

# Master Thesis

Javier González García

February 2021

## 1 Introduction

A sorting network is a formal representation of a sorting algorithm that for any inputs generates monotonically increasing outputs. A sorting network is formed by  $n$  channels each of them carrying one input, which are connected pairwise by comparators. A comparator compares the inputs from its 2 channels and outputs them sorted to the same 2 channels. A comparator network is a sorting network if for any input sequence the output is always the sorted sequence. What makes sorting networks special is their high parallelization capacity, we can create parallel layers of comparators as long as none of them is part of the same input channel at once.

The creation of optimal sorting networks can be divided in 2 tasks. Finding sorting networks with less comparators, also called size optimization. And finding sorting networks with less parallel layers, also called depth optimization. In this thesis we will focus in the size optimization.

The search of optimal size sorting networks involves to test all comparator networks of a given size. For example to prove the optimality of the sorting network with 11 inputs and 35 comparators we should consider  $55 = (11 \times 10)/2$  possibilities to place each comparator in 2 out of 11 channels. Therefore the search space is of  $55^{35} \approx 9 \times 10^{60}$  comparator networks.

This problem can only be addressed by using symmetry breaking rules to trim the search space. For this matter first I implemented the method used in [7] called generate and prune. This method is formed by 2 phases. In the generate phase, starting with a one comparator network it iteratively creates new networks with one comparator more in all possible positions. In the prune phase the redundant networks (networks equivalent to others in the set) are removed. This way the search space is reduced to  $2.2 \times 10^{37}$  to around  $3.3 \times 10^{21}$  for the 9 inputs network.

In the first part of this thesis, focused in improving the implementation using modern programming languages and reducing the memory and CPU consumption. This itself allowed to reproduce the same results than in [7] with only 10 hours of compute time in a 64 cores computer more modest than the one used in [7] which took one week of computing. However, when trying to address the next open problem, this itself is not enough. Due to the combinatory explosion it would be necessary around a year of computing time in this computer to find a solution for the 11 input problem. This led to the second part of this thesis where I apply heuristics to the generate and prune method. During the test of heuristic I discovered without prove a method that allows to discover nets with the same size than all the already best size networks until 13 inputs. For 14 inputs and above again the sets are too big to be tested in the available hardware.

The combination of heuristic functions with generate and prune has dealt promising results, finding networks of the same size than the state of the art smallest networks in the interval 3-16. In the following chapters I will state with further details the work performed in this master thesis.

## 2 Sorting networks preliminaries

A comparator network with  $n$  inputs is a sequence of comparators, each comparator is formed by a tuple of channels  $C = (i_1, j_1); \dots; (i_k, j_k)$  where  $(1 \leq i_l < j_l \leq n)$ . We name size,  $S(n)$ , to the number of comparators the network has. An input  $\bar{x} = x_1 \dots x_n \in \{0, 1\}^n$  outputs the network as follows:  $\bar{x}_0 = \bar{x}$  for  $0 < l \leq k$ ,  $\bar{x}^l$  is a permutation of  $\bar{x}^{l-1}$  exchanging  $\bar{x}_{i_l}^{l-1}$  and  $\bar{x}_{j_l}^{l-1}$  if  $\bar{x}_{i_l}^{l-1} > \bar{x}_{j_l}^{l-1}$ . A comparator network is a sorting network if for any  $n$  inputs the outputs are the ascending ordered sequence. The reason we take only binary sequences is because of the zero-one principle[6] which states that a comparator network orders all sequences in  $\{0, 1\}$  if and only if it sorts all sequences in any ordered set such as the integers set. This also allows to test if a comparator network is a sorting network without testing  $n!$  combination of sequences. If a comparator network tests all the binary sequences in  $1-n$  it is enough to state that it is indeed a sorting network.

