Journal of Machine Learning Research 14 (2013) 303-331 Submitted 1/12; Revised 10/12; Published 2/13

Using Symmetry and Evolutionary Search to Minimize Sorting Networks

Vinod K. Valsalam VKV@CS.UTEXAS.EDU
Risto Miikkulainen RISTO@CS.UTEXAS.EDU
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712, USA

Editor: Una-May O'Reilly

Abstract

Sorting networks are an interesting class of parallel sorting algorithms with applications in multiprocessor computers and switching networks. They are built by cascading a series of comparison-exchange units called comparators. Minimizing the number of comparators for a given number of inputs is a challenging optimization problem. This paper presents a two-pronged approach called Symmetry and Evolution based Network Sort Optimization (SENSO) that makes it possible to scale the solutions to networks with a larger number of inputs than previously possible. First, it uses the symmetry of the problem to decompose the minimization goal into subgoals that are easier to solve. Second, it minimizes the resulting greedy solutions further by using an evolutionary algorithm to learn the statistical distribution of comparators in minimal networks. The final solutions improve upon half-century of results published in patents, books, and peer-reviewed literature, demonstrating the potential of the SENSO approach for solving difficult combinatorial problems.

Keywords: symmetry, evolution, estimation of distribution algorithms, sorting networks, combinatorial optimization

1. Introduction

A sorting network of n inputs is a fixed sequence of comparison-exchange operations (comparators) that sorts all inputs of size n (Knuth, 1998). Since the same fixed sequence of comparators can sort any input, it represents an oblivious or data-independent sorting algorithm, that is, the sequence of comparisons does not depend on the input data. The resulting fixed pattern of communication makes them desirable in parallel implementations of sorting, such as those on graphics processing units (Kipfer et al., 2004). For the same reason, they are simple to implement in hardware and are useful as switching networks in multiprocessor computers (Batcher, 1968; Kannan and Ray, 2001; Baddar, 2009).

Driven by such applications, sorting networks have been the subject of active research since the 1950's (Knuth, 1998). Of particular interest are minimal-size networks that use a minimal number of comparators. Designing such networks is a hard combinatorial optimization problem, first investigated in a U.S. Patent by O'Connor and Nelson (1962) for $4 \leq n \leq 8$. Their networks had the minimal number of comparators for 4, 5, 6, and 8 inputs, but required two extra comparators for 7 inputs. This result was improved by Batcher (1968), whose algorithmic construction produces provably minimal networks for $n \leq 8$ (Knuth, 1998).

©2013 Vinod K. Valsalam and Risto Miikkulainen.

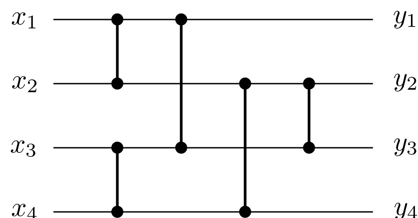


Figure 1: A 4-input sorting network. The input values $\{x_1, x_2, x_3, x_4\}$ at the left side of the horizontal lines pass through a sequence of comparison-exchange operations, represented by vertical lines connecting pairs of horizontal lines. Each such comparator sorts its two values, resulting in the horizontal lines containing the sorted output values $\{y_1 \leq y_2 \leq y_3 \leq y_4\}$ at right. This network is minimal in terms of the number of comparators. Such minimal networks are not known in general for input sizes larger than 8 and designing them is a challenging optimization problem.

Still today, provably minimal networks are known only for $n \leq 8$. Finding the minimum number of comparators for $n > 8$ is thus a challenging open problem. It has been studied by various researchers using specialized techniques, often separately for each value of n (Knuth, 1998; Koza et al., 1999). Their efforts during the last few decades have improved the size of the networks for $9 \leq n \leq 16$. For larger values of n, all best known solutions are simply merges of smaller networks; the problem is so difficult that it has not been possible to improve on these straightforward constructions (Baddar, 2009).

This paper presents a two-pronged approach to this problem, using symmetry and evolutionary search, which makes it possible to scale the problem to larger number of inputs. This approach, called Symmetry and Evolution based Network Sort Optimization (SENSO), learns the comparator distribution of minimal networks from a population of candidate solutions and improves them iteratively through evolution. Each such solution is generated by sampling comparators from the previous distribution such that the required network symmetry is built step-by-step, thereby focusing evolution on more likely candidates and making search more effective. This approach was able to discover new minimal networks for 17, 18, 19, 20, 21, and 22 inputs. Moreover, for the other $n \leq 23$, it discovered networks that have the same size as the best known networks. These results demonstrate that the approach makes the problem more tractable and suggests ways in which it can be scaled further and applied to other similarly difficult combinatorial problems.

This paper is organized as follows. Section 2 begins by describing the problem of finding minimal sorting networks in more detail and reviews previous research on solving it. Section 3 presents the SENSO approach, based on symmetry and evolution. Section 4 discusses the experimental setup for evaluating the approach and presents the results. Section 5 concludes with an analysis of the results and discussion of ways to make the approach even more effective and general in the future.

2. Background

Figure 1 illustrates a 4-input sorting network. The horizontal lines of the network receive the input values $\{x_1, x_2, x_3, x_4\}$ at left. Each vertical line represents a comparison-exchange operation that

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Batcher	**0**	**1**	**3**	**5**	**9**	**12**	**16**	**19**	26	31	37	41	48	53	59	63
Best	**0**	**1**	**3**	**5**	**9**	**12**	**16**	**19**	25	29	35	39	45	51	56	60

Table 1: The fewest number of comparators known to date for sorting networks of input sizes $n \leq 16$. These networks have been studied extensively, but the best results have been proven to be minimal only for $n \leq 8$ (shown in bold; Knuth, 1998). Such small networks are interesting because they optimize hardware resources in implementations such as multiprocessor switching networks.

takes two values and exchanges them if necessary such that the larger value is on the lower line. As a result of these comparison-exchanges, the output values appear at the right side of the network in sorted order: $\{y_1 \leq y_2 \leq y_3 \leq y_4\}$.

Sorting networks with $n \leq 16$ have been studied extensively with the goal of minimizing their sizes. The smallest sizes of such networks known to date are listed in Table 1 (Knuth, 1998). The number of comparators has been proven to be minimal only for $n \leq 8$ (Knuth, 1998). These networks can be constructed using Batcher's algorithm for odd-even merging networks (Batcher, 1968). The odd-even merge builds larger networks iteratively from smaller networks by merging two sorted lists. The odd and even indexed values of these two lists are first merged separately using small merging networks. Comparison-exchange operations are then applied to the corresponding values of the resulting small sorted lists to obtain the full sorted list.

Finding the minimum number of comparators required for $n > 8$ remains an open problem. The results in Table 1, for these values of n, improve on the number of comparators used by Batcher's method. For example, the 16-input case, for which Batcher's method requires 63 comparators, was improved by Shapiro who found a 62-comparator network in 1969. Soon afterwards, Green (1972) found a network with 60 comparators (Figure 2), which still remains the best in terms of the number of comparators.

In Green's construction, comparisons made after the first four levels (i.e., the first 32 comparators) are difficult to understand, making his method hard to generalize to larger values of n. For such values, Batcher's method can be extended with more complex merging strategies to produce significant savings in the number of comparators (Van Voorhis, 1974; Drysdale and Young, 1975). For example, the best known 256-input sorting network due to Van Voorhis requires only 3651 comparators, compared to 3839 comparators required by Batcher's method (Drysdale and Young, 1975; Knuth, 1998). Asymptotically, the methods based on merging require $O(n \log^2 n)$ comparators (Van Voorhis, 1974). In comparison, the *AKS network* by Ajtai et al. (1983) produces better upper bounds, requiring only $O(n \log n)$ comparators. However, the constants hidden in its asymptotic notation are so large that these networks are impractical. Although still not practical, Leighton and Plaxton (1990) showed that small constants are actually possible in networks that sort all but a superpolynomially small fraction of the $n!$ input permutations.

Since better algorithms are not known for constructing networks that sort all $n!$ input permutations, Batcher's or Van Voorhis' algorithms are often used in practice for large values of n, despite their non-optimality. For example, these algorithms were used to obtain the networks for $17 \leq n \leq 32$ listed in Table 2 by merging the outputs of smaller networks from Table 1 (Van Voorhis, 1971; Baddar, 2009).

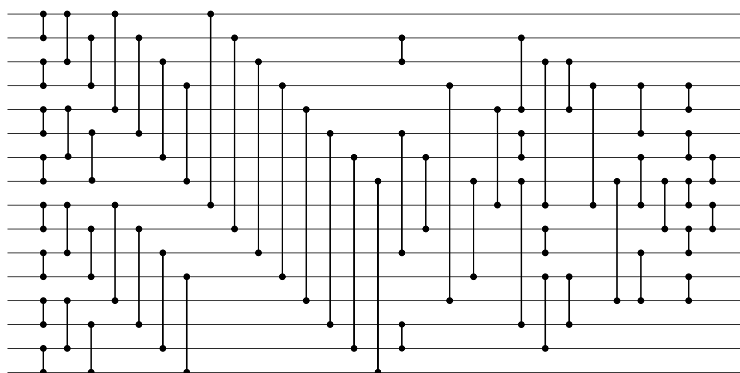


Figure 2: The 16-input sorting network found by Green. This network has 60 comparators, which is the fewest known for 16 inputs (Green, 1972; Knuth, 1998). The comparators in such hand-designed networks are often symmetrically arranged about a horizontal axis through the middle of the network. This observation has been used by some researchers to bias evolutionary search on this problem (Graham and Oppacher, 2006) and is also used as a heuristic to augment the symmetry-building approach described in Section 3.

n	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Best	73	79	88	93	103	110	118	123	133	140	150	156	166	172	180	185

Table 2: The fewest number of comparators known to date for sorting networks of input sizes $17 \leq n \leq 32$. Networks for these values of n were obtained by merging the outputs of smaller networks from Table 1 using the non-optimal Batcher's or Van Voorhis' algorithms (Van Voorhis, 1971; Baddar, 2009). The methods used to optimize networks for $n \leq 16$ are intractable for these larger values of n because of the explosive growth in the size of the search space. The approach presented in this paper mitigates this problem by constraining search to promising solutions and improves these results for input sizes 17, 18, 19, 20, 21, and 22.

The difficulty of finding such minimal sorting networks prompted researchers to attack the problem using evolutionary techniques. In one such study by Hillis (1991), a 16-input network having 61 comparators was evolved. He facilitated the evolutionary search by initializing the population with the first four levels of Green's network, so that evolution would need to discover only the remaining comparators. This (host) population of sorting networks was co-evolved with a (parasite) population of test cases that were scored based on how well they made the sorting networks fail. The purpose of the parasitic test cases is to nudge the solutions away from getting stuck on local optima.

Juillé (1995) improved on Hillis' results by evolving 16-input networks that are as good as Green's network (60 comparators), from scratch without specifying the first 32 comparators. Moreover, Juillé's method discovered 45-comparator networks for the 13-input problem, which was an improvement of one comparator over the previously known best result. His method, based on the Evolving Non-Determinism (END) model, constructs solutions incrementally as paths in a search

tree whose leaves represent valid sorting networks. The individuals in the evolving population are internal nodes of this search tree. The search proceeds in a way similar to beam search by assigning a fitness score to internal nodes and selecting nodes that are the most promising. The fitness of an internal node is estimated by constructing a path incrementally and randomly to a leaf node. This method found good networks with the same number of comparators as in Table 1 for all $9 \leq n \leq 16$.

