Evolving Constructors for Infinitely Growing Sorting Networks and Medians

Lukáš Sekanina

Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
e–mail: sekanina@fit.vutbr.cz

Abstract. An approach is presented in which the object under design can grow continually and infinitely. First, a small object (that we call the embryo) has to be prepared to solve the trivial instance of a problem. Then the evolved program (the constructor) is applied on the embryo to create a larger object (solving a larger instance of the problem). Then the same constructor is used to create a new instance of the object from the created larger object and so on. Every new instance of the object is able to perform the function of all previous instances. As an example, constructors for growing sorting and median networks are evolved and analyzed.

1 Introduction

In the past few years, evolutionary algorithms have successfully been applied to automatically design various objects including computer programs, neural networks, electronic circuits, etc. [1, 16]. However, these methods have produced interesting results only for design of relatively small objects.

For instance, the problem of scale is usually considered as a major problem of evolvable hardware. It is practically impossible to evolve really complex circuits from scratch nowadays. Complex systems require huge number of gates (inputs, outputs, etc.) to be implemented, i.e. long genotypes in the case of the evolutionary approach. Long genotypes imply large search spaces. Then it is usually difficult to design an efficient search algorithm. Miller et al. offer two ways to build large systems [17]: (1) to discover a general scalable principle of design or (2) to produce building blocks as efficient and large as possible. Three major approaches have been developed in order to overcome the scaling problem [19]: functional level evolution, incremental evolution and the embryonic approach.

The concept of development in which the entire organism is built from a mother cell was adopted from biology to allow the "growth" of objects' complexity. When such a concept is implemented, the chromosome has to contain a prescription for constructing a target object.

It is a common feature of artificial developmental systems that the object under construction is not functional during its development. In contrary, biological systems are able to perform some operations at any given time point of the development. These "skills" are improved and new skills are continually

created during the growth of the system. The organism does not usually forget the obtained skills.

In this paper we present an approach in which an object can grow continually and infinitely. First, a small object (that we call the embryo) has to be prepared to solve the trivial instance of a problem. Then the evolved program (the constructor) is applied on the embryo to create a larger object (solving a larger instance of the problem). Then the constructor is used to create a new instance of the object from the created larger object and so on. Every new instance of the object is able to perform the function of all previous instances.

The main objective of this research is to design the constructor automatically by means of evolutionary techniques. The constructor will consist of two basic operations: copy and modify. It is shown that such the constructor can be evolved. As examples large sorting networks and median networks will be constructed because it is difficult to evolve large instances of these objects directly.

This paper is organized as follows. In Section 2 related research is presented in which evolutionary algorithms were combined with developmental systems. Sorting networks and medians are considered as the application domain in Section 3. The evolutionary algorithm utilized to design constructors of an infinitely growing median network is described in Section 4. Section 5 summarizes and discusses the obtained results. Finally conclusions are given in Section 6.

2 Related Work

Nature approaches the problem of scale by using a complicated mapping embodied in the process of biological development. Biological genomes contain a complex process of regulated gene expression to map genotype to phenotype.

In bioinspired hardware and software systems this mapping is often implemented by means of re-writing systems. Boers and Kuiper have utilized L-systems to create the architecture of feed-forward artificial neural networks (numbers of neurons and their connections) [2]. Haddow et al. have adopted L-system in order to evolve scalable circuits [7]. Kitano have applied a matrix re-writing to develop digital circuits. Three dimensional mechanical objects have been designed by evolution that also utilized a variant of L-system [9].

John Koza has introduced an original method in which analog circuits (competitive with best human designs) have been constructed according to the instructions produced by genetic programming [16]. Koza's team employed this technique for routine duplication of fourteen patented inventions in the analog circuit domain [20].

In another approach, Gordon and Bentley have utilized the interaction of genes and proteins to model the development in digital circuits [6]. CAM Brain machine [5] and POEtic platform [21] are examples of those systems that use cellular automata-based development.

Miller and Thomson have invented a developmental method for growing graphs and circuits using Cartesian genetic programming in order to evolve similar constructors to ours (referred to as iterators in [18]). Because they worked at

a very low level of abstraction (as configuration bits of a hypothetical reconfigurable hardware) no general constructor has been found for even parity circuits. However, other researchers have successfully evolved completely general solutions to even-parity problems; for instance Huelsbergen, who has worked at the machine code level [10].