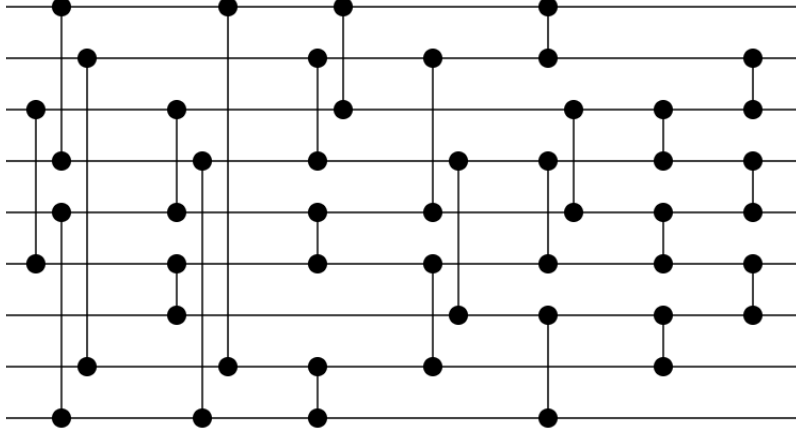


Figure 1: Size 8 sorting network

As stated before the creation of sorting networks with optimal size is the problem of finding networks with the smallest possible set of comparators. Until this date optimal size sorting networks exist for  $n \leq 12$ , in [2] Floyd and Knuth found optimal size sorting networks for  $n \leq 8$ . In [7] the optimal size networks for  $n = 9$  and  $n = 10$  are proved and [5] proves the optimal size networks for  $n = 11$  and  $n = 12$  using SAT encoders. The following lemma [10] is used to establish lower size bounds:

**Lemma 2.1**  $Size(n + 1) \geq Size(n) + \log_2 n$  for all  $n \geq 1$

Using the above lemma the sizes of  $S(10)$  was implied from  $S(9)$  in [7] and the size of  $S(12)$  was implied from  $S(11)$  in [5].

The method followed in this thesis is same than the one used in [7]. It makes use of the symmetries present in comparator networks to reduce the search space. These symmetries are formed by the permutation of channels. Given a comparator network  $C = c_1; c_2; \dots; c_k$  with size  $n$  where  $c = (i_t; j_t)$  with  $i \leq j \leq n$  and a permutation  $\pi$ .  $\pi(C)$  is the sequence  $\pi(c_1); \dots; \pi(c_k)$ . We call  $\pi(C)$  a generalized comparator network. This networks have the same properties than comparator networks with the exception that  $i_t$  can be bigger than  $j_t$ . In [6] an algorithm to convert any generalized sorting network to a standard sorting network with same size and depth is proposed.

To prove the size of  $S(n)$  we need to create a finite set of comparator networks that contains a sorting network. In the naive approach this set is all the comparator networks with  $n$  channels. In the next section I will explain how to reduce this set exploiting the symmetries in comparator networks.

### 3 Generate and Prune

As we have seen before, our objective is to generate a complete set for all the comparator networks with  $n$  channels and  $k$  comparators. To do that we start in the empty net and consecutively we add a new comparator in all possible ways. We call this complete set  $R_k^n$  in the naive approach adding a comparator in all possible positions deals to  $n*(n-1)/2$  comparator networks, as we have to repeat this step  $k$  times we end with  $(n*(n-1)/2)^k$  comparator networks. For  $k = 9$ ,  $R_{25}^9 \approx 3*10^{38}$  making this approach infeasible to compute. To reduce the search space the method Generate and Prune is introduced in [7] this algorithms take advantage of the symmetries in comparator networks. To explain the generate and prune approach we should introduce the next definition.

**Definition 3.1** *Given 2 comparator networks  $C_a$  and  $C_b$  and a permutation of the outputs  $\pi$  we say that  $C_a$  subsumes  $C_b$  if it exists any  $\pi(\text{outputs}(C_a)) \subseteq \text{outputs}(C_b)$ .*

**Lemma 3.2** *Given 2 comparator networks  $C_a$  and  $C_b$  if  $C_a$  subsumes  $C_b$  if there is a sorting network of the type  $C_b \cup C$  with size  $k$ , there is also a sorting network of the type  $C_a \cup C'$  with size  $k$ .*

The previous lemma implies that we can create a complete set of comparator networks while discarding the nets that are subsumed by any other net in the naive complete set. This is the base of the generate and prune algorithms. Starting with the empty set  $R_0^n$ , which corresponds to a comparator network with no comparators. The algorithms generate and prune work in the following way:

- Generate: Given the set  $R_k^n$  extends it to the set  $N_k^n + 1$  by adding a one extra comparator to each element in the previous set in any possible way.
- Prune: Given the set  $N_k^n + 1$  creates the set  $R_{k+1}^n$  removing any net subsumed by those nets that are not removed.