Motivated by observations of symmetric arrangement of comparators in many sorting networks (Figure 2), Graham and Oppacher (2006) used symmetry explicitly to bias evolutionary search. They compared evolutionary runs on populations initialized randomly with either symmetric or asymmetric networks for the 10-input sorting problem. The symmetric networks were produced using symmetric comparator pairs, that is, pairs of comparators that are vertical mirror images of each other. Although evolution was allowed to disrupt the initial symmetry through variation operators, symmetric initialization resulted in higher success rates compared to asymmetric initialization. A similar heuristic is used to augment the SENSO approach discussed in this paper.

Evolutionary approaches must verify that the solution network sorts all possible inputs correctly. A naive approach is to test the network on all $n!$ permutations of n distinct numbers. A better approach requiring far fewer tests uses the *zero-one principle* (Knuth, 1998) to reduce the number of test cases to 2^n binary sequences. According to this principle, if a network with n input lines sorts all 2^n binary sequences correctly, then it will also sort any arbitrary sequence of n non-binary numbers correctly. However, the increase in the number of test cases remains exponential and is a bottleneck in fitness evaluations. Therefore, some researchers have used FPGAs to mitigate this problem by performing the fitness evaluations on a massively parallel scale (Koza et al., 1998; Korenek and Sekanina, 2005). In contrast, this paper develops a Boolean function representation of the zero-one principle for fitness evaluation, as discussed next.

3. Approach

This section presents the new SENSO approach based on symmetry and evolutionary search to minimize the number of comparators in sorting networks. It begins with a description of how the sorting network outputs can be represented as monotone Boolean functions, exposing the symmetries of the network. This representation makes it possible to decompose the problem into subgoals, which are easier to solve. Each subgoal constitutes a step in building the symmetries of the network with as few comparators as possible. The resulting greedy solutions are optimized further by using an evolutionary algorithm to learn the distribution of comparators that produce minimal networks.

3.1 Boolean Function Representation

The zero-one principle (Section 2) can be used to express the inputs of a sorting network as Boolean variables and its outputs as functions of those variables. It simplifies the sorting problem to counting the number of inputs that have the value 1 and setting that many of the lowermost outputs to 1 and the remaining outputs to 0. In particular, the function $f_i(x_i, \ldots, x_n)$ at output i takes the value 1 if and only if at least $n+1-i$ inputs are 1. That is, f_i is the disjunction of all conjunctive terms with exactly $n+1-i$ variables.

Since these functions are implemented by the comparators in the network, the problem of designing a sorting network can be restated as the problem of finding a sequence of comparators that compute its output functions. Each comparator computes the conjunction (upper line) and disjunction (lower line) of their inputs. As a result, a sequence of comparators computes Boolean functions

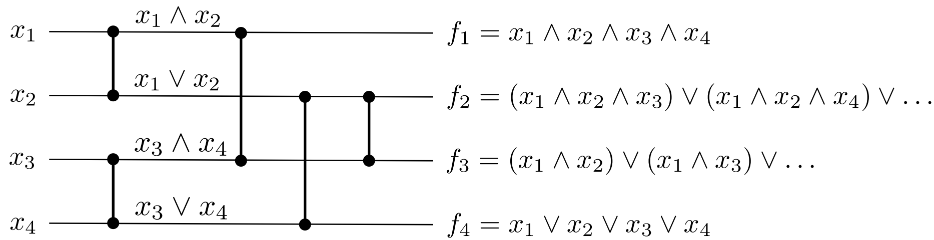
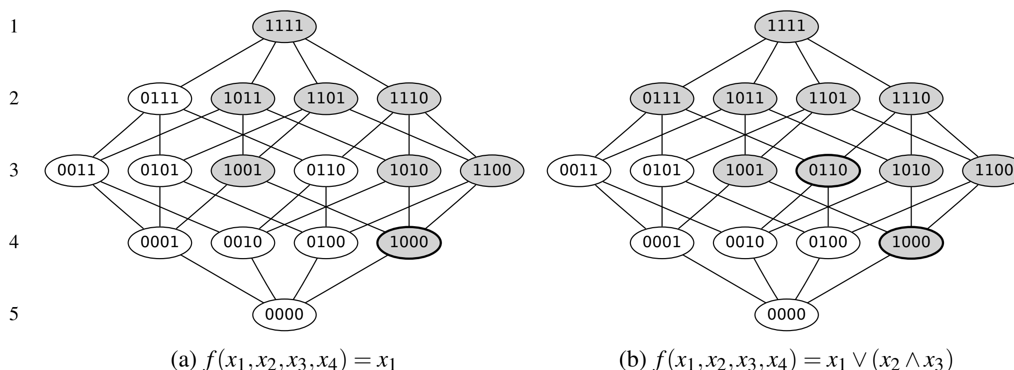


Figure 3: Boolean output functions of a 4-input sorting network. The zero-one principle can be used to represent the inputs of the network as Boolean variables. Each comparator produces the conjunction of its inputs on its upper line and their disjunction on its lower line. As a result, the functions at the outputs of the network are compositions of conjunctions and disjunctions of the input variables, that is, they are monotone Boolean functions. In particular, the output function f_i at line i is the disjunction of all conjunctive terms with exactly $n+1-i$ variables. Therefore, a sorting network is a sequence of comparators that compute all its output functions from its input variables. This representation makes it possible to express network symmetry, which turns out to be useful in constructing minimal networks.

that are compositions of conjunctions and disjunctions of the input variables (Figure 3). Since the number of terms in these functions can grow combinatorially as comparators are added, it is necessary to use a representation that makes it efficient to compute them and to determine whether all output functions have been computed.

Such functions computed using only conjunctions and disjunctions without any negations are called *monotone Boolean functions* (Korshunov, 2003). For example, the functions for the 4-input sorting network in Figure 3 are all monotone Boolean functions. Such a function f on n binary variables has the property that $f(\mathbf{a}) \leq f(\mathbf{b})$ for any distinct binary n-tuples $\mathbf{a} = a_1, \ldots, a_n$ and $\mathbf{b} = b_1, \ldots, b_n$ such that $\mathbf{a} \prec \mathbf{b}$, where $\mathbf{a} \prec \mathbf{b}$ if $a_i \leq b_i$ for $1 \leq i \leq n$. The set of all 2^n binary n-tuples ordered by $\prec$ is a partially ordered set called a *Boolean lattice*, which makes it possible to represent monotone Boolean functions conveniently. The Boolean lattice for $n = 4$ is illustrated in Figure 4 as an undirected graph (Hasse diagram) of $2^4 = 16$ nodes. Any two nodes in the lattice are *comparable* and are connected by a path if they can be ordered by $\prec$. A subset of nodes that are pair-wise incomparable is called an *antichain*. A subset X of nodes is said to be *bounded above* by the node $\mathbf{y}$ if $\mathbf{x} \prec \mathbf{y}$ for all $\mathbf{x} \in X$. The term *bounded below* is defined in a similar manner. These concepts are used to characterize monotone Boolean functions in sorting networks.

For any monotone Boolean function f, the subset of lattice nodes at which it takes the value 1 are bounded above by the topmost node in the lattice and are bounded below by an antichain of nodes corresponding to the conjunctive terms in its disjunctive normal form. That is, the nodes in this antichain form a boundary in the lattice, separating the nodes at which f takes the value 1 from those at which it takes the value 0. Therefore, it is sufficient to specify the antichain of boundary nodes to define a monotone Boolean function. Moreover, nodes in the same level i (numbered from the top of the lattice) form an antichain of boundary nodes because they all have the same number $n+1-i$ of 1s in their binary representations and are therefore incomparable. In fact, they are the boundary nodes of function f_i at output i of the sorting network since they correspond to the



(a) $f(x_1,x_2,x_3,x_4) = x_1$

(b) $f(x_1,x_2,x_3,x_4) = x_1 \vee (x_2 \wedge x_3)$

Figure 4: Representation of monotone Boolean functions on four variables in the Boolean lattice. The $2^4 = 16$ nodes of the lattice are organized in levels (numbered on the left), each containing the binary value assignments to the 4-tuple $x_1x_2x_3x_4$ with the same number of 1s. The truth table for any Boolean function $f(x_1,x_2,x_3,x_4)$ can be represented in this lattice by shading all the nodes for which f takes the value 1. Furthermore, a node $b_1b_2b_3b_4$ has a path to a lower node $a_1a_2a_3a_4$ if $a_i \leq b_i$ for $1 \leq i \leq 4$. As a result, if node $a_1a_2a_3a_4$ is shaded for a monotone function f, then all higher nodes reachable from it are also shaded, that is, f is defined completely by the nodes in the lower boundary of its shaded region. This set of nodes (shown by bold outline) corresponds to the conjunctive terms in the disjunctive normal form of f. For example, it contains just the node 1000 for $f = x_1$ and two nodes 1000 and 0110 for $f = x_1 \vee (x_2 \wedge x_3)$. This representation makes it possible to compute monotone Boolean functions more efficiently.

disjunction of all conjunctive terms with exactly $n+1-i$ variables. Thus, levels 1 to n of the lattice have a one-to-one correspondence with the output functions of the n-input network. Moreover, it is possible to efficiently determine whether f_i has been computed at output i just by verifying whether it takes the value 1 at all level i boundary nodes.

Monotone Boolean functions can thus be represented by their antichain of boundary nodes in the Boolean lattice. In a lattice of size 2^n, the maximum size of this representation is equal to the size of the longest antichain, which is only $\binom{n}{\lceil n/2 \rceil}$ nodes (by Stirling's approximation, $\binom{n}{\lceil n/2 \rceil} = \mathrm{O}\left(\frac{2^n}{\sqrt{n}}\right)$). However, computing conjunctions and disjunctions using this representation produces combinatorially more redundant, non-boundary nodes that have to be removed (Gunter et al., 1996). A more efficient representation is based on storing the values of the function in its entire truth table as a bit-vector of length 2^n. Its values are grouped according to the levels in the Boolean lattice so that values for any level can be retrieved easily. This representation also allows computing conjunctions and disjunctions efficiently as the bitwise AND and bitwise OR of the bit-vectors, respectively. Moreover, efficient algorithms for bit-counting can be used to determine if a given sorting network is valid by checking if its function at output i has the value 1 at all level i nodes for $1 \leq i \leq n$, which is the case when all output functions f_i are computed correctly.

	DNF	CNF
f_1	$x_1 \wedge x_2 \wedge x_3 \wedge x_4$	$x_1 \wedge x_2 \wedge x_3 \wedge x_4$
f_2	$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee \ldots$	$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge \ldots$
f_3	$(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee \ldots$	$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge \ldots$
f_4	$x_1 \vee x_2 \vee x_3 \vee x_4$	$x_1 \vee x_2 \vee x_3 \vee x_4$

Table 3: Symmetries of the 4-input sorting network in terms of its output functions. Writing the Boolean output functions of the sorting network in both the disjunctive normal form (DNF) and in the conjunctive normal form (CNF) is a good way to visualize the symmetries of the output functions. For example, swapping the conjunctions $\wedge$ and disjunctions $\vee$ in the DNF form of either function f_2 or f_3 yields the CNF form of the other function. Therefore, for the operation of swapping $\wedge$ and $\vee$ in both f_2 and f_3 and also swapping their row positions in the table, the resulting table of functions remains the same as the original table. Moreover, this assertion holds for any pair of functions f_i and f_{5-i}, not just for f_2 and f_3. Such an operation that preserves the output functions of the network is called a symmetry. These symmetries can be used to minimize the number of comparators in the network.

