

# **Exploring size bounds for Sorting Networks**

**Author: Javier González García**

**Supervisors: Luis Cruz-Filipe, Jacopo Mauro**

A thesis presented for Msc  
in Computer Science



Syddansk Universitet  
Denmark  
May 30, 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.

Optimizing sorting networks can be done in two ways. Finding sorting networks with fewer comparators, also called size optimization. And finding sorting networks with fewer parallel layers, also called depth optimization. In this thesis we will focus on size optimization.

The only known way of finding optimal size sorting networks involves testing all comparator networks of a giving 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.

One of the ways to address this problem is by using symmetry breaking rules to trim the search space. For this purpose, the authors of [9] proposed a method that they called generate-and-prune. This method is formed by two phases that alternate, starting from an empty comparator network. In the generate phase, it iteratively creates new networks with one comparator more in all possible positions. In the prune phase, starting with the output set of the Generate algorithm, 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, I focused on implementing and improving generate-and-prune using modern programming languages and reducing the memory and CPU consumption. This itself allowed to reproduce the same results as in [9] (demonstrate that the minimum number of comparators to sort 9 inputs is 25), with only 10 hours of compute time in a 64-core computer more modest than the one used in [9] 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. One of the heuristics tested is able to discover networks with the same size than all the already best known sizes for networks until 13 inputs. For 14 inputs and above again the intermediate sets generated during

generate-and-prune are too big to manage 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  $C = (i_1, j_1); \dots; (i_k, j_k)$  where  $1 \leq i_l < j_l \leq n$ . We call each pair  $(i_l, j_l)$  a comparator. The size of a networks is the number of comparators the network has. We denote by  $S(n)$  the minimum size a sorting network with  $n$  inputs can have.

An input  $\bar{x} = x_1 \dots x_n \in \{0, 1\}^n$  travels the sorting network as follows:  $\bar{x}_0 = \bar{x}$ ; for  $0 < l \leq k$ ,  $\bar{x}^l$  is the permutation of  $\bar{x}^{l-1}$  obtained by exchanging  $\bar{x}_{i_l}^{l-1}$  and  $\bar{x}_{j_l}^{l-1}$  if  $\bar{x}_{i_l}^{l-1} > \bar{x}_{j_l}^{l-1}$ . The output corresponding to  $\bar{x}$  is  $\bar{x}^k$ . A comparator network is a sorting network if the output corresponding to any input is sorted ascending.

The reason we take only binary sequences is because of the zero-one principle [8] 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). This also allows to test if a comparator network is a sorting network without testing  $n!$  combinations of sequences. If a comparator network orders all the  $2^n$  binary sequences in  $0, 1^n$  then it is indeed a sorting network.

A sorting network can be represented as seen in Figure 1 where the horizontal lines represent the input channels, the vertical lines joining 2 channels represent a comparator and the vertical dashed lines represent a parallel layer (operations inside a parallel layer are applied simultaneously).

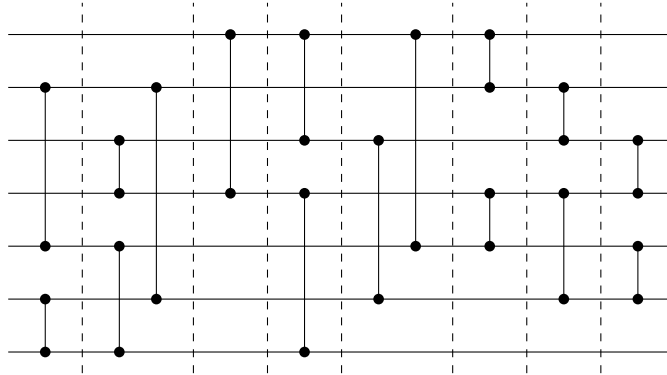


Figure 1: Sorting network with 7 inputs and 16 comparators in 7 layers

As stated before, the optimal-size sorting network problem consists in finding networks with the smallest possible set of comparators. Until this date, optimal size sorting networks are known for  $n \leq 12$ , Optimal size sorting networks for  $n \leq 8$  were found in [4] by Floyd and Knuth.  $S(9)$  and  $S(10)$  was proved in [9] and  $S(11)$  and  $S(12)$  in [7].

The following lemma [2] is used to establish lower size bounds:

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

Using the above lemma the value of  $S(10)$  was derived from  $S(9)$  in [9] and the value of  $S(12)$  was derived from  $S(11)$  in [7].

