Programming Languages

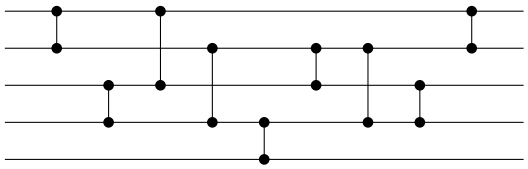
Project 1 (deadline: 27.Oct.2015)

1 Introduction

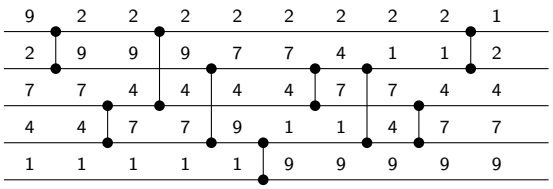
Sorting networks are an example of data-oblivous sorting algorithms, where the sequence of instructions to be executed is independent of the data to be sorted. This is different from the traditional sorting algorithms: for example, the behavior of Quicksort is heavily dependent on the particular elements that are used as pivots. For this reason, sorting networks are suitable for implementation as hardware circuits (as long as the number of inputs to be sorted does not change), or even in software as base cases of more powerful sorting methods.

A sorting network is formed by n parallel channels, where n is the number of elements to sort, and k special gates called *comparators*. The channels contain the input data; each comparator is connected to two channels, and behaves as follows: it reads the data on those two channels, and outputs the same two values sorted. In a more abstract formulation, we can view the channels as the numbers $1, \ldots, n$ and comparators as pairs $\langle i, j \rangle$. Given a sequence $\bar{x}$, the comparator $\langle i, j \rangle$ acts on x by comparing x_i with x_j and interchanging them if $x_j > x_i$. A sorting network is a list of such pairs, and it acts on input vectors sequentially, each comparator's output being the input to the next one.

Example. The following is a sorting network on 5 channels.

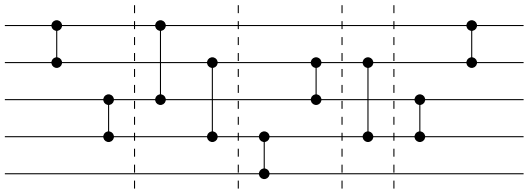


Labeling the channels from 1 to 5, top to bottom, the comparators in the network are $(1, 2)$, $(3, 4)$, $(1, 3)$, $(2, 4)$, $(4, 5)$, $(2, 3)$, $(2, 4)$, $(3, 4)$ and $(1, 2)$. The next figure shows how an example input propagates through the network.



Sorting networks are intrinsically parallel algorithms, as two consecutive comparators that act on disjoint pairs of channels can be executed simultaneously. For this reason, there are two measures of complexity of a sorting network: its *size*, defined as the number of comparators it contains (or its length, if one thinks of it as a list); and its *depth*, defined as the number of execution steps if an arbitrary number of comparators is allowed to run in parallel. It is customary to call a group of comparators executing simultaneously in a sorting network a *layer*.

Example. The previously shown sorting network can actually execute in only five computation steps – five layers –, separated by the dashed lines in the next picture.



The layers in this sorting network are: $\{(1,2),(3,4)\}$, $\{(1,3),(2,4)\}$, $\{(4,5),(2,3)\}$, $\{(2,4)\}$, and $\{(3,4),(1,2)\}$.

2 The problem

The goal of this project is to develop ProLog code to manipulate sorting networks. We are interested both in *checking* that something is a sorting network, and in *finding* sorting networks with special properties. In order to do this, we need the notion of *comparator network* – a sequence of comparators, not necessarily sorting its input.

2.1 Data and its representation

You will need to manipulate comparators (pairs of natural numbers), comparator networks (lists of comparators) and binary sequences of a fixed length. Choose the representation you will use for these, justifying your options.

All the predicates below should be guaranteed to evaluate to true or false (without errors) when all arguments are instantiated, unless otherwise specified.