Finding a minimum sequence of comparators that computes all the output functions is a challenging combinatorial problem. It can be made more tractable by using the symmetries of the network, represented in terms of the symmetries of its output functions, as will be described next.

3.2 Sorting Network Symmetries

A sorting network *symmetry* is an operation on the ordered set of network output functions that leaves the functions invariant, that is, the resulting network outputs remain unchanged. For example, swapping the outputs of all comparators of a network to reverse its sorting order and then flipping the network vertically to restore its original sorting order is a symmetry operation. Swapping the comparator outputs swaps the conjunctions $\wedge$ and disjunctions $\vee$ in the output functions. The resulting reversal of the network sorting order can be expressed as $f_i(x_i, \ldots, x_n, \wedge, \vee) = f_{n+1-i}(x_i, \ldots, x_n, \vee, \wedge)$ for all $1 \leq i \leq n$, that is, the output function f_{n+1-i} can be obtained from f_i by swapping its $\wedge$ and $\vee$, and vice versa. Therefore, in addition to swapping $\wedge$ and $\vee$, if the *dual* functions f_i and f_{n+1-i} are also swapped, then the network outputs remain the same. This type of symmetry is illustrated in Table 3 for the 4-input sorting network.

It is thus possible to define symmetry operations σ_i for $1 \leq i \leq \lceil \frac{n}{2} \rceil$ that act on the ordered set of network output functions by swapping the function f_i and its dual f_{n+1-i} and swapping their $\wedge$ and $\vee$. The compositions of these symmetry operations are also symmetries because σ_i and σ_j operate independently on different pairs of output functions. That is, this set of operations are closed under composition, and they are associative. Moreover, each operation is its own inverse, producing the identity when applied twice in a row. Thus they satisfy all the axioms of a *group* for representing symmetries mathematically. Since every element of this group can be expressed as the composition of finitely many elements of the set $\Sigma = \{\sigma_1, \ldots, \sigma_{\lceil \frac{n}{2} \rceil}\}$, the group is said to be *generated* by Σ and is denoted $\langle \Sigma \rangle$.

Similarly, the *subgroups* of $\langle\Sigma\rangle$, that is, subsets that satisfy the group axioms, can be used to represent the symmetries of partial networks created in the process of constructing a full sorting network. In particular, computing pairs of dual output functions produces symmetries corresponding to a subgroup of $\langle\Sigma\rangle$ (Figure 5). Since each symmetry element in Σ operates on disjoint pairs of dual functions, any such subgroup can be written as $\langle\Gamma\rangle$, where Γ is a subset of Σ.

Initially, before any comparators have been added, each line i in the network has the trivial monotone Boolean function x_i. As a result, the network does not have any symmetries, that is, $\Gamma = \{\}$. Adding comparators to compute the output function f_i and its dual f_{n+1-i} yields $\Gamma = \{\sigma_i\}$ for the resulting partial network. Adding more comparators to compute both f_j and its dual f_{n+1-j} creates a new partial network with $\Gamma = \{\sigma_i, \sigma_j\}$, that is, the new partial network is more symmetric. Continuing to add comparators until all output functions have been constructed produces a complete sorting network with $\Gamma = \Sigma$.

Thus adding comparators to the network in a particular sequence builds its symmetry in a corresponding sequence of increasingly larger subgroups. Conversely, building symmetry in a particular sequence constrains the comparator sequences that are possible. Symmetry can therefore be used to constrain the search space for designing networks with desired properties. In particular, a sequence of subgroups can represent a sequence of subgoals for minimizing the number of comparators in the network. Each subgoal in this sequence is defined as the subgroup that can be produced from the previous subgoal by adding the fewest number of comparators.

Applying this heuristic to the initial network with symmetry $\Gamma = \{\}$, the first subgoal is defined as the symmetry that can be produced from the input variables by computing a pair of dual output functions with the fewest number of comparators (Figure 6). The functions $f_1 = x_1 \wedge \ldots \wedge x_n$ and $f_n = x_1 \vee \ldots \vee x_n$ have the fewest number of variable combinations and can therefore be computed by adding fewer comparators than any other pair of dual output functions. Thus the first subgoal is to produce the symmetry $\Gamma = \{\sigma_1\}$ using as few comparators as possible.

After computing f_1 and f_n, the next pair of dual output functions with the fewest number of variable combinations are f_2 and f_{n-1}. Therefore, the second subgoal is to compute them and produce the symmetry $\Gamma = \{\sigma_1, \sigma_2\}$. In this way, the number of variable combinations in the output functions continues to increase from the outer lines to the middle lines of the network. Therefore, from any subgoal that adds the symmetry σ_k to Γ, the next subgoal adds the symmetry σ_{k+1} to Γ. This sequence of subgoals continues until all the output functions are computed, producing the final goal symmetry $\Gamma = \{\sigma_1, \ldots, \sigma_{\lceil \frac{n}{2} \rceil}\}$.

Although this subgoal sequence specifies the order in which to compute the output functions, it does not specify an optimal combination of comparators for each subgoal. However, it is easier to minimize the number of comparators required for each subgoal than for the entire network, as will be described next.

3.3 Minimizing Comparator Requirement

In order to reach the first subgoal, the same comparator can compute a conjunction for f_1 and also a disjunction for f_n simultaneously (Figure 6). Sharing the same comparator to compute dual functions in this manner reduces the number of comparators required in the network. However, such sharing between dual functions of the same subgoal is possible only in some cases. In other cases, it may still be possible to share a comparator with the dual function of a later subgoal. Thus,

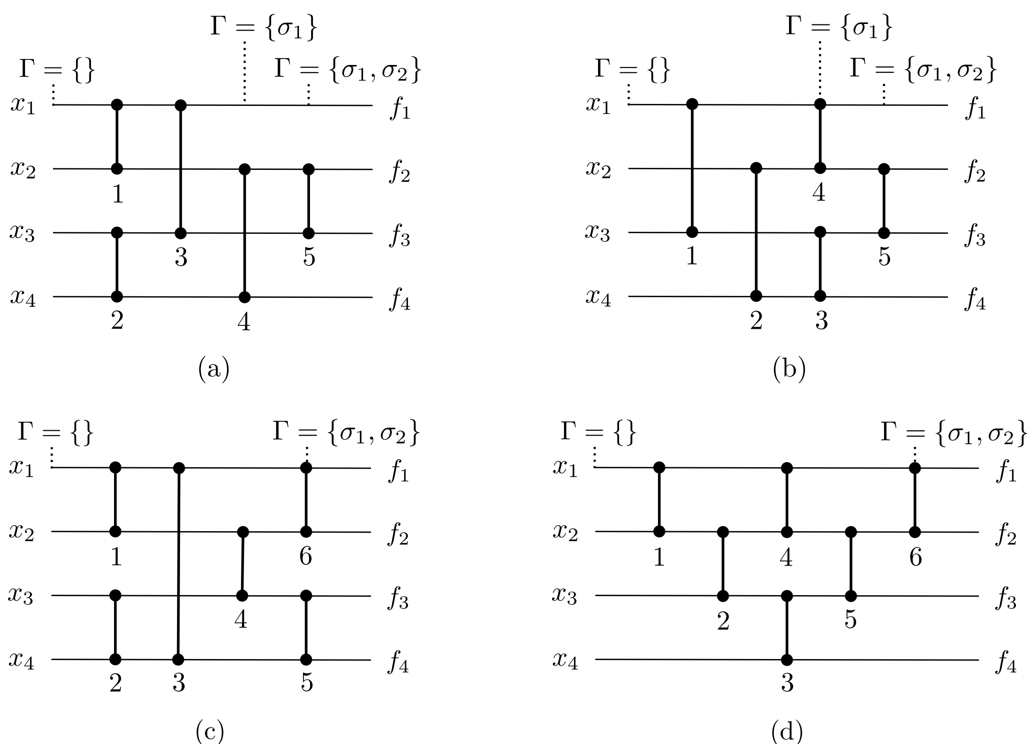


Figure 5: Symmetries of 4-input sorting networks. The numbers below the comparators indicate the sequence in which the comparators are added during network construction. The last comparator touching horizontal line i completes computing the output function f_i for that line. Functions f_i and f_{n+1-i} form a dual, and computing them both gives the network the symmetry σ_i. In network (a), adding comparator 3 completes computing f_1 and when comparator 4 is added to complete computing its dual f_4, the network gets the symmetry σ_1. Adding comparator 5 then completes computing both f_2 and its dual f_3, giving the network its second symmetry σ_2. Network (b) also produces the same sequence of symmetries and has the same number of comparators. In network (c), adding comparator 5 completes computing both f_3 and f_4, but not their duals f_1 and f_2. Only when comparator 6 is added to complete computing f_1 and f_2 does it get both its symmetries σ_1 and σ_2. Network (d) is similar to (c), and they both require one more comparator than networks (a) and (b). Thus the sequence in which the comparators are added determines the sequence in which the network gets its symmetries. Conversely, a preferred sequence of symmetries can be specified to constrain the sequence in which comparators are added and to minimize the number of comparators required.

minimizing the number of comparators requires determining which comparators can be shared and then adding those comparators that maximize sharing.

