Exploring size bounds for Sorting Networks

Author: Javier González García

Supervisors: Luís Cruz-Filipe, Jacopo Mauro

A thesis presented for MSc.
in Computer Science

Syddansk Universitet
Denmark
June 1, 2021

Contents

1 Introduction

A sorting network is a data-oblivious sorting algorithm, which means that the order of comparisons to be performed is only determined by the number of inputs to be sorted and not the order of these values. 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 two channels and outputs them sorted to the same channels. Such a sequence of comparators is called a comparator network. A comparator network is a sorting network if for any input, 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 are 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 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.

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 [8] 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, the method 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 2.2×10^{37} to around 3.3×10^{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 [8] (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, while in [8] they used 144 threads and took over one week of computing. However, when trying to address the next open problem, minimum size for 11-input sorting networks, 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 in order to find sort-

ing networks better than the best known. 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 using 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.

During the next sections I will present the work performed in this master thesis. In section 2 I will give some preliminaries in sorting networks and introduce the notation, in section 3 I will explain how to implement the generate-and-prune algorithm from [8]. Sections 4 and 5 explain the techniques and heuristics to lower the search for the optimal-size problem and are the main focus of this master thesis.

The source code used for the implementation of this thesis can be accessed in: https://github.com/vitota95/SortingNetworks.

2 Sorting networks preliminaries

A comparator network with n inputs is an array $C = (i_1, j_1); ..; (i_k, j_k)$ where $1 \leq i_l < j_l \leq n$. We call each pair (i_l, j_l) a comparator. The size of a network 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 $\bar{x} = x_1...x_n \in \{0,1\}^n$ 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 [7] 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 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).

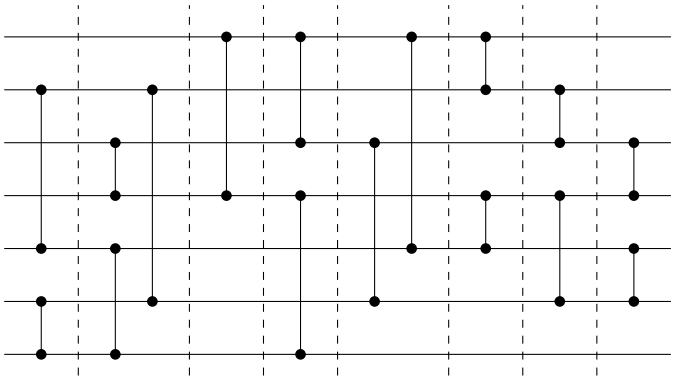


Figure 1: Sorting network with 7 inputs and 16 comparators in 7 layers

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 [3] by Floyd and Knuth. $S(9) = 25$ and $S(10) = 29$ was demonstrated in [8] and $S(11) = 35$ and $S(12) = 39$ was claimed in [6] which is

still under peer-review.

The following lemma [12] 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 values of $S(6)$ and $S(8)$ were derived from $S(5)$ and $S(7)$ respectively in [3], $S(10)$ was derived from $S(9)$ in [8] and $S(12)$ was derived from $S(11)$ in [6].

The method in [8] makes use of the symmetries present in comparator networks to reduce the search space. These symmetries are described using permutation of channels. Given a comparator network $C = c_1; c_2; ..; c_k$ with size n and a permutation π, $\pi(C)$ is the sequence $\pi(c_1); ...; \pi(c_k)$ 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 [7] 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 authors of [8] 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 best known 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 [8].

3.1 Algorithm definition

The algorithm generate-and-prune iteratively creates minimum complete sets of filters for the optimal size problem.

In the Generate phase, adding a comparator in all possible positions deals to $n \times (n-1)/2$ comparator networks. Repeating this step k times would end with $(n \times (n-1)/2)^k$ comparator networks. Therefore, the Prune method reduces the search space ensuring a minimum set of filters for k comparators.

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 π such that* $\pi(outputs(C_a)) \subset outputs(C_b)$.

The following lemma from [8] 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.

