TP3

The course homepage is here :

`http://www.lsv.ens-cachan.fr/~schwoon/enseignement/systemes/ws1415/`.

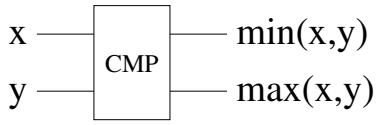
You will find the slides from the course and some other files for the exercise there.

1 Verification of Circuits

A serious problem in hardware design is to ensure that a circuit fulfils its functionality under all circumstances. We shall discuss such a method (for static circuits), using *sorting networks* as example.

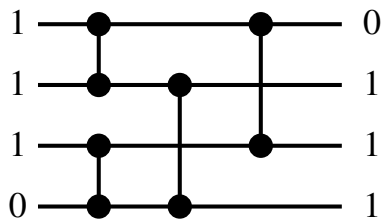
The purpose of a sorting network is to sort n numbers in ascending order. For a given network, n is fixed. It is known that a sorting network works correctly iff it works correctly for all sequences consisting of 0 and 1 (zero-one principle), so we shall assume that all inputs are 0 or 1.

The basic element of a sorting network is a *comparator*. It takes two values and yields two outputs, the upper line being the smaller and the lower line the bigger of the two values.



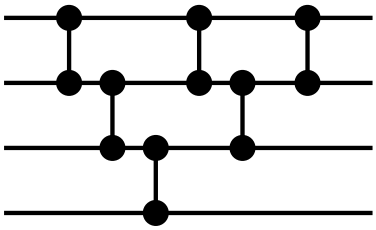
(a) Give a circuit realising the function of the comparator.

A sorting network consists of n wires, for a fixed n, connected by comparators. For convenience, we shall draw connectors simply as vertical wires. Below is a sorting network for $n = 4$ and an example of its input and output.



(b) This network gives the correct result for the example input, but not for all inputs. Find an input for which it gives an incorrectly sorted sequence.

One possibility to easily design a correct network is by taking inspiration from classical, sequential sorting algorithms. For instance, the network below imitates Bubble Sort : The first series of comparisons finds the biggest element, the next series the second biggest etc.



The course hompeage provides a Sorting Network Simulator (`snsim.pl`). Download it and unzip the package. Before running the program, you need to make it *executable*. In the shell, `cd` to the directory where you installed the files, then run `chmod +x *pl`.

The simulator takes as input a file listing the pairs of wires, to be compared. For instance, the network above is given as follows (see the file `bubble-4.sn`).

```
1 2
2 3
3 4
1 2
2 3
1 2
```

You can now simulate this network in the shell by running

```
./snsim.pl bubble-4.sn 0101
```

where the latter is the input sequence.

However, this sorting network is inefficient. As usual, there are two measures for the efficiency of a network : the *size*, i.e. the number of comparators, and the *depth*. The depth has the usual meaning for logical circuits, it can equivalently be defined on wires :

— A wire has depth 0 if it is connected to an input value.

— If the wires entering a comparator have depth d_1 and d_2, then its outgoing wires have depth $1 + max(d_1, d_2)$; we also say that the comparator itself has this depth.

The depth of a network then is the largest depth of any of its wires. For instance, the "bubble sort" network has size 6 and depth 5. The simulator provides you with this data for a given network.

(c) Design a network for $n = 4$ that is both correct and more efficient (with respect to both measures) than the one shown above. Enter its syntax into the network simulator and use it for testing.

(d) Similarly, find networks for $n = 5$ and $n = 6$, trying to minimize size and depth.

How do we verify that a given network correctly sorts, whatever its input ? One possibility is to try for every possible input, of which there are 2^n, either manually or by letting the computer do it. Another possibility is to formulate the question in a suitable logic, in this case propositional logic. Our goal is to transform a network N into a formula ϕ_N such that ϕ_N has a satisfying assignment iff N is incorrect for some sequence. Searching for a satisfying

assignment can be done with so-called *SAT solvers*. (Note that while SAT solving is no more efficient than enumerating 2^n potential solutions in the worst case, it is often much more efficient in practice.)

The variables of our formula ϕ_N will be of the form x_i^d, which takes the value of a wire on the i-th line with depth d. The variables x_i^0, for $i = 1, \ldots, n$, are the inputs.

(e) Let C be a comparator at depth d whose inputs are at depth d_1 and d_2. Using the result from (a), give a formula ϕ_C relating the variables representing the inputs and outputs of C.