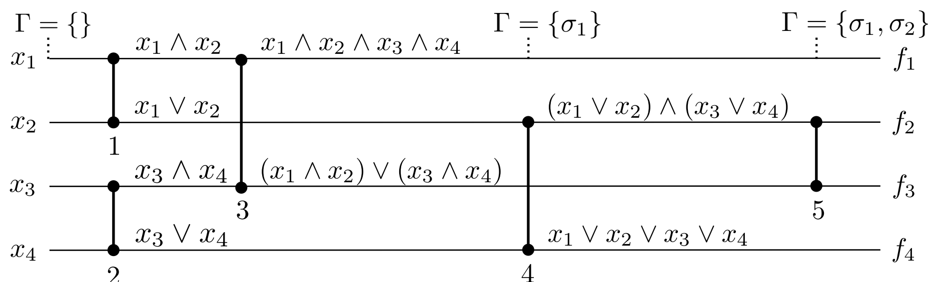
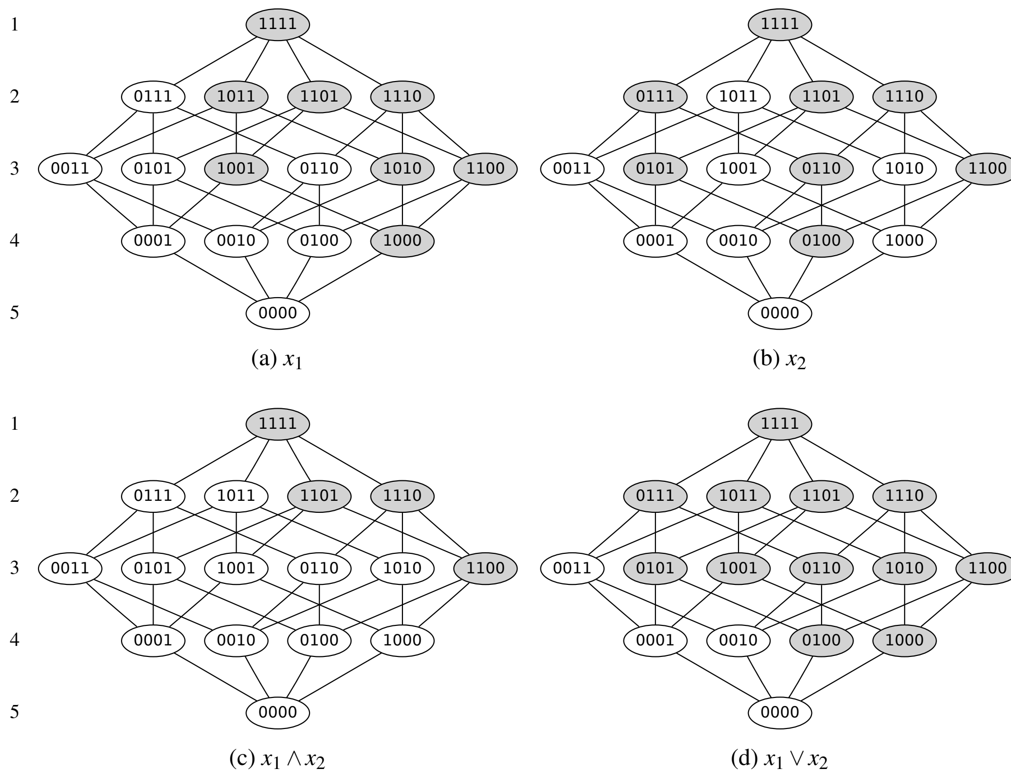


Figure 6: Subgoals for constructing a minimal 4-input sorting network. The final goal is to produce the symmetry $\Gamma = \{\sigma_1, \sigma_2\}$ by computing all four output functions f_i while using the minimum number of comparators. This goal can be decomposed into a sequence of subgoals specified as subgroups of the final symmetry group $\langle\Gamma\rangle$. At any stage in the construction, the next subgoal is the subgroup that can be produced by adding the fewest number of comparators. Initially, the network does not have any symmetries, that is, $\Gamma = \{\}$. The dual functions f_1 and f_4 are the easiest to compute, having fewer variable combinations and therefore requiring fewer comparators than f_2 and f_3. Hence the first subgoal is to produce the symmetry $\Gamma = \{\sigma_1\}$. Notice that comparators 1 and 2 compute parts of both f_1 and f_4 to achieve this subgoal with the minimum number of comparators. The second subgoal is to produce the symmetry $\Gamma = \{\sigma_1, \sigma_2\}$ by computing the functions f_2 and f_3. Adding comparator 5 completes this subgoal since comparators 3 and 4 have already computed f_2 and f_3 partially. Optimizing the number of comparators required to reach each subgoal separately in this way makes it possible to scale the approach to networks with more inputs.

The Boolean lattice representation of functions discussed in Section 3.1 can be used to determine whether or not sharing a comparator for computing parts of two functions simultaneously is possible (Figure 7). Assume that the current subgoal is to compute the output function f_i and its dual f_{n+1-i}. That is, functions for outputs less than i and greater than $n+1-i$ have already been fully computed, implying that each of these functions f_j has the value 1 at all nodes in levels less than or equal to j and the value 0 everywhere else. Moreover, the functions for the remaining outputs have been partially computed. In particular, each of these intermediate functions are guaranteed to have the value 1 at all nodes in levels less than or equal to i and the value 0 at all nodes in levels greater than $n+1-i$. If that was not the case, it will be impossible to compute at least one of the remaining output functions by adding more comparators since conjunctions preserve 0s and disjunctions preserve 1s of the intermediate functions they combine.

The current subgoal of computing function f_i requires setting its value at all nodes in level i to 1 and its value at all nodes in level $i+1$ to 0, thus defining its node boundary in the lattice. Its monotonicity then implies that it has the value 1 at all nodes in levels less than i and the value 0 at all nodes in levels greater than $i+1$. Moreover, since the intermediate functions f'_j on lines $i \leq j \leq n+1-i$ already have the value 1 at all nodes in levels less than or equal to i, computing f_i from them will retain that value at those nodes automatically. Therefore, f_i can be computed just by setting its value at all nodes in level $i+1$ to 0.



(a) x_1 (b) x_2

(c) $x_1 \wedge x_2$ (d) $x_1 \vee x_2$

Figure 7: Comparator sharing to compute dual output functions in a 4-input sorting network. This figure illustrates the Boolean lattice representation of the functions computed by comparator 1 in Figure 6. The levels of the lattices are numbered on the left and the nodes at which the function takes the value 1 are shaded. Comparator 1 computes the conjunction (c) and the disjunction (d) of the functions (a) and (b) for the subgoal of computing the output functions $f_1 = x_1 \wedge x_2 \wedge x_3 \wedge x_4$ and $f_4 = x_1 \vee x_2 \vee x_3 \vee x_4$. Function f_1 can be computed by using conjunctions to set its value at all nodes in level 2 of the lattice to 0. Similarly, f_4 can be computed by using disjunctions to set its value at all nodes in level 4 to 1. Thus, comparator 1 contributes to computing both f_1 and f_4 by setting the values at two nodes in level 2 of its conjunction to 0 and the values at two nodes in level 4 of its disjunction to 1. Sharing comparators in this manner reduces the number of comparators required to construct the sorting network.

The value of f_i at a node in level $i+1$ can be set to 0 by adding a comparator that computes its conjunction with another function that already has the value 0 at that node, thus increasing the number of 0-valued nodes. The disjunction that this comparator also computes has fewer 0-valued nodes than either of its input functions and is therefore not useful for computing f_i. However, the disjunction will be used to compute the other remaining output functions, implying that it has the value 1 at all nodes in level $i+1$ as required by those functions. Since the disjunction does not have any 0-valued nodes in level $i+1$, its inputs do not have any common 0-valued nodes in that level. That is, exactly one of the intermediate functions f'_j has the value 0 for any particular node in level

$i+1$. Adding a comparator between a pair of such functions collects the 0-valued nodes from both functions in their conjunction. Repeating this process recursively collects the 0-valued nodes in level $i+1$ from all functions to the function on line i, thus producing f_i. Similarly, its dual function f_{n+1-i} can be computed from the functions f'_j by using disjunctions instead of conjunctions to set its values at all nodes in level $n+1-i$ to 1.

The leaves of the resulting binary recursion tree for f_i are the functions f'_j that have 0-valued nodes in level $i+1$ and its internal nodes are the conjunctive comparator outputs. Since the number of nodes of degree two in a binary tree is one less than the number of leaves (Mehta and Sahni, 2005), the number of comparators required depends only on the number of functions with which the recursion starts, that is, it is invariant to the order in which the recursion pairs the leaves. However, the recursion trees for f_i and f_{n+1-i} may have common leaves, making it possible to use the same comparator to compute a conjunction for f_i and a disjunction for f_{n+1-i}. Maximizing such sharing of comparators between the two recursion trees minimizes the number of comparators required for the current subgoal.

It may also be possible to share a comparator with a later subgoal, for example, when it computes a conjunction for f_i and a disjunction for f_{n+1-k}, where $i < k \leq \lceil \frac{n}{2} \rceil$. In order to prioritize subgoals and determine which comparators maximize sharing, each pair of lines where a comparator can potentially be added is assigned a utility. Comparators that contribute to both f_i and f_{n+1-i} for the current subgoal get the highest utility. Comparators that contribute to an output function for the current subgoal and an output function for the next subgoal get the next highest utility. Similarly, other comparators are also assigned utilities based on the output functions to which they contribute and the subgoals to which those output functions belong. Many comparators may have the same highest utility; therefore, one comparator is chosen randomly from that set and it is added to the network. Repeating this process produces a sequence of comparators that optimizes sharing within the current subgoal and between the current subgoal and later subgoals.

Optimizing for each subgoal separately in this manner constitutes a greedy algorithm that produces minimal-size networks with high probability for $n \leq 8$. However, for larger values of n, the search space is too large for this greedy approach to find a global optimum reliably. In such cases, stochastic search such as evolution can be used to explore the neighborhood of the greedy solutions for further optimization, as will be described next.

3.4 Evolving Minimal-Size Networks

The most straightforward approach is to initialize evolution with a population of solutions that the greedy algorithm produces. The fitness of each solution is the negative of its number of comparators so that improving fitness will minimize the number of comparators. In each generation, two-way tournament selection based on this fitness measure is used to select the best individuals in the population for reproduction. Reproduction mutates the parent network, creating an offspring network in two steps: (1) a comparator is chosen from the network randomly and the network is truncated after it, discarding all later comparators, and (2) the greedy algorithm is used to add comparators again, reconstructing a new offspring network. Since the greedy algorithm chooses a comparator with the highest utility randomly, this mutation explores a new combination of comparators that might be more optimal than the parent.

This straightforward approach restricts the search to the space of comparator combinations suggested by the greedy algorithm and assumes that it contains a globally minimal network. In some

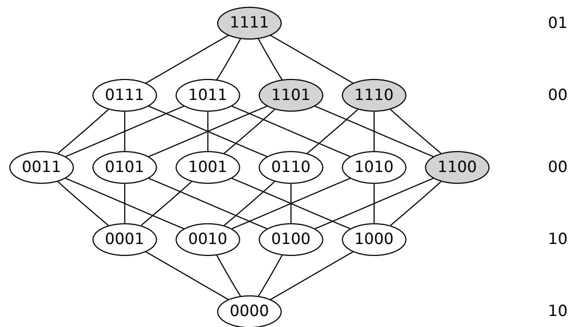


Figure 8: State representation of the function $x_1 \wedge x_2$ used in the EDA. The state (shown on the right) is a bit-string with two bits for each level of the Boolean lattice. The first bit is 1 only if the value of the function for all nodes in that level is 0 and the second bit is 1 only if its value for all nodes in that level is 1. This condensed representation of the function is based on the information used by the symmetry-building greedy algorithm and it is therefore useful for constructing minimal-size sorting networks.

cases, however, the globally minimal networks may use comparators that are different from those suggested by the greedy algorithm. Therefore, a more powerful (but still brute force) approach is to let evolution use such comparators as well: with a probability determined empirically, the suggestions of the greedy algorithm are ignored and instead the next comparator to be added to the network is selected randomly from the set of all potential comparators.

A more effective way to combine evolution with such departures from the greedy algorithm is to use an Estimation of Distribution Algorithm (EDA) (Bengoetxea et al., 2001; Alden, 2007; Mühlenbein and Höns, 2005). The idea is to estimate the probability distribution of comparator combinations in the smallest networks evolved thus far and to use this distribution to generate comparator suggestions for the next generation. The EDA is initialized as before with a population of networks generated by the greedy algorithm. In each generation, a set of networks with the highest fitness are selected from the population. These networks are used in three ways: (1) to estimate the distribution of comparators for a generative model of small networks, (2) as elite networks, passed unmodified to the next generation, and (3) as parent networks, from which new offspring networks are created for the next generation.