The algorithms use two families of complete sets of filters R_k^n and N_k^n. It works as follows:

1. We start with R_0^n, which contains only the empty network.
2. We compute N_{k+1}^n from R_k^n (Generate) by extending each comparator network with a new comparator in all possible positions.
3. We compute R_{k+1}^n from N_{k+1}^n (Prune) by removing the networks that are subsumed by others.

4. We repeat from second step until the set $R^n k$ contains just one network that will be a sorting network

When a sorting network is found, the set R_k^n becomes a singleton because a sorting networks with n inputs and k comparators trivially subsumes any comparator network with n inputs and k comparators.

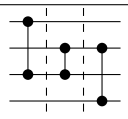
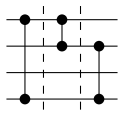
3.2 Representation of comparator networks

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 two more matrices and one 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 two 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 do not contain a 0 or a 1 in that position.

The array contains the number of sequences with k 1s in the outputs in positions $k = 0, 1...n$ and we call it S^1.

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

In Table 1 we can see an example of how the outputs are generated and partitioned for two 4-input comparator networks.

Table 2 W^0 and W^1 matrices and S^1 array for C_1 and C_2

C_1	C_2
$W^0 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$	$W^1 = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$
$W^0 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$	$W^1 = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$
$S^1 = \begin{bmatrix} 2 & 3 & 2 \end{bmatrix}$	$S^1 = \begin{bmatrix} 1 & 4 & 3 \end{bmatrix}$

In Table 2 we present the W^x matrices and S^1 array that are calculated directly from the set of outputs.

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 way we represent the array in an efficient way. 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.

3.3 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 [8]. 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 $n \geq 8$.

3.3.1 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 [8] 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} ← ∅
R_n^k ← complete set of filters for k comparators
C ← comparators array
for n ← 1...N do
    for c in C do
        n' ← n ⋃ c
        if n' is not redundant then
            N_n^{k+1} ← N_n^{k+1} ⋃ n'
        end if
    end for
end for
    return R_n^{k+1}
```

3.3.2 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 ← output of Generate
R_n^k ← ∅
subsumed ← False
for n in N do
    for r in R do
        if r subsumes n then
            subsumed ← True
            break
        end if
        if n subsumes r then
            R ← R \ r
        end if
    end for
    if not subsumed then
        R ← R ⋃ n
    end if
end for
    return R
```

3.3.3 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 [8]. 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 add mark a networks as not subsumed it should be checked against all the other not subsumed networks.

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

Algorithm 4 Remove

```
result ← ∅
N_i ← networks
N_j ← networks
for n_i in N_i do
    for n_j in N_j do
        if n_i subsumes n_j then
            N_j ← N_j \ n_j
        end if
    end for
end forreturn N_j
```

3.4 Optimizing the Subsumption Test

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 network does not subsume other. In this section we explain the techniques used in [8] and discuss implementation details for the subsumption test.

If we take a look at Table **??**, 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 3.3 *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 $0 \geq k \leq n$, $S1_k > S2_k$ C_1 does not subsume C_2.*

The authors of [8] state that 70% of the subsumption tests are discarded on

application of previous lemma.

If we look again to the outputs in the Table **??**, 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 two 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 3.4 *Given two comparator networks C_a and C_b with n channels with their arrays W^x, $x \in \{0,1\}$ and $0 \geq k \geq n$ 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 [8] 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 [8] then try to find a permutation that satisfies the condition of $\pi(outputs(C_a)) \subset outputs(C_b)$. As we stated before the running time of this is $O(n! \times f(n))$ where $f(n)$ is the time to check if $\pi(outputs(C_a)) \subset outputs(C_b)$ in the worst case. However, in many cases we can avoid lots of permutations. With the partitioned output sets stated before we use matrices W^0 and W^1 described before. By combining the matrices of two 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 3.5 *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 $\pi(outputs(C_a)) \subseteq outputs(C_b)$ 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 two algorithms implemented in this thesis to skip permutations.