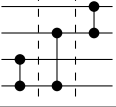
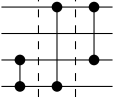
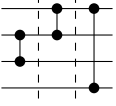
### 4 Generate and Prune implementation

In this section we explain how to implement the generate and prune methods of [7] and some optimizations that can be applied to these algorithms and the subsumption tests.

## 4.1 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. 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 need to store 2 more matrices and 1 array of size  $(n \times n)$  and  $n$  respectively (take into account that the matrices can be stored as a 1 dimension array and use bit operations to access to the matrix positions). The first 2 arrays store the positions where any output contains a 0 and a 1 and we call these 2 matrices  $W^0$  and  $W^1$  they are partitioned by the number of zeroes or ones. The last array contains the number of sequences with  $k$  1s in the outputs given  $k = 0, 1 \dots n$  and we call it  $S_k^1$ .

**Table 1** Comparator network outputs partitioned by number of 1s

| number of 1s  | 0    | 1                    | 2                    | 3            |
|---|------|----------------------|----------------------|--------------|
| $C_1$   | 0000 | 0001<br>1000         | 1010<br>1100         | 1011<br>1110 |
| $C_2$  | 0000 | 1000<br>0010         | 1010<br>1100         | 1110<br>1101 |
| $C_3$  | 0000 | 0010<br>0100<br>1000 | 0110<br>1010<br>1100 | 1110         |

To improve the memory usage as much as possible and make it more computationally efficient, the outputs array is implemented as an array of  $2^n$  bits where a position is set to 1 if the output is set and 0 otherwise. This allows to represent the outputs with only 512 bits in the case of  $n = 9$ . This also allows to generate the outputs very fast by the use of bitwise operations that are way faster in modern programming languages.

## 4.2 Generate and Prune implementation

In this section we will explain some details in the implementation of Generate and Prune, the algorithms are same than the ones in [7]. The generate and prune algorithms are very simple but they run in massive datasets that make their implementation details crucial to be able to handle the problem for inputs sizes in  $n \geq 8$ .

### 4.2.1 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  $N_n^k$ .

We apply the optimization seen in [7] to avoid adding redundant comparators (networks that order less or the same than it's parent network). These networks would be eliminated either way by the prune algorithm, but removing it in this stage is cheaper in computation time. Given the comparator network  $C$  and the comparator  $c : (i, j)$  we can extend this network to  $C' = C \cup (i, j)$ . We say that the comparator  $(i, j)$  is redundant if for all the elements in the sequence  $x_1 \dots x_n \in \text{outputs}(C)$   $x_i \leq x_j$ .

---

#### 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  $n \leftarrow 1 \dots N$  do
  for  $c$  in  $C$  do
     $n' \leftarrow n \cup c$ 
    if  $n'$  is not redundant then
       $N_n^{k+1} \leftarrow N_n^{k+1} \cup n'$ 
    end if
  end for
end for
return  $R_n^{k+1}$ 

```

---

### 4.2.2 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
         $R \leftarrow R \cup n$ 
    end if
end for
return  $R$ 

```

---

#### 4.2.3 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 [7]. 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 behind.

---

**Algorithm 3** Parallel Prune

---

```
 $N \leftarrow networks$   
 $C \leftarrow Divide(N)$   $\triangleright$  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  $N_j$ 
```

---

## 5 Subsume Implementations

The subsumption test involves searching for a permutation of the outputs where  $\pi(outputs(C_1) \subseteq outputs(C_2))$  that can involve in the worst case the check of  $n!$  permutations. In the case of  $n = 9$   $n! = 362880$ . However, in many cases it is not necessary to test any permutation to prove that a net does not subsume other. In this section we explain the techniques used in [7] and discuss implementation details for creating a subsume



**Table 2** Comparator network outputs partitioned by number of 1s

| number of 1s | 1                    | 2                            | 3                    |
|--------------|----------------------|------------------------------|----------------------|
| $C_1$        | 0010<br>1000         | 0110<br>1010<br>1100         | 0111<br>1110         |
| $C_2$        | 0010<br>0100<br>1000 | 0101<br>0110<br>1010<br>1100 | 0111<br>1101<br>1110 |

If we take a look at table 2, we can see that the number of outputs with one 1s in  $C_1$  is smaller than the one in  $C_2$ . Given this, it is clear that  $C_1$  will not subsume  $C_2$  because a permutation of the outputs of  $C_1$ , will never be a subset of the outputs of  $C_2$ . More formally we can define this as the following lemma.