The generative model of the EDA specifies the probability $P(C|S)$ of adding a comparator C to an n-input partial network with state S. The state of a partial network is defined in terms of the n Boolean functions that its comparators compute. These functions determine the remaining comparators that are needed to finish computing the output functions, making them a good representation of the partial network. However, storing the state as the concatenation of the n functions is computationally intractable since each function is represented as a vector of 2^n bits. Therefore, a condensed state representation is computed based on the observation that the greedy algorithm does not use the actual function values for the nodes in the Boolean lattice; it only checks whether the values in a given level are all 0s or all 1s. This information, encoded as $2(n+1)$ bits (Figure 8), is suitable as the state representation for the model as well.

Since the model is estimated from the set of the smallest networks in the population, it is likely to generate small networks as well. Although it can generate new networks from scratch, it is used as part of the above reproduction mechanism to reconstruct a new offspring network from the truncated parent network, that is, it is used in Step 2 of reproduction instead of the greedy algorithm. In this step, some comparators are also chosen randomly from the set of all potential comparators to encourage exploration of comparator combinations outside the model. Moreover, if the model does not generate any comparators for the current state, then the reconstruction step falls back to the greedy algorithm for adding a comparator.

As discussed in Section 3.3, the greedy algorithm chooses the comparator to be added to the network randomly from those that have the highest utility (Variant 1). This random choice can be modified slightly to prefer comparators that are symmetric with respect to another comparator that is already in the network (Variant 2). Doing so makes the arrangement of comparators more bilaterally symmetric about a horizontal axis through the middle of the network. This heuristic was motivated by Graham and Oppacher (2006), who found that biasing evolutionary search using such symmetric comparator pairs was beneficial. The EDA works well with both of these variants of the greedy algorithm, learning to find smaller sorting networks than previous results, as demonstrated next.

4. Results

SENSO was run with a population size of 200 for 500 generations to evolve minimal-size networks for different input sizes. In each generation, the top half of the population (i.e., 100 networks with the fewest number of comparators) was selected for estimating the model. The same set of networks was copied to the next generation without modification. Each of them also produced an offspring network to replace those in the bottom half of the population. A Gaussian probability distribution was used to select the comparator from which to truncate the parent network. This Gaussian distribution was centered at the middle of its comparator sequence with a standard deviation of one-fourth of its number of comparators. As a result, parent networks were more likely to be truncated near the middle than near the ends. When reconstructing the truncated network, the next comparator to be added to the network was generated either by the estimated model (with probability 0.5) or was selected randomly from the set of all potential comparators (with probability 0.5). Results were insensitive to small changes in these probabilities. The SENSO source code to run this experiment is available from the website `http://nn.cs.utexas.edu/?sorting-code`.

The above experiment was repeated 20 times for each variant of the greedy algorithm and for each input size $n \leq 23$, each time with a different random number seed. The smallest network found in each set of 20 runs was recorded as the result for that particular combination of algorithmic variant and input size. This procedure was repeated 25 times for each set of 20 runs to determine which of the two variants produced smaller networks. According to the Mann-Whitney U-test, the median number of comparators in the smallest networks found by variant 2 was significantly fewer for input sizes 13, 15, 18, 20, 22 ($p < 0.02$, one-tailed). There was no significant difference between the two variants for the other input sizes. That is, the symmetry heuristic used in variant 2 makes it better or as good as variant 1 for finding small networks.

The fewest number of comparators found for each input size is listed in Table 4. For input sizes $n \leq 11$, the initial population of SENSO already contained networks with the smallest-known sizes, that is, the greedy algorithm was sufficient to find the smallest-known networks. For input sizes 12

n	12	13	14	15	16	17	18	19	20	21	22	23
Previous best	Hand-design and END					Batcher's and Van Voorhis' merge						
	39	45	51	56	60	73	79	88	93	103	110	118
SENSO	*39*	*45*	*51*	*56*	*60*	**71**	**78**	**86**	**92**	**102**	**108**	*118*

Table 4: Sizes of the smallest networks for different input sizes found by SENSO. For input sizes $n \leq 11$, networks with the smallest-known sizes (Section 2) were already found in the initial population of SENSO, that is, the greedy algorithm using symmetry was sufficient. These sizes are therefore omitted from this table. For larger input sizes, evolution found networks that matched previous best results (indicated in *italics*) or improved them (indicated in **bold**). Appendix A lists examples of these networks. These results demonstrate that the SENSO approach is effective at designing minimal-size sorting networks. Prospects of extending these results to input sizes greater than 23 will be discussed in Section 5.

to 16, and 23, SENSO evolved networks that have the same size as the best known networks. For 15 inputs, networks matching previous best results were obtained indirectly by removing the bottom line of the evolved 16-input networks and all comparators touching that line (Knuth, 1998). Most importantly, SENSO improved the previous best results for input sizes 17, 18, 19, 20, 21, and 22 by one or two comparators. Examples of these networks are listed in Appendix A.

For 23 inputs, SENSO required about 4GB of memory and 46 hours to complete 500 generations on a Xeon X5440 processor running at 2.83GHz. These requirements approximately double for every unit increase in the number of inputs due to the $O(2^n)$ complexity of the algorithm. Prospects for mitigating the effects of this exponential growth and for extending the results to $n > 23$, including to larger power-of-two networks, will be discussed in Section 5.

The previous best results for input sizes 12 through 16 were obtained either by hand design or by the END evolutionary algorithm (Knuth, 1998; Juillé, 1995; Van Voorhis, 1971; Baddar, 2009). The END algorithm improved a 25-year old result for the 13-input case by one comparator and matched the best known results for other input sizes up to 16. However, it is a massively parallel search algorithm, requiring very large computational resources, for example, a population size of 65,536 on 4096 processors to find minimal-size networks for 13 and 16 inputs (Table 5). In contrast, the SENSO approach finds such networks with much less resources (e.g., population size of 200 on a single processor in a similar number of generations), making it promising for larger problems, as will be discussed in the next section.

5. Discussion and Future Work

Previous results on designing minimal-size networks automatically by search have been limited to small input sizes ($n \leq 16$) because the number of valid sorting networks near the optimal size is very small compared to the combinatorially large space that has to be searched (Juillé, 1995). The symmetry-building approach presented in Section 3 mitigates this problem by using symmetry to focus the search on the space of networks near the optimal size. As a result, it was possible to search for minimal-size networks with more inputs ($n \leq 23$), improving the previous best results in five cases.

	END	SENSO (variant 2)
Processor family	MasPar MP-2	Xeon X5440
Number of processors used	4,096 @ 17Gop/s	1 @ 2.83GHz
Memory used	unknown	37MB
Population size	65,536	200
Runs that produced 60 comparators	2 out of 3	18 out of 20
Number of generations	300 to 500	500
Execution time for each run	48 to 72 hours	15 min

Table 5: Performance metrics of END and SENSO for the 16-input problem. This table compares the performance of SENSO with the END algorithm (Juillé, 1995) for finding 16-input networks with 60 comparators, which is the best known result for that input size. In contrast to the massively parallel END algorithm, SENSO finds such networks using much less computational resources.

These improvements can be transferred to larger values of n by using Batcher's or Van Voorhis' merge to construct such larger networks from the improved smaller networks (Knuth, 1998). The resulting networks accumulate the combined improvement of the smaller networks. For example, since the 22-input network has been improved by two comparators, two copies of it can be merged to construct a 44-input network with four fewer comparators than previous results. This merging procedure can be repeated to construct even larger networks, doubling the improvement in each step. Such networks are useful in massively parallel applications such as sorting in GPUs with hundreds of cores (Kipfer and Westermann, 2005).

It should be possible to improve these results further by extending SENSO in the following ways. First, the greedy algorithm for adding comparators can be improved by evaluating the sharing utility of groups of one or more comparators instead of single comparators. Such groups that have the highest average utility will then be preferred.

Second, the greedy algorithm can be made less greedy by considering the impact of current comparator choices on the number of comparators that will be required for later subgoals. This analysis will make it possible to optimize across subgoals, potentially producing smaller networks at the cost of additional computations.

Third, the state representation that the EDA algorithm uses contains only sparse information about the functions computed by the comparators. Extending it to include more relevant information should make it possible for the EDA to disambiguate overlapping states and therefore to model comparator distribution more accurately.

Fourth, in some cases, good n-input networks can be obtained from $n+1$-input networks by simply removing its bottom line and all comparators touching that line (Knuth, 1998), as was done in the 15-input case in this paper. This observation suggests that a potentially more powerful approach is to augment the information contained in the state representation of the EDA with the comparator distribution for multiple input sizes.

Fifth, the EDA generates comparators to add to the network only if the state of the network matches a state in the generative model exactly. Making this match graded, based on some similarity measure, may produce better results by exploring similar states when an exact match is not found.

Sixth, evolutionary search can be parallelized, for example, using the massively parallel END algorithm that has been shown to evolve the best known network sizes for $n \leq 16$ (Juillé, 1995). Conversely, using the symmetry-building approach to constrain the search space should make it possible to run the END algorithm on networks with more inputs.

Seventh, the symmetry-building approach itself can be improved. For example, it uses only the symmetries resulting from the duality of the output functions. It may be possible to extend this approach by also using the symmetries resulting from the permutations of the input variables.

Eighth, large networks can be constructed from smaller networks by merging the outputs of the smaller networks. Since smaller networks are easier to optimize, they can be evolved first and then merged by continuing evolution to add more comparators. This approach is similar to constructing minimal networks for $n > 16$ by merging smaller networks (Batcher, 1968; Van Voorhis, 1974).

In addition to finding minimal-size networks, the SENSO approach can also be used to find minimal-delay networks. Instead of minimizing the number of comparators, it would now minimize the number of parallel steps into which the comparators are grouped. Both these objectives can be optimized simultaneously as well, either by preferring one objective over the other in the fitness function or by using a multi-objective optimization algorithm such as NSGA-II (Deb et al., 2000).

Moreover, this approach can potentially be extended to design comparator networks for other related problems such as rank-order filters (Chakrabarti and Wang, 1994; Hiasat and Hasan, 2003; Chung and Lin, 1997). A rank order filter with rank r selects the rth largest element from an input set of n elements. Such filters are widely used in image and signal processing applications, for example, to reduce high-frequency noise while preserving edge information. Since these filters are often implemented in hardware, minimizing their comparator requirement is necessary to minimize their chip area. More generally, similar symmetry-based approaches may be useful for designing stack filters, that is, circuits implementing monotone Boolean functions, which are also popular in signal processing applications (Hiasat and Hasan, 2003; Shmulevich et al., 1995). Furthermore, such approaches can potentially be used to design rearrangeable networks for switching applications (Seo et al., 1993; Yeh and Feng, 1992).

6. Conclusion

Minimizing the number of comparators in a sorting network is a challenging optimization problem. This paper presented an approach called SENSO that simplifies it by converting it into the problem of building the symmetry of the network optimally. The resulting structure makes it possible to construct the network in steps and to minimize the number of comparators required for each step separately. However, the networks constructed in this manner may be sub-optimal greedy solutions, and they are optimized further by an evolutionary algorithm that learns to anticipate the distribution of comparators in minimal networks. This approach focuses the solutions on promising regions of the search space, thus finding smaller networks more effectively than previous methods.