4 Subsumption Test Implementation

As we explained before, the key part for the speed up of the algorithm it's in the subsume phase, as one of this tests is in $O(n!)$. In [8] they make use of Prolog's backtracking mechanism for this matter. However, this mechanism is not available in C# (which is the programming language used for the implementation of this thesis), so I explored 2 different algorithms for skipping this permutations. Both of them make use of Lemma 3.5 and the W^x matrices.

The first one uses a backtracking algorithm that avoids expanding branches that cannot lead to a feasible solution, and it is the one used in this thesis. The second one is extracted from [4] and represents the positions matrix as a bipartite graph to find possible permutations.

4.1 Permutations with backtracking

As an example we have the next two 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 in Table **??** we can create their W^x matrices. Which allow us to know in which positions the outputs contain a 0 or a 1.

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 $0, ..n$ 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 over the elements in W^x 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.

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

Figure 2: Positions matrix

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

The Subsumes algorithm can be implemented as follows:

Algorithm 5 Subsumes

```
n networks input size
C_1 comparator network
C_2 comparator network
P positions matrix
j ← 0
π ← ∅                                                    ▷ permutation
procedure TryPermutations(π, j)
    for i ← 1..n do
        if P[i][j] = 1 then
            π[column] ← row
            if TryPermutations(π, column + 1) then return TRUE
            end if
        end if
    end for
    if π(outputs(C_2)) ⊂ outputs(C_1) then return TRUE
    else return FALSE
    end if
end procedure
```

4.2 Bigraph perfect matchings

This approach takes into consideration the same positions matrix but, instead of the backtracking mechanism, it understand it as an adjacency matrix of a bipartite graph. A bipartite graph is composed by two disjoint sets given $G = (U, V, E)$ all edges connect U with V. We can easily understand this as the positions where a certain bit it's allowed to be positioned.

The authors of [4] propose an algorithm that finds all the possible permutations. For that they find all the perfect matchings in the bipartite graph.

A matching M in the graph G is an independent set of edges (no vertex is repeated), if that matching is of size n (contains all the vertex in U and V) we call it perfect matching. In Figure 3 we can find the bipartite graph representation of Figure 2.

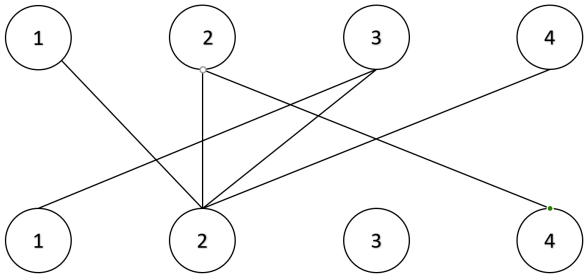


Figure 3: Bipartite graph representation of the positions matrix for C_1 and C_2

The authors of [4] introduce the following Lemma:

Lemma 4.1 *Given 2 comparator networks C_a and C_b with n channels. If C_a subsumes C_b via the permutations π then π represents a perfect match in the bipartite graph.*

Therefore, only the permutations that are a perfect match in the bipartite graph should be tried.

In order to obtain all the perfect matchings in the bipartite graph, they implement the algorithm in [10] that takes as start point a perfect matching that can be found with the Ford-Fulkerson algorithm and later finds the rest perfect matchings taking $O(n)$ time per matching.

During the implementation of the thesis, I played with the idea of using this method. However, the permutations with backtracking gave better results and the results given in [4] state that their implementation takes more than one day of computation time in 32-core computer so we decided to not continue with this implementation.

5 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 three methods were used. The first two 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.

5.1 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 larger number of outputs until a given population size. This heuristic is one of the most common in the literature such as [9] and [2].

5.2 Bad positions heuristic

This heuristic is defined in [5]. 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:

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

5.3 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 π 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 $O(n! \times f(n))$ 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 Compared sizes of filter sets 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
R_k^9	914,444	607,164	274,212	94,085	25,786	5699	1107	250	73	27	8	1
S_k^9	586	570	519	413	314	230	179	123	57	24	8	1

We actually suspect that this heuristic could be complete, the networks surviving this prune create a minimum filter for the optimal size problem, which bring us to the next conjecture:

Conjecture 5.1 *Given two comparator networks C_a and C_b with n channels in which we are checking if C_a subsumes C_b. If Lemmas 3.3 and 3.4 do not discard the subsumption then C_a subsumes C_b.*

5.4 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.