3 Sorting Networks and Medians

The concept of sorting networks was introduced in 1954; Knuth traced the history of this problem in his book [14].

A *compare–swap* of two elements (a, b) compares and exchanges a and b so that we obtain $a \leq b$ after the operation. A sorting network is a sequence of compare–swap operations that depends only on the number of elements to be sorted, not on the values of the elements [14].

Although a standard sorting algorithm such as quicksort usually requires a lower number of compare operations than a sorting network, the advantage of the sorting network is that the sequence of comparisons is fixed. Thus it is suitable for parallel processing and hardware implementation, especially if the number of sorted elements is small. Figure 1 shows an example of a sorting network.

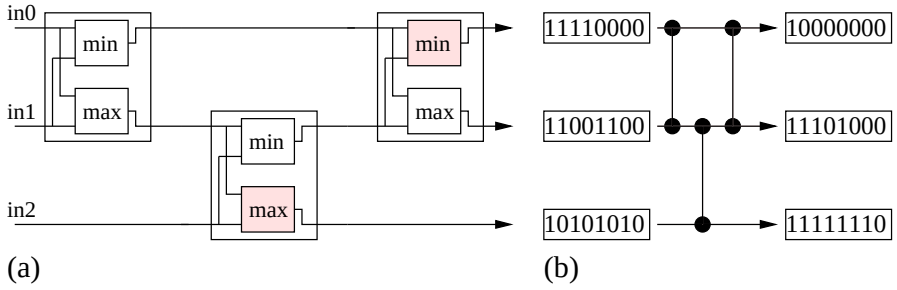


Fig. 1. (a) A 3-sorting network consists of 3 components, i.e. of 6 subcomponents (elements of maximum or minimum). A 3-median network consists of 4 subcomponents. (b) Alternative symbol. This sorting network can be tested in a single run if 2^3bits can be stored in a single data unit.

Having a sorting network for N inputs, the *median* is simply the output value at the middle position (we are interested in odd N only in this paper). For example, efficient calculation of the median value is important in signal processing where median filters are widely used with N = 3x3 or 5x5 [19].

The number of compare–swap components and the delay are two crucial parameters of any sorting network. Since we will only be interested in the number of components in this paper, the following Table 1 shows the number of components of some of the best currently know sorting networks, i.e. those which require the least number of components for sorting N elements. Some of these networks

($N = 13$–16) were discovered using evolutionary techniques [3, 8, 12, 16]. However, the evolutionary approach is not scalable. For instance, we were not able to directly evolve any 25-median network up to now.

Note that the compare-swap consists of two subcomponents: maximum and minimum. Because we need the middle output value only in the case of the median implementation, we can omit some subcomponents (dead code at the output marked in gray in Fig. 1) and so to reduce implementation cost in hardware. Hence in the case of K components, we obtain $2K - K + 1$ subcomponents (Table 1, line 3). However, in addition to deriving median networks from sorting networks, specialized networks have been proposed to implement optimal median networks. Table 1 (line 2) also presents the best-known numbers of subcomponents for optimal median networks. These values are derived from the table on page 226 of Knuth's book [14] and from papers [4, 15, 22]. The space complexity of the general algorithm constructing sorting networks is $O(N(logN)^2)$ [14].

Table 1. Best known minimum-comparison sorting networks and median networks for some N. $c(N)$ denotes the number of compare–swap operations, $s(N)$ is the number of subcomponents. The last line holds for median networks derived from sorting networks.

N	3	4	5	6	7	8	9	10	11	12	13	14	15	16	25
sortnet, $c(N)$	3	5	9	12	16	19	25	29	35	39	45	51	56	60	144
median, $s(N)$	4	-	10	-	20	-	30	-	42	-	52	-	66	-	174
median, $s(N)$	4	-	14	-	26	-	42	-	60	-	78	-	98	-	264

The *zero–one* principle helps with evaluating sorting networks. It states that if a sorting network with N inputs sorts all 2^N input sequences of 0's and 1's into nondecreasing order, it will sort any arbitrary sequence of N numbers into nondecreasing order [14]. Furthermore, if we use a proper encoding, on say 32 bits, and binary operators AND instead of minimum and OR instead of maximum, we can evaluate 32 test vectors in parallel and thus reduce the testing process 32 times. Figure 1 illustrates this idea for 3 bits. Note that it is usually impossible to obtain the general solution if only a subset of input vectors is utilized during the evolutionary design [11].