**Lemma 5.1** *Given 2 comparator networks  $C_a$  and  $C_b$  with  $n$  channels and their arrays containing the number of sequences with  $k$  1s  $S1_k^1$  and  $S2_k^1$ . If for any  $k \geq 0 \leq n$ ,  $S1_k^1 > S2_k^1$   $C_1$  does not subsume  $C_2$ .*

In [7] they state that 70% of the subsumption tests are discarded on application of previous lemma.

If we look again to the outputs in the figure above, in the column that contains the outputs with  $X$  number of 1s we can also notice that  $C_1$  does not subsume  $C_2$ . That's because the digit 0 appears only in  $X$  positions in the sequences in  $C_1$  while in  $Y$  positions in the sequences of  $C_2$  and that will stay the same given any permutation. More formally it can be expressed in the following lemma:

**Lemma 5.2** *Given 2 comparator networks  $C_a$  and  $C_b$  with  $n$  channels with their arrays  $W^x$ ,  $x \in \{0,1\}$  and  $0 \leq k \leq n$  we denote  $P_k^x$  to the number of positions that  $x$  appears in  $k$  partition of  $W^x$ . If  $P^x(C_a, k) > W^x(C_b, k)$  then  $C_a$  does not subsume  $C_b$ .*

In [7] they state that 15% of the subsumption tests that are not discarded on application of Lemma 4.1 are discarded on application of Lemma 4.2.

If the subsumption test is not discarded in application of previous lemmas, there is no other way than trying to find a permutation that satisfies the condition of  $\pi(\text{outputs}(C_a)) \subseteq \text{outputs}(C_b)$  as we stated before the running time of this is  $O(n!)$  in the case that  $C_a$  does not subsume  $C_b$ . However, in many cases

we can avoid lots of checks by using the  $W^0$  and  $W^1$  arrays. Which leads to the following lemma:

**Lemma 5.3** *Given 2 comparator networks  $C_a$  and  $C_b$  with  $n$  channels with their arrays  $W^x$ ,  $x \in \{0,1\}$  and  $0 \leq k \leq n$ . If it exists a permutation  $\pi(\text{outputs}(C_a)) \subseteq \text{outputs}(C_b)$  then  $\pi(W^x(C_a, k)) \subseteq (W^x(C_b, k))$ .*

The above lemma allows in practice to skip thousands of permutations and is the base for the 2 algorithms implemented in this thesis to skip permutations. The first one uses a backtracking algorithm that avoids expanding branches that cannot lead to a feasible solution. The second one is extracted from [3] and represents the positions matrix as a bipartite graph to find possible permutations. In the next 2 sections we will explain them in more detail.

## 5.1 Permutations with backtracking

As an example we have the next 2 Comparator networks represented by their comparators pairs  $C_1 = [(2, 4), (2, 3), (1, 3)]$  and  $C_2 = [(1, 4), (3, 4), (1, 3)]$ . Now we represent the outputs of these networks partitioned by their number of 1s (we discard the trivial outputs with all ones and all zeroes).

With the partitioned sets of 2 we can create their  $W^x$  matrices. Which allow us to know in which positions the outputs contain a zero or a one.

**Table 3** W matrices partitioned by number of 1s or 0s

| number of 1s or 0s | 1    | 2    | 3    |
|--------------------|------|------|------|
| $W1^0$             | 1111 | 1111 | 1001 |
| $W1^1$             | 1110 | 1110 | 1111 |
| $W2^0$             | 1111 | 1111 | 1011 |
| $W2^1$             | 1110 | 1111 | 1111 |

The above matrices show the positions that  $x \in \{0,1\}$  appears in the Outputs array of the comparator network. If there is a 0 in certain position it means that  $x$  doesn't appear in that position in the outputs array. For example, if we look at the partition with 3 zeroes in the  $W1^0$  matrix we see that 0 only appears in the positions 1 and 4.