Let $\mathcal{C}$ be the set of comparators and d_i the final depth of the i-th input. Then ϕ_N can be formulated as follows :

$$\phi_N := \left(\bigwedge_{C \in \mathcal{C}} \phi_C \right) \wedge \neg \phi^{sort}(x_1^{d_1}, \ldots, x_n^{d_n})$$

where ϕ^{sort} expresses that the final output is sorted.

(f) Find the formula ϕ^{sort}.

On the course homepage, you find a program that assembles ϕ_N, given a network N. It then feeds ϕ_N into the SAT solver Z3 and tells you whether there is an incorrect input. All that is missing are ϕ_C and ϕ_s.

(g) Complete `snverif.pl` with the missing formulae. Run the completed program on your networks to check whether they are correct. Again, using the Bubble Sort network as example, the syntax is `./snverif.pl bubble.sn 4`, where 4 is the value of n.

2 Synthesis of circuits

A more far-reaching application than verification is to automatically synthesize a circuit with desired properties. Instead of asking whether, given a fixed circuit, there exists an input violating its desired properties, we would ask, given desired properties, whether there exists a circuit satisfying them for all its inputs.

We shall discuss this idea, again using Sorting Networks as example. The discussion is based on the paper *Synthesis of Parallel Sorting Networks using SAT Solvers* by Morgenstern and Schneider, 2011.

The basic idea is that a sorting network of depth d can be expressed as a sequence of *permutation vectors* $P_1, \ldots, P_d$, where P_i specifies the behaviour of all comparators of depth i. More precisely, we suppose that $P_{k,i} = j$ and $P_{k,j} = i$ if a comparator with depth k is connected to wires i and j. Moreover, $P_{k,i} = i$ if the i-th wire is not connected to any comparator at depth k. The example below gives the five permutation vectors for the Bubble Sort example :

$P_1 = (2\ 1\ 3\ 4), \quad P_2 = (1\ 3\ 2\ 4), \quad P_3 = (2\ 1\ 4\ 3), \quad P_4 = (1\ 3\ 2\ 4), \quad P_5 = (2\ 1\ 3\ 4)$

Given n and d, we can now construct a formula $\phi_{n,d}^{synth}$ that asks for the existence of a sorting network for n values with depth d. Its variables are the integer-valued elements of P_k and the boolean values x_i^k ($i = 1, \ldots, n$ and $k = 0, \ldots, d$). This formula has three components that ensure

(i) that P_k is indeed a permutation vector, for all k (ϕ^{perm});

(ii) that the values x_i^k are consistent with P_k, for all inputs (ϕ^{cons});

(iii) that the final result is sorted, for all inputs (ϕ^{sort}).

$$\phi_{n,d}^{synth} := \left(\bigwedge_{k=1}^{d} \phi^{perm}(P_k) \right) \wedge \left(\bigwedge_{x_1^0,\ldots,x_n^0 \in \{0,1\}} \left(\bigwedge_{k=1}^{d} \phi^{cons}(P_k, x_{k-1}, x_k) \right) \wedge \phi^{sort}(x_d) \right)$$

As for the first, ϕ^{perm} can exploit that the permutations are all symmetric :

$$\phi^{perm}(P_k) := \bigwedge_{i=1}^{n} P_{k,P_{k,i}} = i$$

ϕ^{cons} adapts our formula ϕ_C from exercise 1(e) :

$$\phi^{cons}(P_k, x_{k-1}, x_k) := \bigwedge_{i=1}^{n} x_i^k = \begin{cases} x_i^{k-1} \vee x_{P_{k,i}}^{k-1} & \text{if } i \leq P_{k,i} \\ x_i^{k-1} \wedge x_{P_{k,i}}^{k-1} & \text{otherwise} \end{cases}$$

ϕ^{sort} finally is like in exercise 1(f).

(a) The program `snfind.pl` generates $\phi_{n,d}^{synth}$ automatically. Use it to find optimal-depth networks for $n = 5, 6, 7, \ldots$

This principle can also be applied to other functions than sorting. Suppose that we want to build some functionality of n inputs only from NAND gates. In a first step, we can still encode the connectivity by symmetric permutation vectors, i.e. for a NAND-gate at depth k with inputs from wires i and j, we have $P_{k,i} = j$ and $P_{k,j} = i$. The only difference is that this time the output goes to the wire with the smaller index, and the wire with the higher index becomes useless.

(b) Adapt ϕ^{cons} to NAND-gates instead of comparators.

(c) Supposing that $n = 2$ and the desired functionality is XOR, find a suitable replacement for ϕ^{sort}. What about other functionalities that you can think of?

(d) Does this scheme always find the circuit with the smallest depth? If not, state why and propose an improvement.

(e) (advanced exercise) Adapt `snfind.pl` to find the smallest circuit for XOR or other functions.