Genetic Programming and Evolvable Machines, 6:3, 319–347, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.
DOI: 10.1007/s10710-005-2987-8

Evolutionary Design of Arbitrarily Large Sorting Networks Using Development

LUKÁŠ SEKANINA sekanina@fit.vutbr.cz
MICHAL BIDLO bidlom@fit.vutbr.cz
Faculty of Information Technology,Brno University of Technology, Božetěchova 2, 612 66, Brno, Czech Republic

Submitted June 17, 2004; Revised February 9, 2005

Published online: 12 August 2005

Communicated by: Julian Miller

Abstract. An evolutionary algorithm is combined with an application-specific developmental scheme in order to evolve efficient arbitrarily large sorting networks. First, a small sorting network (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 sorting network (solving a larger instance of the problem). Then the same constructor is used to create a new instance of the sorting network from the created larger sorting network and so on. The proposed approach allowed us to rediscover the conventional principle of *insertion* which is traditionally used for constructing large sorting networks. Furthermore, the principle was improved by means of the evolutionary technique. The evolved sorting networks exhibit a lower implementation cost and delay.

Keywords: evolutionary algorithm, development, sorting network, scalability

1. Introduction

Evolutionary design has really become a popular and successful design method in many engineering areas in the recent years [4, 6]. For instance, innovative and useful solutions are routinely discovered by evolutionary techniques in the field of evolvable hardware [13, 19, 30, 34]. Evolutionary design has allowed us (1) to discover novel solutions, with features that are beyond the scope of the solutions generated by conventional engineering methods and (2) to perform hard engineering work in some areas automatically.

In the engineering domain we can formulate the goal of the evolutionary design as follows: to produce new, innovative and useful solutions to complex problems that can automatically be created with the minimal effort and domain knowledge of a designer. Therefore, the challenge of conventional design is being substituted by designing an evolutionary process that automatically performs the design for a given problem. This may be harder than performing the creative design directly, but makes automation possible.

In fact, only relatively simple designs were successfully evolved so far. As Torresen commented on the classical paradigm of evolutionary design, large objects require longer chromosomes, i.e. the search space is also larger and so difficult to be effectively explored by evolutionary algorithm [38]. It also gets difficult and time consuming to evaluate candidate solutions as they get more complex. For instance, in case of digital combinational circuits, the time of evaluation of a circuit doubles with adding a single input variable.

In order to eliminate the problem of scale in the evolutionary circuit design, designers have introduced various approaches, which can be divided into three classes: functional level evolution (e.g. [32]), incremental evolution (e.g. [38]) and development (e.g. [14, 16]). We will be interested in development in this paper.

When a sort of development is included into an evolutionary algorithm, a chromosome has to contain a prescription for constructing a target object rather than a description of a target object itself. A number of approaches to development were tested and described in literature to solve various problems. However, in most cases, the obtained solutions have shown the same complexity as the solutions generated without development (e.g. [14, 16, 31]).

The goal of this paper is to combine evolutionary design with a form of development in order to evolve "infinitely scalable" objects, in particular, arbitrarily large sorting networks. We chose the sorting networks because (1) conventional solutions to designing arbitrarily large sorting networks exist and, therefore, we can compare the results, (2) evolutionary techniques have already been utilized to design a sorting network with the predefined number of inputs (but not to design an arbitrarily large sorting network), and (3) sorting networks are suitable for implementation in hardware which is our main research objective in general (but not in this paper).

An approach is presented in which a sorting network can grow continually and infinitely. First, a small sorting network (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 sorting network (solving a larger instance of the problem). Then the same constructor is used to create a new instance of the sorting network from the created larger sorting network and so on. Every new instance of the sorting network is able to perform the function of all its previous instances. We will demonstrate that the constructor can be designed automatically by means of evolutionary techniques. Furthermore, it will be shown that some of evolved constructors are able to produce much more efficient sorting networks (in terms of the comparison count and delay) than a traditional conventional solution can offer. The proposed method improves Sekanina's initial approach, described in [35], which did not yield better solutions than conventional methods. His method also did not deal with delay of resulting circuits.

This paper is organized as follows. Section 2 briefly surveys principles, models and applications of development. In Section 3 basic concepts and design approaches to sorting networks are presented. Section 4 introduces the proposed approach to the evolutionary design of sorting networks with development. The obtained results are summarized in Section 5 and discussed in Section 6. Conclusions are given in Section 7.

2. Development in evolutionary design

A multicellular organism is determined by its genetic information and the environment in which lives. In the process of development an adult organism is formed from a zygote. Genes, inherited from parent(s), are used to create proteins. Proteins activate or suppress other genes, work as signals among cells, influence internal functions of the cells and perform many other important roles. Therefore, they control the growth, position and behavior of all cells. All these processes are very complex and not fully understood [1].

In case of evolutionary algorithms, the process of development is usually considered as a nontrivial *genotype–phenotype mapping*. While genetic operators work with genotypes, the fitness calculation is applied on phenotypes created by means of a developmental system. Various approaches have been investigated in order to utilize non-trivial genotype–phenotype maps (see survey in [28]). For instance, Dawkins's *biomorphs* represent a very nice example [11]. These techniques have been referred to as, for example, developmental encodings, morphogenesis, embryogenesis, generative systems, neurogenesis, computational embryology, etc. Recently, Kumar has introduced a more general, collective umbrella term, *Computational Development* [28].

In the context of evolutionary algorithms, computational development might be utilized to achieve diverse objectives, including: adaptation, compacting genotypes, reduction of search space, allowing more complex solutions in solution space, regulation, regeneration, repetition, robustness, scalability, evolvability, parallel construction, emergent behavior and decentralized control [28]. In particular, we mainly deal with scalability in this paper.