4 Development Using Copy and Modify

Consider that we have a 3-median network (i.e. $N = 3$ as seen in Fig. 1) and we are going to evolve a program (constructor) that will create a 5-median network from the 3-median network. The same program has to be able to create a 7-median network from the 5-median network and so on. Another available information is the number of active inputs (i.e. N) of the currently constructed network.

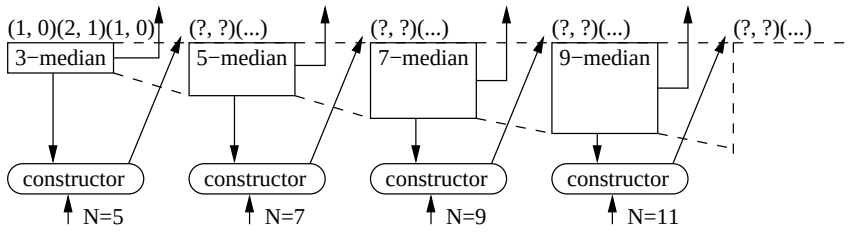


Fig. 2. Designing larger sorting networks from smaller sorting networks by means of constructor.

4.1 Representation

The 3-median network is represented by the sequence of pairs (1, 0)(2, 1)(1, 0) indicating the ordering of compare–swap operations over the inputs 0, 1 and 2. The constructor is also a sequence of instructions: each of which is encoded as three integers. Only three instructions are utilized: *copy*, *modify* and *skip*. Table 2 introduces their operational codes and parameters. The sequences representing sorting networks as well as constructors are implemented using variable-length arrays. A sentinel (STOP) indicates the end of the valid sequence.

Table 2. Instruction set. The pc pointer is increased to $pc \leftarrow pc + 3$ after execution of each instruction. The former sequence (a, b)(c, d)... is transcribed to new sequence (a', b')(c', d')... M denotes the number of inputs of the currently created median network.

Instruction	Opcode	Op1	Op2	Description
ModifyN	0	x	y	$(a', b') \leftarrow ((a + x) \bmod M, (b + y) \bmod M)$ $pn \leftarrow pn + 2$
ModifyR	1	x	y	$(a', b') \leftarrow ((a + x) \bmod M, (b + y) \bmod M)$ $pf \leftarrow pf + 2$, $pn \leftarrow pn + 2$
CopyN	2	x	y	copies $M - x$ pairs from former to new sequence $pn \leftarrow pn + 2(M - x)$
CopyR	3	x	y	copies $M - x$ pairs from former to new sequence $pf \leftarrow pf + 2(M - x)$, $pn \leftarrow pn + 2(M - x)$
Skip	4	x	y	$pf \leftarrow pf + 2$

The constructor, according to its program, sequentially reads the embryo and copies (or copies and modifies) the embryo into the next instance. Three pointers are utilized in order to indicate the current position in sequences: the embryo pointer (pf), the next instance pointer (pn) and the constructor pointer (pc). The constructor creates the next instance of the network not from the entire previous instance, but only from its newest part. The process of construction terminates when either STOP symbol is read in the sequence of "embryo" or

all instructions of the constructor have been executed. The constructed median network is then tested in the process of fitness calculation.

4.2 Evolutionary algorithm

Any chromosome consists of a sequence of integers that represents a constructor. We initially approached the problem with variable-length chromosomes. However, the approach did not produce general constructors. Hence we had to use the fixed size of chromosomes. After some experiments we learned that useful constructors consist of 5–8 instructions, i.e. 15–24 integers.

A typical setting of the evolutionary algorithm is as follows. Initial population of 320 individuals is seeded randomly using alleles of 0–4. New individuals are generated using operators of crossover ($p_c = 60\%$) and mutation (1 integer per chromosome). Tournament selection with base 2 is combined with elitism. The evolutionary algorithm is left running until a fully correct individual is found or 2000 generations are exhausted. We also increase mutation rate if no improvement is observable during the last 30 generations.