1. Define a predicate is_network/1 that holds precisely when its argument is a comparator network according to your representation.
2. Define a predicate channels/2 such that channels(C,N) holds when C is a comparator network (in your representation) on N channels.

   This predicate should also be invokable with a variable as first argument, iteratively generating all comparator networks on the given number of channels.
3. Define a predicate run/3 such that run(C,Input,Output) holds precisely when executing C on Input returns Output. You may assume that the arguments to run are well formed.

   This predicate should also be invokable with a variable as third argument, computing the result of executing the network on the given input.
4. Define a predicate is_SN/2 such that is_SN(S,N) holds when S is a sorting network on N channels.
5. Define a predicate find_SN/3 such that find_SN(N,K,S) will return all sorting networks on N channels with length K. It should also be possible to use a variable as second argument.

2.2 Standard comparator networks and untangling

In general, comparators can choose how to sort their inputs (ascending or descending). However, it is usual to assume that the minimum is placed on top, as in the examples above. If all comparators satisfy this property, the network is said to be *standard*.

Any comparator network S can be transformed into a standard one by the following procedure: let (i,j) be the first comparator in S where $j > i$; replace it with (i,j), and interchange i and j in all comparators after it. Repeat until the network is standard.

Standardization naturally leads to a notion of behavioural equivalence. Two standard comparator networks C_1 and C_2, both on the same number n of channels, are *equivalent* if there is a permutation π of $1..n$ such that C_2 can be obtained by renaming the channels in C_1 according to π and standardizing the result.

6. Define a predicate is_standard/1 that holds when its argument is a standard comparator network.
7. Define a predicate standardize/2 such that standardize(C,S) holds when S is the network obtained from C by the standardization procedure described above.

   This predicate should work when C and S are both instantiated, or when only C is instantiated.
8. Define a predicate equivalent/3 such that equivalent(N,C1,C2) holds if C1 and C2 are equivalent comparator networks on N channels.

   Leaving C2 uninstantiated, this predicate should return all comparator networks equivalent to C1.

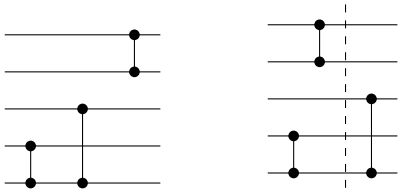
2.3 Parallelism and depth

We now turn our attention to the problem of parallelism, and consider comparator networks where the layers are made explicit.

9. Define a suitable representation for layered comparator networks.

10. Define a predicate layered/2 such that layered(C,L) holds if L is a layered version of C.

    This predicate should work when C and L are *both* instantiated, and it should take into account that the comparators in C and L do not necessarily appear in the same order: in the picture below, the network on the right is a correct layered version of the network on the left.



11. Define a predicate network_to_layered/2 such that: when its first argument is instantiated with a comparator network C, the answer substitution for its second argument is an optimal layering of C.

12. Define a predicate layered_to_network/2 such that: when its first argument is instantiated with a layered comparator network C, the answer substitution for its second argument is the comparator network corresponding to C.

13. Can you implement a single predicate combining the functionalities of layered/2, network_to_layered/2 and layered_to_network/2?

2.4 Useful ProLog predicates

The following predicates may help you defining some of your predicates. Check the interactive help for their usage.

- integer/1 tests whether its argument is an integer number
- between/3 tests whether a number is within some range
- nth1/3 and nth1/4 return the nth element from a list, counting from 1
- append/2 holds when its first argument is a list of lists and its second argument is the concatenation of all those lists

3 Report & evaluation

You should hand in a succint report describing all implementation options, including (but not restricted to) the following: auxiliary predicates that you chose to define, specific ProLog mechanisms that were used, and a justification of the cuts employed. The report should also include a full listing of the program with appropriate documentation, as well as a section with example executions.

The evaluation will take into account the following aspects, in order of importance:

- Report: clearness, completeness and quality.
- Program design: code quality and correct employment of specific ProLog mechanisms (cut, backtracking and unification).
- Program soundness: execution, correctness and efficiency.