The method in [9] 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  $n$  and a permutation  $\pi$ ,  $\pi(C)$  is the sequence  $\pi(c_1); \dots; \pi(c_k)$  where  $\pi(i, j) = (\pi(i), \pi(j))$ . The network  $\pi(C)$  is a generalized comparator network. Generalized comparator networks are defined as comparator networks with the exception that  $i_t$  can be bigger than  $j_t$ . In [8] it is shown that any generalized sorting network can be converted to a standard sorting network with the same size and depth.

In order to find the value of  $S(n)$ , we create what is called a complete set of filters for the optimal-size sorting network problem. The author of [9] give the following definition.

**Definition 2.2** *A finite set  $F$  of comparator networks on  $n$  channels is a complete set of filters for the optimal size sorting network problem on  $n$  channels if there exists an optimal-size sorting network on  $n$  channels created by extension of the current set of networks for some  $C \in F$ .*

In the naive approach this set contains all the comparator networks with  $n$  channels. In the next section, we will explain how to reduce this set exploiting the symmetries in comparator networks. Afterwards, we explore some heuristics that help restricting more the search space discarding candidates based in different evaluation functions to try to get smaller networks than the actual ones.

### 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 will use the generate-and-prune method seen in [9].

We start in the empty net and consecutively we add a new comparator in all possible ways. For comparator networks with  $n$  inputs and  $k$  comparators, we call this complete set  $N_k^n$ . In the naive approach adding a comparator in all possible positions deals to  $n \times (n - 1)/2$  comparator networks, as we have to repeat this step  $k$  times we end with  $(n \times (n - 1)/2)^k$  comparator networks, we call this Generate. However, for  $n = 9$ ,  $N_{25}^9 \approx 3 \times 10^{38}$  making this approach infeasible to compute. To reduce the search space, the Prune method will take the set  $N_k^n$  and reduce it to the set  $R_k^n$  that contains the minimum complete set of filters for  $k$  comparators, afterward the Generate method will receive  $R_k^n$  and extend it to  $N_{k+1}^n$  by extending each comparator network with a new comparator in all possible positions. This sequence is repeated until the set  $R^nk$  contains just one network that will be a sorting network, as the size of the minimum set of filters for the size problem when it contains sorting network is 1.

**Definition 3.1** *Given 2 comparator networks  $C_a$  and  $C_b$  we say that  $C_a$  subsumes  $C_b$  if there exists some permutation  $\pi$  such that  $\pi(\text{outputs}(C_a)) \subset \text{outputs}(C_b)$ .*

The following lemma from [9] proves the completeness of the set of filters after Prune.

**Lemma 3.2** *Given 2 comparator networks  $C_a$  and  $C_b$  where  $C_a$  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 networks that are subsumed by any other network 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 the empty comparator network. The algorithms work in the following way:

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

## 4 Generate and Prune implementation

In this section we explain how to implement the generate and prune methods of [9] 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 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  $k = 0, 1, \dots, n$  and we call it  $S^1$ .

<b>Table 1</b> Comparator network outputs partitioned by number of 1s				
number of 1s	0	1	2	3
$C_1$	0000	0001 1000	1010 1100	1011 1110
$C_2$	0000	1000 0010	1010 1100	1110 1101
$C_3$	0000	0010 0100 1000	0110 1010 1100	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.

## 4.2 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 [9]. 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  $n \geq 8$ .

### 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  $N_n^k$ .

Given the comparator network  $C$  and the comparator  $(i, j)$  we can extend this network to  $C' = C \cup (i, j)$ . We apply the optimization seen in [9] 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  $x_1 \dots x_n \in \text{outputs}(C)$   $x_i \leq x_j$ .