Combining the information of these 4 matrices we can obtain another matrix that will tell us the forbidden positions for the permutations. We call this matrix

$P$ , each row refers to an index  $0, \dots, n$  in the outputs, if there is a 0 in a specific position it means that the permutation cannot contain that number in that specific position. To create the positions matrix we iterate the elements in both  $W$  matrices for  $C_1$ . If there is a 1 in that position for any of the matrices we check if there is also a 1 in the correspondent matrices of  $C_2$ , in that case that position is allowed and we set it to 1. For the given comparator networks we obtain the following positions matrix.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

If we look carefully into the matrix, we can see that the 3rd column contains only zeroes. So, no index is allowed in the third position. This way we can say that  $C_1$  does not subsume  $C_2$  without trying any of the permutations. This applies also to the case that a row is all zeroes, which means that the bit in that position cannot be permuted to any of the other positions. This matrix allows to speed up the subsume operation and implement it as a backtracking algorithm which skips a huge amount of permutations. In table 4 there is a comparison between the number of permutations checked while using the positions matrix with respect to not using it for comparator networks with 6 inputs.

**Table 4** Permutations tested for  $n = 6$  with and without skipping permutations

| comparator       | 1     | 2     | 3     | 4    | 5    | 6    | 7    | 8    | 9    | 10    | 11  | 12 |
|------------------|-------|-------|-------|------|------|------|------|------|------|-------|-----|----|
| skipping         | 14    | 31    | 181   | 330  | 1031 | 1373 | 1301 | 966  | 281  | 93    | 17  | 4  |
| not skipping $P$ | 2.8e3 | 4.3e3 | 4.5e3 | 99e3 | 27e4 | 54e4 | 59e4 | 41e4 | 19e4 | 9.5e3 | 259 | 4  |

---

**Algorithm 5** Subsumes

---

$n$  networks input size  
 $C_1$  comparator network  
 $C_2$  comparator network  
 $P$  positions matrix  
 $column \leftarrow 0$   
 $\pi \leftarrow \emptyset$   $\triangleright$  permutation  
**procedure** TRYPERMUTATIONS( $\pi, column$ )  
  **for**  $row \leftarrow 1..n$  **do**  
    **if**  $P[row][column] = 1$  **then**  
       $\pi[column] \leftarrow row$   
      **if** TRYPERMUTATIONS( $\pi, column + 1$ ) **then return** TRUE  
      **end if**  
    **end if**  
  **end for**  
  **if**  $\pi(outputs(C_2)) \subset outputs(C_1)$  **then return** TRUE  
  **else return** FALSE  
  **end if**  
**end procedure**

---

## 5.2 Bigraph perfect matchings

# 6 Heuristics

In this section we study heuristic methods that can reduce the population size after the Generate and Prune algorithms while still leading to promising results. In particular 3 methods were used. The first 2 methods discard networks based in some characteristics in their outputs sets. And the last one sets as subsumed any network that arrives to the last phase of the subsumption test, so instead of checking the permutations of the outputs we just throw that network.

### 6.1 Remove big outputs heuristic

A sorting network of size  $n$  has  $n + 1$  binary outputs. Consequently, we can say that networks containing less outputs order "better" than those containing more. In this Heuristic we order the set of networks by their number of outputs and remove the ones with bigger output's number until a given population size. This heuristic is one of the most common in the literature such as [8] and [1].

## 6.2 Bad positions heuristic

This heuristic is defined in [4]. It takes a comparator network and gives it a score based on the number of "bad positions" that its outputs contain. A bad zero is defined as follows:

Let  $C$  be a comparator network of with  $n$  inputs, if  $C$  is a sorting network then for any input sequence  $x$  that contains  $p$  zeroes the output will be the sequence that contains zeroes in the first  $p$  positions and ones in the rest  $n - p$  positions. We call bad position to those ones that contain a zero in a place that a 1 should be. The evaluation function is as follows:

$$f(C) = \frac{1}{(n+1) \cdot (2^n - 1)} * (2^n * |bad(C)| + |outputs(C)| - n - 1)$$