Acknowledgments

We would like to thank Greg Plaxton for useful suggestions on formalizing the problem. This research was supported in part by the National Science Foundation under grants IIS-0915038, IIS-

0757479, and EIA-0303609; the Texas Higher Education Coordinating Board under grant 003658-0036-2007; and the College of Natural Sciences.

Appendix A. Evolved Minimal-Size Sorting Networks

This appendix lists examples of minimal-size sorting networks evolved by SENSO. For each example, the sequence of comparators is illustrated in a figure and also listed as pairs of horizontal lines numbered from top to bottom.

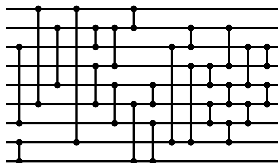


Figure 9: Evolved 9-input network with 25 comparators: [3, 7], [1, 6], [2, 5], [8, 9], [1, 8], [2, 3], [4, 6], [5, 7], [6, 9], [2, 4], [7, 9], [1, 2], [5, 6], [3, 8], [4, 8], [4, 5], [6, 7], [2, 3], [2, 4], [7, 8], [5, 6], [3, 5], [6, 7], [3, 4], [5, 6].

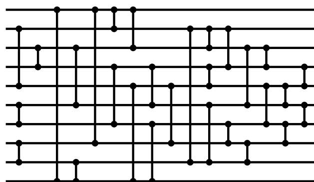


Figure 10: Evolved 10-input network with 29 comparators: [2, 5], [8, 9], [3, 4], [6, 7], [1, 10], [3, 6], [1, 8], [9, 10], [4, 7], [5, 10], [1, 2], [1, 3], [7, 10], [4, 6], [5, 8], [2, 9], [4, 5], [6, 9], [7, 8], [2, 3], [8, 9], [2, 4], [3, 6], [5, 7], [3, 4], [7, 8], [5, 6], [4, 5], [6, 7].

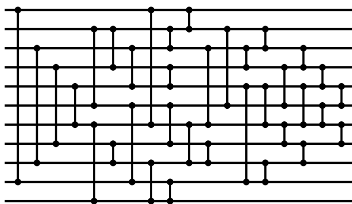


Figure 11: Evolved 11-input network with 35 comparators: [1, 10], [3, 9], [4, 8], [5, 7], [2, 6], [2, 4], [3, 5], [7, 11], [8, 9], [6, 10], [1, 7], [2, 3], [9, 11], [10, 11], [1, 2], [6, 8], [4, 5], [7, 9], [3, 7], [2, 6], [8, 9], [5, 10], [3, 4], [9, 10], [2, 3], [5, 7], [4, 6], [7, 8], [8, 9], [3, 4], [5, 7], [6, 7], [4, 5], [7, 8], [5, 6].

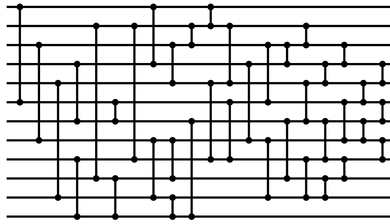


Figure 12: Evolved 12-input network with 39 comparators: [1, 6], [3, 8], [5, 11], [4, 7], [9, 12], [2, 10], [6, 7], [2, 9], [1, 4], [3, 5], [10, 12], [8, 11], [8, 10], [11, 12], [2, 3], [7, 12], [1, 2], [5, 9], [6, 9], [2, 5], [4, 8], [3, 6], [8, 11], [7, 10], [3, 4], [5, 7], [9, 11], [2, 3], [10, 11], [7, 9], [4, 5], [9, 10], [3, 4], [6, 8], [5, 6], [7, 8], [8, 9], [6, 7], [4, 5].

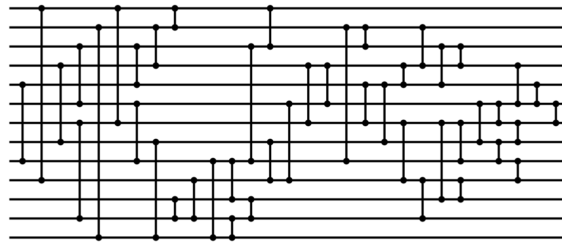


Figure 13: Evolved 13-input network with 45 comparators: [5, 9], [1, 10], [4, 8], [3, 6], [7, 12], [2, 13], [1, 7], [3, 5], [6, 9], [8, 13], [2, 4], [11, 12], [10, 12], [1, 2], [9, 13], [9, 11], [3, 9], [12, 13], [1, 3], [8, 10], [6, 10], [4, 7], [4, 6], [2, 9], [5, 7], [5, 8], [11, 12], [7, 10], [4, 5], [2, 3], [10, 12], [2, 4], [7, 11], [3, 5], [3, 4], [10, 11], [7, 9], [6, 8], [6, 7], [8, 9], [4, 6], [9, 10], [5, 6], [7, 8], [6, 7].

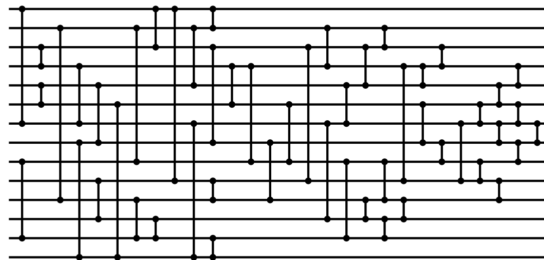


Figure 14: Evolved 14-input network with 51 comparators: [1, 7], [3, 4], [9, 13], [5, 6], [2, 11], [8, 14], [10, 12], [4, 7], [5, 8], [6, 14], [2, 9], [11, 13], [1, 3], [12, 13], [1, 10], [2, 5], [7, 14], [13, 14], [1, 2], [3, 8], [4, 6], [10, 11], [4, 9], [8, 11], [6, 9], [3, 10], [7, 12], [5, 7], [9, 13], [2, 4], [11, 12], [3, 5], [12, 13], [2, 3], [9, 11], [4, 10], [4, 5], [3, 4], [11, 12], [6, 8], [8, 9], [7, 10], [6, 7], [5, 6], [9, 10], [7, 8], [10, 11], [4, 5], [6, 7], [8, 9], [7, 8].

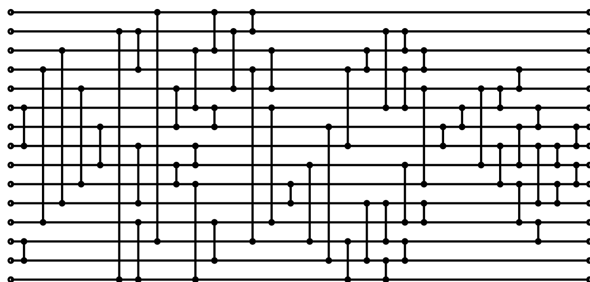


Figure 15: Evolved 15-input network with 56 comparators: [13, 14], [6, 8], [4, 12], [3, 11], [5, 10], [7, 9], [2, 15], [12, 15], [2, 4], [8, 11], [1, 13], [5, 7], [3, 6], [9, 10], [1, 3], [10, 15], [2, 5], [1, 2], [6, 7], [8, 9], [12, 14], [4, 13], [6, 12], [10, 11], [9, 13], [3, 5], [7, 14], [4, 8], [3, 4], [13, 15], [11, 14], [2, 6], [14, 15], [2, 3], [4, 6], [11, 13], [13, 14], [3, 4], [9, 12], [5, 10], [11, 12], [7, 8], [6, 7], [5, 9], [8, 10], [5, 6], [10, 12], [12, 13], [4, 5], [7, 9], [8, 11], [10, 11], [6, 7], [8, 9], [9, 10], [7, 8].

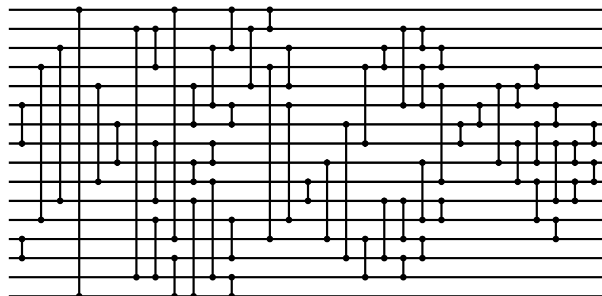


Figure 16: Evolved 16-input network with 60 comparators: [13, 14], [6, 8], [4, 12], [3, 11], [1, 16], [5, 10], [7, 9], [2, 15], [12, 15], [2, 4], [8, 11], [1, 13], [5, 7], [3, 6], [9, 10], [14, 16], [11, 16], [1, 3], [10, 15], [2, 5], [1, 2], [15, 16], [6, 7], [8, 9], [12, 14], [4, 13], [6, 12], [10, 11], [9, 13], [3, 5], [7, 14], [4, 8], [3, 4], [13, 15], [11, 14], [2, 6], [14, 15], [2, 3], [4, 6], [11, 13], [13, 14], [3, 4], [9, 12], [5, 10], [11, 12], [7, 8], [6, 7], [5, 9], [8, 10], [5, 6], [10, 12], [12, 13], [4, 5], [7, 9], [8, 11], [10, 11], [6, 7], [8, 9], [9, 10], [7, 8].

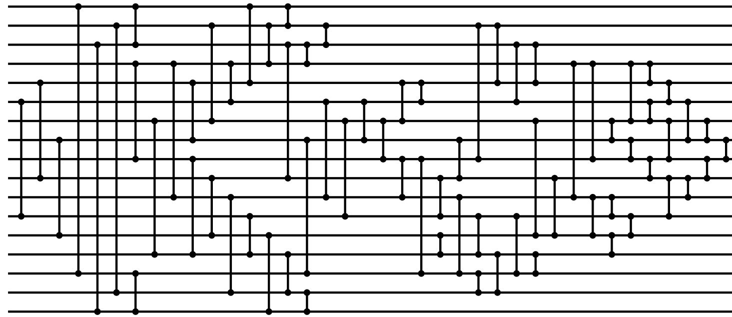


Figure 17: Evolved 17-input network with 71 comparators: [6, 12], [5, 10], [8, 13], [1, 15], [3, 17], [2, 16], [4, 9], [7, 14], [4, 11], [9, 14], [5, 8], [10, 13], [1, 3], [15, 17], [2, 7], [11, 16], [4, 6], [12, 14], [1, 5], [13, 17], [2, 4], [14, 16], [1, 2], [16, 17], [3, 10], [8, 15], [6, 11], [7, 12], [6, 8], [7, 9], [9, 11], [3, 4], [9, 15], [10, 12], [13, 14], [5, 7], [11, 15], [5, 6], [8, 10], [12, 14], [2, 3], [15, 16], [2, 9], [14, 16], [2, 5], [3, 6], [12, 15], [14, 15], [3, 5], [7, 13], [10, 13], [4, 11], [4, 9], [7, 8], [11, 13], [4, 7], [4, 5], [13, 14], [11, 12], [6, 7], [12, 13], [5, 6], [8, 9], [9, 10], [7, 9], [10, 12], [6, 8], [7, 8], [10, 11], [9, 10], [8, 9].