The objective is to evolve a general constructor. However, because of scalability problems only several instances of the median network can be evaluated in the fitness calculation process. Hence a candidate constructor is used to build the 5-median, 7-median, 9-median and 11-median network from the 3-median embryo. We have not been able to evolve general constructors by testing smaller networks. The fitness value is calculated as follows:

$$fitness = m_5 + m_7 + m_9 + m_{11},$$

where m_k denotes the number of median values calculated correctly from 2^k testing binary vectors of size k. Hence 32+128+512+2048=2720 is the best possible value that we could obtain.

5 Results and Discussion

If a constructor is able to create the median network for a sufficiently high value of N (N=27 in our case) then we consider the constructor as general. In our experiments, 108 of 180 runs led to the perfect fitness. However, we identified only 11 general constructors. These general constructors (listed in Table 3) consist of 5 to 8 instructions. For example, gr5-3 and gr5-4 are practically the same programs because they differ only in the last integer, which represents the second (meaningless) operand of the Modify instruction as seen in Table 2.

We were also interested in reducing the number of components in the evolved designs. Table 4 shows the size of two median networks generated using gr5-4 and gr6-1 constructors. While the gr5-4 constructor consists of six instructions, the gr6-1 utilizes only five instructions. The gr5-4 constructor – and thus also gr5-1 and gr5-3 constructors – are probably the best constructors we have ever evolved. We were not able to reduce the size of networks if only 5 instructions should be used. Furthermore, Table 4 also shows that we were not able to

Table 3. Chromosomes of eleven general constructors evolved.

Constructor	Chromosome
gr3-6	0,2,2, 0,3,3, 2,2,4, 0,2,2, 3,4,4, 3,4,0, 2,1,3, 2,1,3
gr4-6	0,2,2, 0,3,2, 0,2,2, 0,3,3, 0,2,1, 3,0,4, 2,2,0
gr4-7	0,2,2, 0,3,1, 0,2,4, 0,3,3, 0,2,2, 3,0,1, 2,2,3
gr4-8	0,2,2, 0,2,3, 0,0,2, 0,3,3, 0,2,1, 3,3,4, 3,2,3
gr4-9	0,2,2, 0,1,1, 0,3,2, 0,3,3, 3,2,3, 3,1,4, 0,2,0
gr5-1	0,2,2, 0,3,2, 0,3,1, 0,3,3, 3,1,0, 2,3,2
gr5-3	0,2,2, 0,3,1, 0,3,3, 0,2,2, 3,2,3, 2,0,3
gr5-4	0,2,2, 0,3,1, 0,3,3, 0,2,2, 3,2,3, 2,0,4
gr6-1	1,2,1, 0,2,2, 0,1,1, 2,3,2, 2,1,1
gr13-8	1,2,1, 0,2,2, 0,1,1, 2,3,0, 3,1,2
gr13-9	1,2,1, 0,2,2, 0,1,1, 2,3,2, 3,1,3

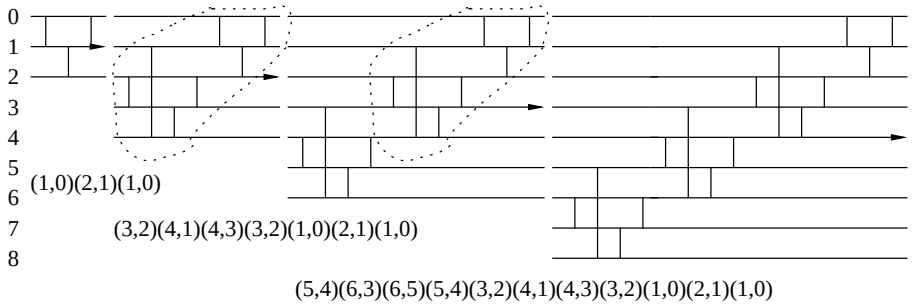
beat the well-known general approach for designing larger networks illustrated in Fig. 5.

Table 4. The number of compare-swap operations $c(N)$ used by two evolved constructors and the conventional approach (according to Fig. 5) to realize growing median networks.

N	3	5	7	9	11	13	15	17	19	21	23	25	27
gr5-4	3	10	21	36	55	78	105	136	171	210	253	300	351
gr6-1	3	10	23	40	61	86	115	148	185	226	271	320	373
conventional	3	10	21	36	55	78	105	136	171	210	253	300	351

Figure 3 shows that the gr4-5 constructor generates regular pattern of compare–swap operations. First, new four compare–swap operations are generated in order to deal with two emerging inputs. Then the median network is copied from the previous instance. It is interesting to see that the first two comparisons can be performed in parallel.