## 6.3 Prune with no permutations heuristic

The last heuristic that we tried is pruning every network which subsumption is not discarded based in the Lemmas of section 5. So, given  $C_1$  and  $C_2$  comparator networks when checking if  $C_1$  subsumes  $C_2$ . If the subsumption is not discarded applying the Lemmas of section 5, instead of trying to find a permutation  $\pi$  that  $\pi(outputs(C_a)) \subset outputs(C_b)$  we directly mark  $C_b$  as subsumed. When applying this rule alone, the sizes of the biggest sets are reduced up to a 1% of those of the Generate and Prune algorithm studied before. This heuristic alone was able to find always a sorting network with the already best known sizes in the interval 2...13 and in combination with the Remove big outputs heuristic we found networks with the best known sizes in the sizes until  $n = 16$ . In 5 we compare the sets after prune for  $n = 9$  of both implementations:

**Table 5** Sizes of filters for  $n = 9$

| $k$     | 1 | 2 | 3 | 4  | 5  | 6   | 7   | 8    | 9      | 10     | 11      | 12      | 13      |
|---------|---|---|---|----|----|-----|-----|------|--------|--------|---------|---------|---------|
| $R_k^9$ | 1 | 3 | 7 | 20 | 59 | 208 | 807 | 3415 | 14,343 | 55,951 | 188,730 | 480,322 | 854,638 |
| $S_k^9$ | 1 | 2 | 3 | 7  | 13 | 22  | 41  | 77   | 136    | 229    | 302     | 403     | 531     |

| k    | 14      | 15      | 16      | 17     | 18     | 19   | 20   | 21  | 22 | 23 | 24 | 25 |
|------|---------|---------|---------|--------|--------|------|------|-----|----|----|----|----|
| R9k  | 914,444 | 607,164 | 274,212 | 94,085 | 25,786 | 5699 | 1107 | 250 | 73 | 27 | 8  | 1  |
| R9k' | 586     | 570     | 519     | 413    | 314    | 230  | 179  | 123 | 57 | 24 | 8  | 1  |

Sorting networks with reduced population sets combining big outputs and prune with no permutations heuristics:

| size            | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-----------------|----|----|----|----|----|----|----|----|
| 100 population  | 25 | 30 | 35 | 39 | 48 | 53 | 59 | 63 |
| 250 population  | 25 | 30 | 35 | 39 | 48 | 53 | 59 | 63 |
| 500 population  | 25 | 29 | 35 | 39 | 48 | 53 | 59 | 61 |
| 1000 population | 25 | 29 | 35 | 39 | 48 | 51 | 59 | 61 |
| 5000 population | 25 | 29 | 35 | 39 | 47 | 51 | 57 | 60 |
| Best known size | 25 | 29 | 35 | 39 | 45 | 51 | 56 | 60 |

Found sorting network with 19 inputs and 86 comparators. Same size than [9]. Would have been record in 2013.

## 7 Conclusion

## References

- [1] Sung-Soon Choi and Byung-Ro Moon. Isomorphism, normalization, and a genetic algorithm for sorting network optimization. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, GECCO'02*, page 327–334, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.
- [2] ROBERT W. FLOYD and DONALD E. KNUTH. Chapter 15 - the bose-nelson sorting problem††the preparation of this report has been supported in part by the national science foundation, and in part by the office of naval research. In JAGDISH N. SRIVASTAVA, editor, *A Survey of Combinatorial Theory*, pages 163–172. North-Holland, 1973.
- [3] Cristian Frăsinaru and Mădălina Răschip. An improved subsumption testing algorithm for the optimal-size sorting network problem. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 292–303, Cham, 2019. Springer International Publishing.
- [4] Cristian Frăsinaru and Mădălina Răschip. Greedy best-first search for the optimal-size sorting network problem. *Procedia Computer Science*, 159:447–454, 2019. Knowledge-Based and Intelligent Information Engineering Systems: Proceedings of the 23rd International Conference KES2019.
- [5] Jannis Harder. An answer to the bose-nelson sorting problem for 11 and 12 channels, 2021.

- [6] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.
- [7] Michael Codish; Luís Cruz-Filipe; Michael Frank; Peter Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *Journal of Computer and System Sciences*, 2016.
- [8] Lukáš Sekanina and Michal Bidlo. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, Sep 2005.
- [9] Vinod K. Valsalam and Risto Miikkulainen. Using symmetry and evolutionary search to minimize sorting networks. *Journal of Machine Learning Research*, 14(Feb):303–331, 2013.
- [10] David C. Van Voorhis.