---

#### 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 [9]. 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  $N_j$ 

```

---

## 5 Subsume Implementations

The subsumption test involves searching for a permutation of the outputs where  $\pi(\text{outputs}(C_1) \subseteq \text{outputs}(C_2))$  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 network does not subsume other. In this section we explain the techniques used in [9] and discuss implementation details for the subsumption test.

**Table 2** 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

If we take a look at table 2, we can see that the number of outputs in the partition 1 of  $C_1$  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 state 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  $S1$  and  $S2$ . If for any  $k \geq 0 \leq n$ ,  $S1_k > S2_k$   $C_1$  does not subsume  $C_2$ .*

The authors of [9] state that 70% of the subsumption tests are discarded on application of previous lemma.

If we look again to the outputs in the table 2, in the partition 3 we can also notice that  $C_1$  does not subsume  $C_2$ . That's because the digit 0 appears only in 2 positions in the sequences for  $C_1$  while in 3 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 [9] it is stated that 15% of the subsumption tests that are not discarded on application of Lemma 5.1 are discarded on application of Lemma 5.2.

If the subsumption test is not discarded in application of previous lemmas, the authors of [9] then try to find a permutation that satisfies the condition of  $\pi(\text{outputs}(C_a)) \subset \text{outputs}(C_b)$ . As we stated before the running time of this is  $O(n! \times f(n))$  where  $f(n)$  is the time to check if  $\pi(\text{outputs}(C_a)) \subset \text{outputs}(C_b)$  in the worst case. However, in many cases we can avoid lots of permutations checks. With the partitioned output sets stated before we use matrices  $W^0$  and  $W^1$  described before. By combining the matrices of 2 different comparator networks we can create another matrix that will contain the forbidden positions for the output indices. This way we avoid testing permutations with positions that are known to be incorrect.

**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 [5] 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 comparator 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.

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.

**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

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  $0..n$  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
 $j \leftarrow 0$ 
 $\pi \leftarrow \emptyset$  ▷ permutation
procedure TRYPERMUTATIONS( $\pi, j$ )
  for  $i \leftarrow 1..n$  do
    if  $P[i][j] = 1$  then
       $\pi[column] \leftarrow row$ 
      if TRYPERMUTATIONS( $\pi, column + 1$ ) then return TRUE
    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 generate-and-prune 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 avoids the permutations check in the prune phase, setting as subsumed any network that arrives to that stage. Combinations of these heuristics have dealt promising results finding minimal sorting networks.

### 6.1 Remove networks with more 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 [10] and [3].

## 6.2 Bad positions heuristic

This heuristic is defined in [6]. 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 that contain a zero in a place where a 1 should be. The evaluation function is as follows:

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

## 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. 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. This way we reduce the subsume operation complexity from  $O(n!)$  to  $O(1)$ . When applying this rule alone, the sizes of the biggest sets are reduced up to a around 0.05% of those of the subsumption test 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 with inputs up to  $n = 16$ . In table 5 we compare the sets after prune for  $n = 9$  of both implementations. To put it into perspective, while the traditional approach took 10 hours and a half in a computer with 126 processors to deal a result for  $R_9^{25}$ , with this heuristic we get a sorting network of size 25 in one second using a laptop.

**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

## 6.4 Heuristics results

In this section we state the results of using the heuristics and combinations of them. Both remove networks with more outputs and bad positions heuristic have similar results and in combination with prune with no permutations lots of networks were found with relatively small population sizes. In the next tables we will find comparisons of the proposed heuristic with respect to the best known sorting network sizes, using different population sizes. The information for the sizes of the networks has been extracted from [1].

In the next table we can see the results obtained when combining the heuristics of remove networks with more outputs and prune with no permutations.

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

As we can see

The bad positions heuristic combined with prune without permutations have accomplished poorer results in comparison. With populations up to 5000, we could only get the best results for the networks from 9 to 12 inputs, while the best sizes for the bigger input networks are far from the state of the art best ones.

inputs	9	10	11	12	13	14	15	16
$S(n)$ population 100	25	31	37	46	52	62	72	82
$S(n)$ population 250	25	30	37	42	51	60	70	80
$S(n)$ population 500	25	30	36	41	51	59	69	78
$S(n)$ population 1000	25	30	35	41	49	58	67	77
$S(n)$ population 5000	25	29	35	39	48	54	64	X
Best known $S(n)$	25	29	35	39	45	51	56	60

Found sorting network with 19 inputs and 86 comparators. Same size than [11]. Would have been record in 2013.



## 7 Conclusion

## References

- [1] Smallest and fastest sorting networks for a given number of inputs. [http://users.telenet.be/bertdobbelare/SorterHunter/sorting\\_networks.html](http://users.telenet.be/bertdobbelare/SorterHunter/sorting_networks.html). Accessed: 2021-05-26.
- [2] Van Voorhis; David C.; Thatcher; James W., and Bohlinger; Jean D. *Toward a Lower Bound for Sorting Networks*, pages 119–129. Springer US, Boston, MA, 1972.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] Jannis Harder. An answer to the bose-nelson sorting problem for 11 and 12 channels, 2021.
- [8] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.
- [9] Michael Codish; Luís Cruz-Filipe; Michael Frank; Peter Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *Journal of Computer and System Sciences*, 2016.
- [10] 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.

- [11] 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.