1 Generate and Prune implementation

In this section we explain how to implement the generate and prune methods of [?] and some optimizations that can be applied to these algorithms and the subsumption tests.

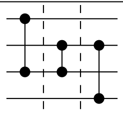
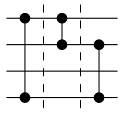
1.1 Representation of comparator networks

To represent a comparator network we just need an array of comparators. However, as we explained before for the subsumption operation we need to apply permutations to the outputs. So, we choose to represent the comparator network together with its set of outputs for efficiency. This allows to speed also the generation of this outputs when adding a new comparator as we will only need to apply the new comparator to the binary sequences of the previous outputs. Finally, in order to apply some optimizations that we will explain in the next subsections we also need to store 2 more matrices and 1 array, of sizes $n \times n$ and n respectively. For efficiency, the matrices are stored as a 1 dimension array and we use bit operations to access the matrix positions.

The first 2 matrices store the positions where any output contains a 0 or a 1 and we call them W^0 and W^1, the row number defines partitions of outputs with a given number of 0s or 1s, if a position (i, j) contains a 1 it means that the comparator network outputs contain a 0 or a 1 in that position, if it contains a 0 it means that the outputs don't contain a 0 or a 1 in that position.

The last array contains the number of sequences with k 1s in the outputs in positions $k = 0, 1...n$ and we call it S^1. In Table [?] we can see an example of how the outputs are generated and partitioned for two 4-input comparator networks.

Table 1 Comparator network outputs partitioned by number of 1s

number of 1s	**1**	**2**	**3**
 C_1	0010 1000	0110 1010 1100	0111 1110
 C_2	0010 0100 1000	0101 0110 1010 1100	0111 1101 1110

To improve the memory usage as much as possible and make it more computationally efficient, the outputs array is implemented as array of 2^n bits where a position is set to 1 if the output is set and 0 otherwise. This is the most efficient representation for the set of all output. It also allows to generate the outputs quickly by the use of bitwise operations that are way faster in modern programming languages. This also applies to the W^0 and W^1 matrices where each row is encoded in one array position.

1.2 Generate and Prune implementation

In this section we will describe the algorithms Generate and Prune and some optimizations, the algorithms are same than the ones in [**?**]. They are very simple but run in massive datasets that make their implementation details crucial to be able to handle the problem for inputs sizes in $n \geq 8$.

1.2.1 Generate

The generate algorithm takes the set N_n^k containing the comparator networks with k comparators and extends it to R_n^{k+1} by adding a comparator in every possible positions for each element on N_n^k.

Given the comparator network C and the comparator (i, j) we can extend this network to $C' = C \bigcup (i, j)$. We apply the optimization seen in [**?**] to avoid adding redundant comparators which would led to networks that order the same than its parent network. These networks would be eliminated either way by the prune algorithm, but removing them in this stage is cheaper in computation time. We say that the comparator (i, j) is redundant if for all the elements in the sequence $x_1...x_n \in outputs(C)$ $x_i \leq x_j$.

Algorithm 1 Generate

```
$N_n^{k+1} \leftarrow \emptyset$
$R_n^k \leftarrow$ complete set of filters for k comparators
$C \leftarrow$ comparators array
**for** $n \leftarrow 1...N$ **do**
    **for** $c$ in $C$ **do**
        $n' \leftarrow n \bigcup c$
        **if** $n'$ is not redundant **then**
            $N_n^{k+1} \leftarrow N_n^{k+1} \bigcup n'$
        **end if**
    **end for**
**end for**
      **return** $R_n^{k+1}$
```

1.2.2 Prune

The prune method takes the set of networks N_n^k and reduces it to a complete set where comparator networks subsumed by any other are removed. This generates the set R_n^k. The algorithm travels the array of comparator networks and adds to the result array those that are not subsumed by any other. As subsume is not a symmetric operation it also crosschecks and removes networks in the result array that are subsumed by the actual network. The following algorithm shows the implementation of the Prune algorithm.

Algorithm 2 Prune

```
$N_n^k \leftarrow$ output of Generate
$R_n^k \leftarrow \emptyset$
$subsumed \leftarrow False$
for $n$ in $N$ do
    for $r$ in $R$ do
        if $r$ subsumes $n$ then
            $subsumed \leftarrow True$
            break
        end if
        if $n$ subsumes $r$ then
            $R \leftarrow R \setminus r$
        end if
    end for
    if not $subsumed$ then
        $R \leftarrow R \bigcup n$
    end if
end for
    return $R$
```

1.2.3 Parallel implementation

To be able to reduce the running time and make use of all the processors in the test machine, we implement a parallel version of the algorithms as in [**?**]. The parallelization of Generate is straightforward as every comparator network can be extended independently. So, we simply divide the work to be performed by all the processors. However, the Prune algorithm requires more work as the networks should be crosschecked through the whole array.

To implement the parallel Prune we make use of another algorithm that is slight modification of the Prune algorithm, we call it Remove and it works in the same way than Prune but checks the subsumption just in one direction. We start by dividing the set to be pruned in as many clusters as processors, we apply prune algorithm to each of this clusters by separate and as a result, no network is subsumed by others in the same cluster. After that we need to still remove networks that are subsumed by others in the remaining clusters. So, for each cluster we apply the Remove method in parallel with the other clusters. If we have n clusters this results in n Prune operations and $n-1$ Remove operations per processor, resulting in around n^2 operations per processing unit. The Parallel Prune and Remove algorithms can be found below.

Algorithm 3 Parallel Prune

```
$N \leftarrow networks$
$C \leftarrow Divide(N)$                    ▷ Divide N in as many Clusters as processor
Each processor performs:
    Prune($C_i$)
    **for** $c$ in $C$ **do**
        Remove($c, C \setminus c$)
    **end for**
    **return** $N$
```

Algorithm 4 Remove

```
$result \leftarrow \emptyset$
$N_i \leftarrow networks$
$N_j \leftarrow networks$
**for** $n_i$ in $N_i$ **do**
    **for** $n_j$ in $N_j$ **do**
        **if** $n_i$ subsumes $n_j$ **then**
            $N_j \leftarrow N_j \setminus n_j$
        **end if**
    **end for**
**end for**
     **return** $N_j$
```