It was surprising for us that although we wanted to evolve general constructors only to create median networks, we obtained general constructors for building sorting networks. We used all 11 general constructors to generate sorting networks and they worked! An open question is whether general constructors exist that create median networks only but they do not create sorting networks. We can observe after the analysis of growing networks that their structure is very similar to the well-known principle of building a sorting network for $N + 1$ elements from a sorting network of N elements (see [14]). Thus we rediscovered this principle by artificial evolution.



constructor:
ModifyN 2,2; ModifyN 3,1; ModifyN 3,3; ModifyN 2,2; CopyR 2,3; CopyN 0,4;

Fig. 3. Constructing larger networks using the evolved gr5-4 constructor.

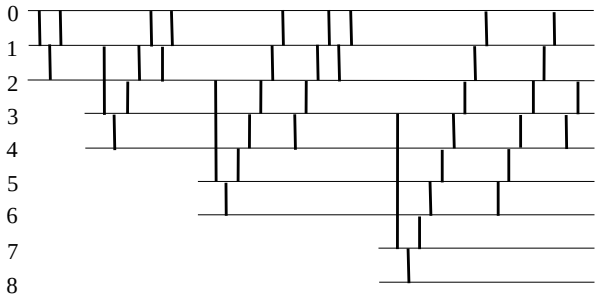


Fig. 4. Constructing larger networks using the evolved gr6-1 constructor.

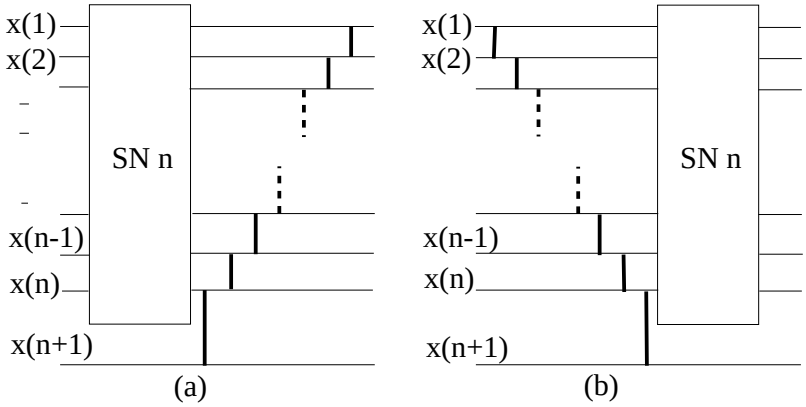


Fig. 5. Making (n+1)-sorters from n-sorters: (a) insertion, (b) selection.

Another question is whether the created networks are of practical interest. Although a number of developmental systems have been proposed to make the evolutionary design scalable, only a few of them have been applied to design objects more complex than we can do without development. The created networks are large and fully operational; however, inefficient in terms of compare–swap operations.

The proposed algorithm produced the expected results, since a lot of problem-domain knowledge (such as the usage of copy and modify instructions) has been presented in its inductive bias. The idea of evolving constructors for infinitely growing objects is generally applicable. However, it is difficult to define embryo and appropriate domain knowledge for a particular problem. Although it seems that no really innovative designs can be discovered by means of development, large sorting and median networks represent typical examples that can benefit from inspiration in ontogeny.

6 Conclusions

A simple method with strong inductive bias was proposed for evolving constructors of infinitely growing median networks. It was not a problem to evolve general constructors for sorting network since they are the same as for growing median networks. However, the open questions are whether it is possible to evolve general constructors for creating (1) more area–efficient median networks and (2) those area–efficient networks that do not fully operate during development. These problems will be investigated in our future research.