Table 6 Best sizes combining 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

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.

Table 7 Best sizes combining bad positions and prune with no permutations

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	74
Best known $S(n)$	25	29	35	39	45	51	56	60

When it comes to bigger instances of the problem, we tried input sizes of 17, 18, 19 and 21 mixing prune with no permutations and remove networks with more outputs heuristics.

We were able to find a network with 19 inputs and 86 comparators, same size than [11] using a maximum population size of 50,000 networks and that would have been record in 2013. Since then another network with 85 comparators has been found.

In the following table we can see the best network sizes found during this master thesis:

Table 8 Best sizes obtained in big networks combining heuristics

inputs	17	18	19	20	21
Best found size	72	78	86	-	104
Best known size	71	77	85	91	100

When it comes to use the prune with no permutations heuristic the algorithm used the following times in finding a sorting network for values of n in the interval [7..13]. The next table shows the times spent to find a sorting network for the prune with no permutations heuristic.

Table 9 Best sizes obtained in big networks combining heuristics

inputs	7	8	9	10	11	12	13
Time in seconds	0.1	0.2	1	5	63	251	21823

As it is noticeable, the computation time scalates quickly in bigger instances. While for $n = 12$ the process finishes in the order of minutes, for $n = 13$ it already takes around 6 hours of computation. However, we believe that using this solution with the appropriate hardware and in combination with other heuristics, could find better networks for the bigger instances.

6 Conclusion

During this thesis I have studied in depth the optimal size problem for sorting networks. The main focus has been in implementing the algorithms in [8] with advanced data structures to improve running times of the algorithms and the study of heuristics to try finding better size sorting networks. The results are promising and show how the generate-and-prune algorithms can address the problem of finding better sorting networks and solve bigger instances of $S(n)$ if more powerful hardware provided.

During this months I have been able to find sorting networks that would have been size records some years ago in a modest hardware, this opens the possibility to find better sorting networks in the future. Also, if proved that the heuristic used in section 5.3 is actually complete, the solution of $S(n)$ for bigger values could be addressed.

As future work to continue this investigation, the generate-and-prune approach could be implemented using a cluster paradigm such as Spark [13]. During this thesis I did some investigation and started implementing the algorithms using Spark capabilities. However, we decided to abandon this line of work to center the thesis in the study of heuristics.

Also there is a lot of potential to find better sorting networks in the use of genetics and evolutionary algorithms such as the ones used in [11] and [9].

References

[1] Smallest and fastest sorting networks for a given number of inputs. `http://users.telenet.be/bertdobbelaere/SorterHunter/sorting_networks.html`. Accessed: 2021-05-26. [18]

[2] 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. [17]

[3] ROBERT W. FLOYD and DONALD E. KNUTH. Chapter 15 - the Bose-Nelson sorting problem. In JAGDISH N. SRIVASTAVA, editor, *A Survey of Combinatorial Theory*, pages 163–172. North-Holland, 1973. [4, 5]

[4] 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. [13, 15, 16]

[5] 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. [17]

[6] Jannis Harder. An answer to the Bose-Nelson sorting problem for 11 and 12 channels, 2021. [4, 5]

[7] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997. [4, 5]

[8] Michael Codish; Luís Cruz-Filipe; Michael Frank; Peter Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *Journal of Computer and System Sciences*, 2016. [2, 3, 4, 5, 6, 8, 10, 11, 12, 13, 21]

[9] 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. [17, 21]

[10] Takeaki Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In Hon Wai Leong, Hiroshi Imai, and Sanjay Jain, editors, *Algorithms and Computation*, pages 92–101, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. [16]

[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. [19, 21]

[12] David C. van Voorhis. Toward a lower bound for sorting networks. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 119–129. Plenum Press, New York, 1972. [5]

[13] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016. [21]