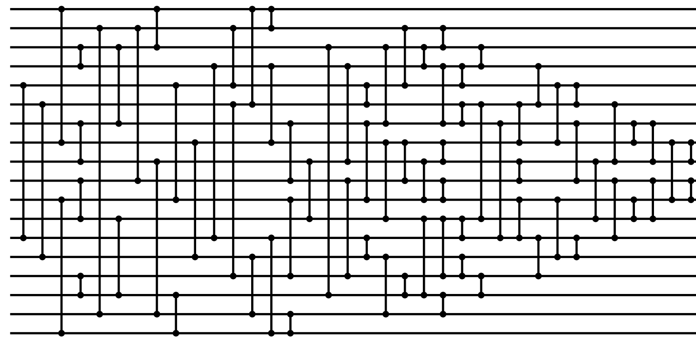


Figure 18: Evolved 18-input network with 78 comparators: [5, 13], [6, 14], [1, 8], [11, 18], [3, 4], [15, 16], [7, 9], [10, 12], [2, 17], [3, 7], [12, 16], [2, 10], [9, 17], [5, 11], [8, 14], [4, 13], [6, 15], [1, 3], [16, 18], [2, 5], [14, 17], [1, 6], [13, 18], [1, 2], [17, 18], [4, 8], [11, 15], [7, 10], [9, 12], [3, 16], [4, 9], [10, 15], [5, 6], [13, 14], [7, 11], [3, 7], [8, 12], [2, 5], [14, 17], [15, 16], [3, 4], [12, 16], [16, 17], [2, 3], [12, 15], [4, 7], [14, 15], [4, 5], [15, 16], [3, 4], [6, 7], [12, 13], [8, 10], [9, 11], [10, 11], [8, 9], [6, 12], [7, 13], [11, 13], [6, 8], [13, 15], [4, 6], [11, 14], [5, 8], [13, 14], [5, 6], [9, 10], [7, 10], [9, 12], [10, 13], [6, 9], [7, 8], [11, 12], [7, 9], [10, 12], [8, 11], [10, 11], [8, 9].

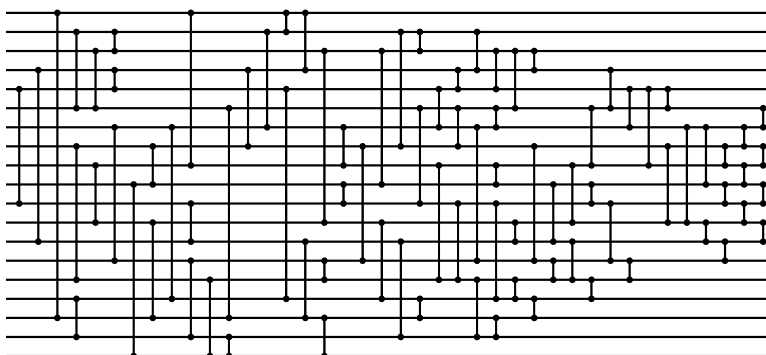


Figure 19: Evolved 19-input network with 86 comparators: [5, 11], [4, 13], [1, 17], [8, 15], [9, 12], [7, 14], [16, 18], [2, 6], [10, 19], [3, 6], [12, 17], [8, 10], [2, 3], [7, 16], [11, 13], [4, 5], [14, 18], [1, 9], [15, 19], [6, 17], [4, 8], [18, 19], [2, 7], [5, 16], [1, 2], [13, 17], [1, 4], [17, 19], [3, 12], [10, 11], [14, 15], [7, 9], [8, 14], [3, 10], [12, 16], [2, 8], [6, 11], [13, 18], [9, 15], [5, 7], [11, 15], [4, 5], [16, 17], [2, 3], [15, 18], [2, 4], [17, 18], [6, 8], [7, 14], [6, 7], [11, 16], [3, 5], [15, 16], [3, 6], [12, 13], [16, 17], [3, 4], [9, 10], [8, 14], [10, 13], [9, 12], [10, 11], [14, 15], [6, 9], [13, 15], [15, 16], [4, 6], [5, 7], [11, 14], [5, 9], [5, 6], [14, 15], [8, 12], [7, 12], [7, 10], [8, 9], [12, 13], [7, 8], [13, 14], [6, 7], [10, 11], [11, 12], [12, 13], [9, 10], [8, 9], [10, 11].

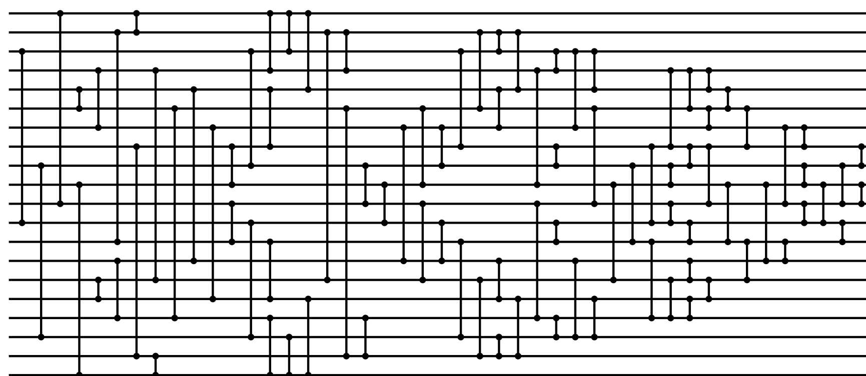


Figure 20: Evolved 20-input network with 92 comparators: [3, 12], [9, 18], [1, 11], [10, 20], [5, 6], [15, 16], [4, 7], [14, 17], [2, 13], [8, 19], [4, 15], [6, 17], [1, 2], [19, 20], [5, 14], [7, 16], [8, 10], [11, 13], [3, 9], [12, 18], [5, 8], [13, 16], [1, 4], [17, 20], [1, 3], [18, 20], [1, 5], [16, 20], [2, 15], [6, 19], [9, 11], [10, 12], [7, 14], [6, 10], [11, 15], [2, 4], [17, 19], [7, 9], [12, 14], [3, 8], [13, 18], [2, 6], [2, 3], [15, 19], [5, 7], [14, 16], [18, 19], [16, 19], [2, 5], [4, 10], [11, 17], [3, 4], [17, 18], [14, 18], [3, 7], [16, 18], [3, 5], [8, 9], [12, 13], [6, 11], [10, 15], [9, 13], [8, 12], [4, 8], [13, 17], [4, 6], [15, 17], [16, 17], [4, 5], [6, 7], [14, 15], [15, 16], [5, 6], [11, 12], [9, 10], [12, 13], [8, 9], [8, 11], [10, 13], [6, 8], [13, 15], [10, 14], [7, 11], [7, 8], [11, 12], [13, 14], [9, 10], [10, 12], [12, 13], [9, 11], [8, 9], [10, 11].

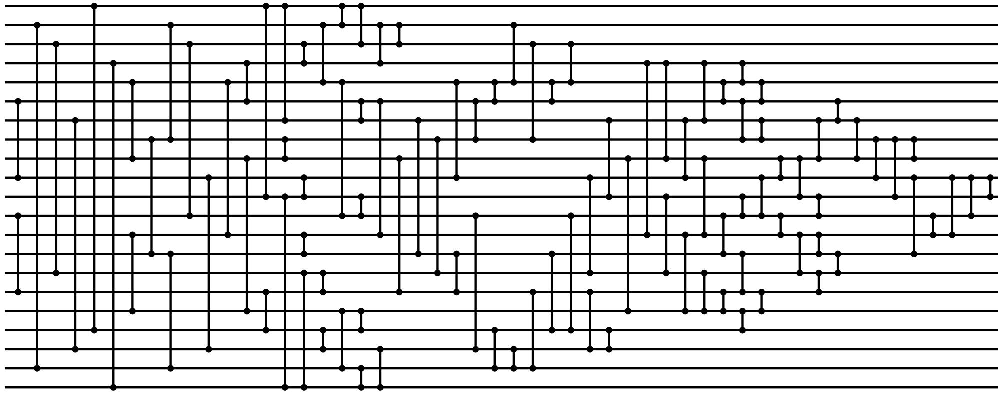


Figure 21: Evolved 21-input network with 102 comparators: [6, 10], [12, 16], [2, 20], [3, 15], [7, 19], [1, 18], [4, 21], [5, 9], [13, 17], [8, 14], [2, 8], [14, 20], [3, 12], [10, 19], [5, 13], [9, 17], [4, 6], [16, 18], [1, 11], [11, 21], [1, 7], [15, 21], [3, 4], [18, 19], [2, 5], [17, 20], [1, 2], [20, 21], [1, 3], [19, 21], [8, 9], [13, 14], [10, 11], [5, 12], [6, 7], [15, 16], [11, 12], [6, 13], [9, 16], [7, 14], [8, 15], [17, 18], [2, 4], [5, 10], [6, 8], [14, 16], [12, 19], [18, 20], [2, 3], [19, 20], [5, 6], [2, 5], [16, 20], [14, 18], [3, 8], [12, 18], [10, 15], [5, 6], [16, 19], [18, 19], [3, 5], [7, 11], [9, 17], [4, 13], [11, 15], [13, 17], [4, 9], [7, 10], [15, 17], [9, 13], [4, 7], [5, 6], [16, 17], [17, 18], [4, 5], [12, 14], [6, 8], [14, 16], [7, 8], [16, 17], [5, 6], [11, 12], [10, 12], [9, 10], [12, 13], [13, 15], [9, 11], [7, 9], [15, 16], [6, 7], [13, 14], [14, 15], [7, 9], [8, 10], [11, 12], [8, 11], [8, 9], [10, 14], [12, 13], [10, 13], [10, 12], [10, 11].

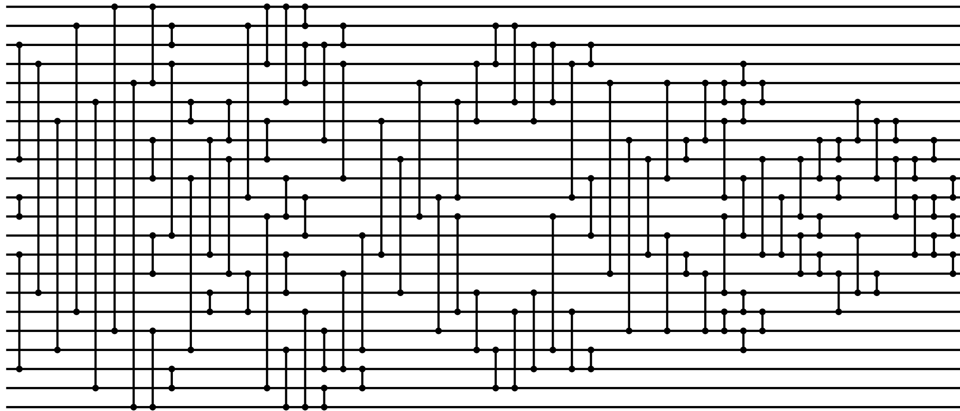


Figure 22: Evolved 22-input network with 108 comparators: [11, 12], [3, 9], [14, 20], [4, 16], [7, 19], [2, 17], [6, 21], [1, 18], [5, 22], [8, 10], [13, 15], [1, 5], [18, 22], [4, 13], [10, 19], [2, 3], [20, 21], [8, 14], [9, 15], [6, 7], [16, 17], [6, 8], [15, 17], [2, 11], [12, 21], [1, 4], [19, 22], [1, 6], [17, 22], [1, 2], [21, 22], [7, 9], [14, 16], [3, 5], [18, 20], [10, 12], [11, 13], [3, 8], [15, 20], [4, 10], [13, 19], [7, 14], [9, 16], [5, 12], [11, 18], [6, 11], [12, 17], [4, 7], [16, 19], [2, 3], [20, 21], [2, 4], [19, 21], [2, 6], [17, 21], [3, 7], [16, 20], [12, 19], [3, 6], [17, 20], [4, 11], [3, 4], [19, 20], [10, 13], [5, 15], [8, 18], [9, 14], [13, 18], [5, 10], [14, 15], [8, 9], [5, 8], [15, 18], [5, 6], [17, 18], [18, 19], [4, 5], [7, 11], [12, 16], [6, 7], [16, 17], [5, 6], [17, 18], [10, 13], [9, 14], [11, 14], [9, 12], [8, 10], [13, 15], [8, 9], [14, 15], [15, 17], [6, 8], [10, 11], [12, 13], [7, 10], [13, 16], [15, 16], [7, 8], [9, 12], [11, 14], [9, 10], [13, 14], [8, 9], [14, 15], [11, 12], [12, 13], [10, 11].

