.t.\{(result_i = -1\} \tag{9}$$

The previous constraint is not complete, it can be bounded a little more. As the system will be having at most *max_nodes*, it makes sense that the index of every input to any node cannot be greater than that value.

$$\forall i \in |result| \quad result_{i+1} < max\_nodes \text{ and } result_{i+2} < max\_nodes \quad s.t.\{(result_i = -1\} \tag{10}$$

I stated in the problem definition that a NOR gate can have at most two inputs, but I forgot to say that the same input cannot be reused for two different NOR gates. So it is necessary to ensure that every two last elements of the four-tuple are different. This can be done taking into account the position in the array and ensuring that that value only appears once in that position of the four-tuple (for every four-tuple).

$$\forall e \in result \quad \#\{i \in \{0,..,|result|-1\} | (i-1)\%4 = 0 \rightarrow ((result_{i+2}! = result_{i+1}) \text{ and}$$

$$(result_{i+1} = e \text{ or } result_{i+2} = e))\} = 1$$

As it was mentioned in a previous constraint, the index of the inputs of a NOR will appear exactly twice in the program, one for be the input and another one to be defined. As the definition of the input happens after their declaration as input in another gate, it makes sense that their index must be greater than the index of the NOR gate they are serving as input.

$$\forall i \in [0, |result|-1] \quad result_{i+1} > result_{i-1} \text{ and } result_{i+2} > result_{i-1} \quad s.t.\{(result_i = -1\} \tag{11}$$

This constraint gives more information about the previous one, that informed us about the order of the inputs. Our program follows a preorder strategy to traverse the tree when creating it, that is, if we think about the tree of image 1, we are traversing first the left branch of the tree and, when it is fully traversed, the next step would be starting with the right side, traversing again, first of all, the left-hand side of each subtree. What this constraint tries to impose is that the left input of a NOR gate has to be the next element defined in the tree.

$$\forall i \in [0, |result|-1] \quad result_{i+1} = result_{i+3} \quad s.t.\{i = -1\} \tag{12}$$

3.4 Main constraint

The main goal of the project was to find an equivalent NLS circuit for the truth table passed as an input. With the constraints imposed in the previous section, the condition is not ensured yet, so that is why this is considered the main constraint of the problem.

As the predicted behavior is quite complex I will explain the algorithm that helps to fulfill the constraint as a pseudocode. The idea of this algorithm is that it will traverse the tree from the root to the leaves until it gets the result outputted by performing the needed NOR operations. This result will be the final truth table of the tree, and once we get this truth table, we can state the constraint such that, the resulting truth table has to be exactly the same as the truth table we received as a parameter. It will be more clear with pseudocode:

Algorithm 1 Nor algorithm

```
 1: function norOperation(list, index)
 2:     $left\_index = findIndex(list, index + 2)$
 3:     $right\_index = findIndex(list, index + 3)$
 4:
 5:     initialize left and right as empty lists
 6:
 7:     // We traverse the left-hand side if the tree
 8:     if ($list_{[left\_index+1]}$ != − 1) then
 9:         $left = getInputs(left\_index + 1)$
10:     else
11:         $left = norOperation(list, left\_index)$
12:
13:     // Now the same for the right-hand side
14:     if ($list_{[right\_index+1]}$! = −1) then
15:         $right = getInputs(right\_index + 1)$
16:     else
17:         $left = norOperation(list, right\_index)$
18:     Return $norResult(left, right)$
19:
20: /*
21: * Returns the index in the list were the input is defined
22: */
23: function findIndex(list, index)
24:     for each index $i$ in $|list|$ do
25:         if $(((i - 1)\%4 = 0)$ and $(list_{[i]} = index))$ then
26:             Return $i$
27:
28: /*
29: * Returns a list holding the result of the NOR operation for two lists
30: */
31: function norResult(left, right)
32:     for each index $i$ in $|left|$ do
33:         $result\_nor_{[i]} = ($ not $(left_{[i]}$ or $right_{[i]}))$
34:     Return $result\_nor$
```

Figure 2: Pseudocode for Nor algorithm

The first algorithm being executed from the previous, is the one called *norOperation* with *result* and $index = 0$ as parameters. In this way, we start the execution from the root node and we traverse the whole tree by recursive calls, receiving the truth table values for each of the subtrees. Once the whole tree has been traversed, we can perform the last NOR operation between the left input and the right input of the root node, this value is the output of our algorithm (let's call it *solution*). So, we can impose a new constraint as follows:

$$\forall i \in [0, |solution| - 1] \quad result_{[i]} = solution_{[i]} \tag{13}$$

4 Issues of the project

Although the program can be compiled without errors and the constraints are defined, I was not able to find a suitable way to implement the main constraint. The problem is that, to impose some constraints, I had to know before-hand the current value of a given variable, e.g. to check whether the element is a NOR gate or an input, and the variables are not assigned initially. I was not able to find out how to ensure the value a variable must have in case other variable fulfills some conditions.

Even though, the code is sent as requested in the deliverable so the error can be checked, which is indeed the value assignment missing, by removing the constraints that contain an invocation to the method $val()$.