Acknowledgment

The research was performed with the Grant Agency of the Czech Republic under No. 102/03/P004 *Evolvable hardware based application design methods* and the Research intention No. CEZ MSM 262200012 – *Research in information and control systems.*

References

1. Bentley, P.: Evolutionary Design By Computers. Morgan Kaufmann Publishers, San Francisco CA 1999
2. Boers, E. J. W., Kuiper, H.: Biological Metaphors and the Design of Artificial Neural Networks. Master Thesis, Departments of Computer Science and Experimental and Theoretical Psychology, Leiden University, 1992
3. Choi, S. S., Moon, B. R.: More Effective Genetic Search for the Sorting Network Problem. In. Proc. of the Genetic and Evolutionary Computation Conference GECCO'02, Morgan Kaufmann, 2002, p. 335–342
4. Devillard, N.: Fast Median Search: An ANSI C Implementation. 1998 `http://ndevilla.free.fr/median/median/index.html`

5. de Garis, H. et al.: ATR's Artificial Brain (CAM-Brain) Project: A Sample of What Individual "CoDi-1 Bit" Model Evolved Neural Net Modules Can Do With Digital and Analog I/O. In Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware, IEEE CS Press, 1999, p. 102–110
6. Gordon, T. G. W., Bentley P.: Towards Development in Evolvable Hardware. In Proc. of the 2002 NASA/DoD Conference on Evolvable Hardware, IEEE CS Press, 2002, p. 241–250
7. Haddow, P., Tufte, G., van Remortel, P.: Shrinking the Genotype: L-systems for EHW? In Proc. of the 4th International Conference on Evolvable Systems: From Biology to Hardware, LNCS 2210, Springer–Verlag, p. 128–139
8. Hillis, W.D.: Co-evolving parasites improve simulated evolution as an optimization procedure. Physica D 42, 1990, p. 228–234
9. Hornby, G. S., Pollack, J. B.: The Advantages of Generative Grammatical Encodings for Physical Design. In. Proc. of the 2001 Congress on Evolutionary Computation CEC2001, IEEE CS Press, p. 600–607
10. Huelsbergen, L.: Finding General Solutions to the Parity Problem by Evolving Machine-Language Representations. In Proc. of Conf. on Genetic Programming, 1998, p. 158–166
11. Imamura, K., Foster, J. A., Krings, A. W.: The Test Vector Problem and Limitations to Evolving Digital Circuits. In: Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, IEEE CS Press, 2000, p. 75–79
12. Juillé, H.: Evolution of Non-Deterministic Incremental Algorithms as a New Approach for Search in State Spaces. In Proc. of 6th Int. Conf. on Genetic Algorithms, Morgan Kaufmann, 1995, p. 351–358
13. Kitano, H.: Morphogenesis for Evolvable Systems. In Towards Evolvable Hardware: The Evolutionary Engineering Approach, LNCS 1062, Springer–Verlag, 1996, p. 99–117
14. Knuth, D. E.: The Art of Computer Programming: Sorting and Searching (2nd ed.), Addison Wesley, 1998
15. Kolte, P., Smith, R., Su, W.: A Fast Median Filter Using AltiVec. In Proc. of the IEEE Conf. on Computer Design, Austin, Texas, IEEE CS Press, 1999, p. 384–391
16. Koza, J. R., Bennett III., F. H., Andre, D., Keane, M. A.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, 1999
17. Miller, J., Job, D., Vassilev, V.: Principles in the evolutionary design of digital circuits – Part II. Genetic Programming and Evolvable Machines. **1**(2), 2000, p. 259–288
18. Miller, J., Thomson, P.: A Developmental Method for Growing Graphs and Circuits. In: Proc. of the 5th Conf. on Evolvable Systems: From Biology to Hardware ICES 2003, LNCS 2606, Springer–Verlag, 2003, p. 93–104
19. Sekanina, L.: Evolvable Components: From Theory to Hardware Implementations. Natural Computing Series, Springer–Verlag, 2003
20. Streeter, M. J., Keane, M. A., Koza, J. R.: Routine Duplication of Post-2000 Patented Inventions by Means of Genetic Programming. In: Proc. of the 5th European Conference on Genetic Programming, Kinsale, Ireland, LNCS 2278, Springer-Verlag, 2002, p. 26–36
21. Tempesti, G. et al.: Ontogenetic Development and Fault Tolerance in the POEtic Tissue. In: Proc. of the 5th Conf. on Evolvable Systems: From Biology to Hardware ICES 2003, LNCS 2606, Springer–Verlag, 2003, p. 141–152
22. Zeno, R.: A Reference of the Best-Known Sorting Networks for up to 16 Inputs. 2003, `http://www.angelfire.com/blog/ronz/`