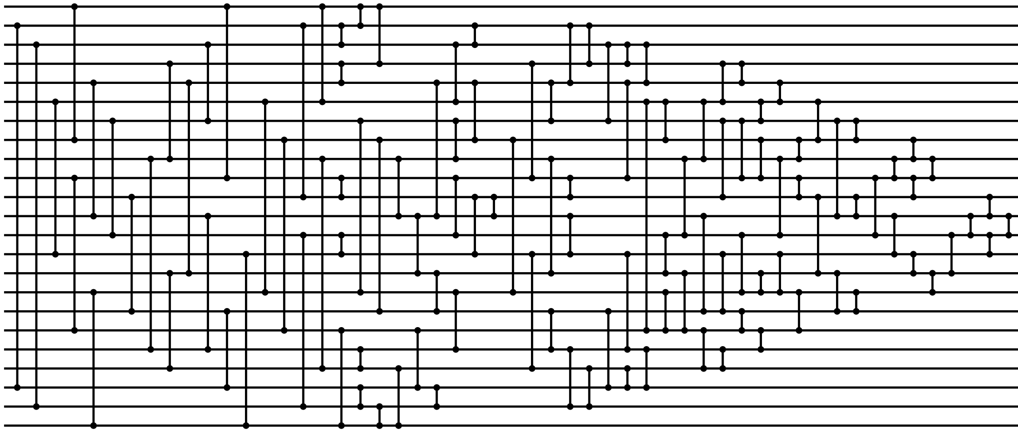


Figure 23: Evolved 23-input network with 118 comparators: [2, 21], [3, 22], [6, 14], [10, 18], [1, 8], [16, 23], [5, 12], [7, 13], [11, 17], [9, 19], [15, 20], [4, 9], [5, 15], [12, 19], [3, 7], [17, 21], [1, 10], [14, 23], [6, 16], [8, 18], [2, 11], [13, 22], [9, 20], [18, 23], [1, 6], [21, 22], [2, 3], [19, 20], [4, 5], [22, 23], [1, 2], [20, 23], [1, 4], [13, 14], [10, 11], [7, 16], [8, 17], [9, 12], [12, 15], [5, 12], [7, 9], [15, 17], [18, 21], [3, 6], [10, 13], [11, 14], [16, 19], [11, 12], [5, 8], [21, 22], [2, 3], [8, 16], [4, 10], [14, 20], [17, 19], [9, 15], [5, 7], [19, 22], [2, 5], [20, 22], [2, 4], [10, 11], [12, 14], [3, 7], [17, 21], [5, 10], [14, 19], [20, 21], [3, 4], [19, 21], [3, 5], [6, 18], [13, 15], [9, 13], [6, 8], [16, 18], [6, 9], [15, 18], [4, 6], [18, 20], [4, 5], [19, 20], [7, 11], [12, 17], [14, 17], [7, 10], [17, 18], [6, 7], [5, 6], [8, 10], [18, 19], [13, 16], [15, 16], [9, 13], [8, 9], [14, 16], [16, 18], [6, 8], [10, 11], [11, 15], [7, 12], [15, 17], [16, 17], [7, 8], [11, 12], [10, 13], [12, 14], [14, 15], [9, 10], [8, 9], [15, 16], [10, 11], [9, 10], [13, 15], [12, 13], [13, 14], [11, 12], [12, 13].

References

M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983. ISSN 0209-9683. doi: http://dx.doi.org.ezproxy.lib.utexas.edu/10.1007/BF02579338.

M. E. Alden. *MARLEDA: Effective Distribution Estimation Through Markov Random Fields*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2007. URL http://nn.cs.utexas.edu/keyword?alden:phd07. Technical Report AI07-349.

S. W. A. Baddar. *Finding Better Sorting Networks*. PhD thesis, Kent State University, 2009. URL http://rave.ohiolink.edu/etdc/view?acc_num.

K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

E. Bengoetxea, P. Larranaga, I. Bloch, and A. Perchant. Estimation of distribution algorithms: A new evolutionary computation approach for graph matching problems. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 454–469. Springer, 2001. URL http://dx.doi.org.ezproxy.lib.utexas.edu/10.1007/3-540-44745-8_30.

C. Chakrabarti and L.-Y. Wang. Novel sorting network-based architectures for rank order filters. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):502–507, 1994. ISSN 1063-8210. doi: http://dx.doi.org.ezproxy.lib.utexas.edu/10.1109/92.335027.

K.-L. Chung and Y.-K. Lin. A generalized pipelined median filter network. *Signal Processing*, 63(1):101 – 106, 1997. ISSN 0165-1684. doi: DOI:10.1016/S0165-1684(97)00144-8. URL http://www.sciencedirect.com/science/article/B6V18-3SNYT5C-1B/2/38110bb682311c8cc6169b64b233dac5.

K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *PPSN VI*, pages 849–858, 2000.

R. L. Drysdale and F. H. Young. Improved divide/sort/merge sorting networks. *SIAM Journal on Computing*, 4(3):264–270, 1975.

L. Graham and F. Oppacher. Symmetric comparator pairs in the initialization of genetic algorithm populations for sorting networks. *IEEE Congress on Evolutionary Computation, 2006 (CEC 2006)*, pages 2845–2850, 2006. doi: 10.1109/CEC.2006.1688666.

M. W. Green. Some improvements in non-adaptive sorting algorithms. In *Proceedings of the Sixth Annual Princeton Conference on Information Sciences and Systems*, pages 387–391, 1972.

C. A. Gunter, T.-H. Ngair, and D. Subramanian. Sets as anti-chains. In *ASIAN '96: Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security*, pages 116–128, London, UK, 1996. Springer-Verlag. ISBN 3-540-62031-1.

A. Hiasat and O. Hasan. Bit-serial architecture for rank order and stack filters. *Integration, the VLSI Journal*, 36(1-2):3 – 12, 2003. ISSN 0167-9260. doi: DOI:10.1016/S0167-9260(03)00017-8. URL http://www.sciencedirect.com/science/article/B6V1M-48NKGY5-1/2/53c0e6c9dfb4ef44292e616c4eab3356.

W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In J. D. Farmer, C. Langton, S. Rasmussen, and C. Taylor, editors, *Artificial Life II*. Addison-Wesley, Reading, MA, 1991.

H. Juillé. Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 351–358, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

R. Kannan and S. Ray. Sorting networks with applications to hierarchical optical interconnects. In *2001 International Conference on Parallel Processing Workshops*, pages 327–332. IEEE Computer Society, 2001. ISBN 0-7695-1260-7. doi: http://doi.ieeecomputersociety.org/10.1109/ICPPW.2001.951969.

P. Kipfer and R. Westermann. Improved GPU sorting. In M. Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 46. Addison-Wesley, 2005.

P. Kipfer, M. Segal, and R. Westermann. Uberflow: A gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM. ISBN 3-905673-15-0. doi: http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/1058129.1058146.

D. E. Knuth. *Art of Computer Programming: Sorting and Searching*, volume 3, chapter 5, pages 219–229. Addison-Wesley Professional, 2 edition, April 1998.

J. Korenek and L. Sekanina. Intrinsic evolution of sorting networks: A novel complete hardware implementation for FPGAs. In *Evolvable Systems: From Biology to Hardware*, pages 46–55. Springer, 2005. URL http://dx.doi.org.ezproxy.lib.utexas.edu/10.1007/11549703_5.

A. D. Korshunov. Monotone boolean functions. *Russian Mathematical Surveys*, 58(5):929, 2003. URL http://stacks.iop.org/0036-0279/58/i=5/a=R02.

J. R. Koza, J. R. Koza, I. Forest H. Bennett, I. Forest H. Bennett, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre. Evolving computer programs using rapidly reconfigurable field-programmable gate arrays and genetic programming. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 209–219, New York, NY, USA, 1998. ACM. doi: 10.1145/275107.275141.

J. R. Koza, D. Andre, F. H. Bennett, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*, chapter 21, pages 335–348. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1558605436.

T. Leighton and C. G. Plaxton. A (fairly) simple circuit that (usually) sorts. In *SFCS '90: Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 264–274 vol.1, Washington, DC, USA, 1990. IEEE Computer Society. ISBN 0-8186-2082-X. doi: http://dx.doi.org.ezproxy.lib.utexas.edu/10.1109/FSCS.1990.89545.

D. P. Mehta and S. Sahni. *Handbook of Data Structures and Applications*, chapter 3, pages 3.4–3.7. CRC Press, 2005. ISBN 9781584884354.

H. Mühlenbein and R. Höns. The estimation of distributions and the minimum relative entropy principle. *Evolutionary Computation*, 13(1):1–27, 2005.

D. G. O'Connor and R. J. Nelson. Sorting system with n-line sorting switch. United States Patent number 3,029,413, April 1962. Filed Feb 21, 1957. Issued Apr 10, 1962.

S.-W. Seo, T. yun Feng, and Y. Kim. A simulation scheme in rearrangeable networks. In *Proceedings of the 36th Midwest Symposium on Circuits and Systems*, pages 177 – 180 vol. 1, Aug 1993. doi: 10.1109/MWSCAS.1993.343100.

I. Shmulevich, T. M. Sellke, M. Gabbouj, and E. J. Coyle. Stack filters and free distributive lattices. In *Proceedings of the 1995 IEEE Workshop on Nonlinear Signal and Image Processing*, pages 927–930. IEEE Computer Society, 1995.

D. C. Van Voorhis. A generalization of the divide-sort-merge strategy for sorting networks. Technical Report 16, Digital Systems Laboratory, Stanford University, Stanford, California, August 1971.

D. C. Van Voorhis. An economical construction for sorting networks. In *Proceedings of AFIPS National Computer Conference*, pages 921–927, New York, NY, USA, 1974. ACM. doi: http://doi.acm.org/10.1145/1500175.1500347. URL `http://doi.acm.org/10.1145/1500175.1500347`.

Y.-M. Yeh and T.-y. Feng. On a class of rearrangeable networks. *IEEE Transactions on Computers*, 41(11):1361–1379, 1992. ISSN 0018-9340. doi: http://dx.doi.org.ezproxy.lib.utexas.edu/10.1109/12.177307.