2.1. Evolvability and scalability

The little understood capacity to be able to reach good solutions via evolution is called evolvability [33]. Evolvability is the ability to evolve easily. Wagner and Altenberg noted that in evolutionary algorithms it was found that the Darwinian process of mutation, recombination and selection is not universally effective in improving complex systems like computer programs or circuits. For adaptation to occur, these systems must possess evolvability, i.e. the ability of random variations to sometimes produce improvement. It was found that evolvability critically depends on the way of mapping genetic variation onto phenotypic variation, an issue known as the representation problem. The genotype–phenotype map is the common theme underlying such varied biological phenomena as developmental constraints, biological versatility, developmental dissociability, morphological integration, and some others [39].

Scalability is considered as one of the most difficult problems in the evolutionary design field in general and in the evolvable hardware field in particular. Despite increased interest in techniques of effective encoding, smart search strategies and clever fitness functions [14, 16, 32, 34, 38], only very small circuits (in comparison to the circuits designed conventionally) were evolved up to now. Hence developmental approaches have become very popular in the recent years.

2.2. Models and applications of development

Models of development were surveyed in Chapter 2 of [28]. Scientists construct these models either to learn how development works in nature or to solve the problems of practical evolutionary design in engineering or in the field of artificial life. In this section we will briefly recall only a class of models related (in some way) to our work — evolutionary design of arbitrarily large sorting networks.

In bio-inspired hardware and software systems the genotype phenotype mapping is often implemented by means of *rewriting systems*. The first rewriting developmental (neuro)system was investigated by Kitano [25]. Later, among others, Boers and Kuiper [7] have utilized L-systems to create the architecture of feed-forward artificial neural networks. Haddow et al. [17] have adopted L-system in order to evolve scalable digital circuits.

Three-dimensional mechanical objects have been designed by evolution that also utilized a variant of L-system in its genotype–phenotype map [21].

John Koza introduced an original method in which novel analog circuits have been constructed according to the instructions produced by genetic programming [27]. Among other activities, Koza's team employed this technique for routine duplication of fourteen patented inventions in the analog circuit domain [36].

In another approach, Gordon and Bentley have utilized the interaction of artificial genes and proteins to model development in digital circuits [14]. CAM Brain machine [12] and POEtic platform [37] are examples of those systems that use cellular automata-based development. Gruau proposed a genetic encoding scheme for artificial neural networks based on a cellular duplication and differentiation process. The construction starts with a single cell that undergoes a number of duplications and transformations phases ending up in a complete artificial neural network. The genotype is considered as a collection of rules governing the process of cell division and transformations [15].

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 [31]). 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 their task, i.e. the design of large even parity circuits. However, other researchers have successfully evolved completely general solutions to the even-parity problem; for instance Huelsbergen, who has worked at the machine code level [22].

In order to evolve 3D shape and form Kumar has used complex and, therefore, realistic models of development inspired by genetic regulatory networks [28]. Bentley has invented fractal proteins for the same purpose. A fractal protein is a finite square subset of Mandelbrot set, defined by three artificial codons that form the coding region of a gene in the genome of a cell [5].

These methods have illustrated various approaches to the development; however, only a few of them were successful with designing large systems for real-world applications.

3. Sorting networks and their design

The concept of sorting networks was introduced in 1954; Knuth traced the history of this problem in his book [26]. A sorting network is defined as a sequence of compare–swap operations (comparators) that depends only on the number of elements to be sorted, not on the values of the elements. A *compare–swap* of two elements (a, b) compares and exchanges a and b so that we obtain $a \leq b$ after the operation.

The main 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 3-input sorting network.

The number of compare–swap components and delay are two crucial parameters of any sorting network. By delay we mean the minimal number of groups of compare–swap components that must be executed sequentially. Designers try to minimize the number of comparators, delay or both parameters. Table 1 shows the number of comparators and delay of some of the best currently known sorting networks. Some of these networks were designed (or rediscovered) using evolutionary techniques [8–10, 20, 24, 27]. In most cases the evolutionary approach was based on the encoding given in Figure 1 (in which

Table 1. The number of comparators and delay of the best currently known sorting networks.

Inputs (N)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Delay	0	1	3	3	5	5	6	6	7	8	8	9	10	10	10	10
Comparators	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60

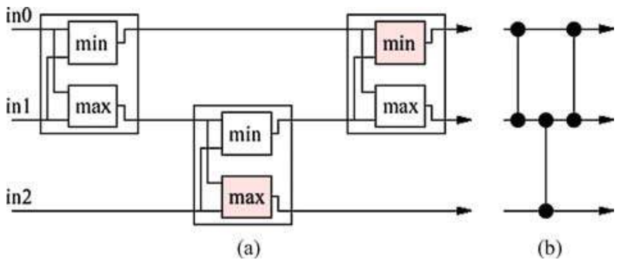

Figure 1 (a) A three-input sorting network consists of three comparators. (b) Alternative symbol. This network can be described using the string (0,1) (1,2) (0,1).

comparator inputs are encoded using two integers). Evolutionary techniques were also utilized to discover fault-tolerant sorting networks [18, 29].

In order to find out whether an N-input sorting network operates correctly we should test $N!$ input combinations. Thanks to the *zero–one* principle this number can be reduced. This principle states that if an N-input sorting network 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 [26]. 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. Unfortunately, it is usually impossible to obtain the general solution if only a subset of input vectors is utilized during the evolutionary design [23].

Sorting networks are usually designed for a fixed number of inputs. It is also valid for the mentioned evolutionary approaches. Note that the evolutionary approach is not scalable. Some conventional approaches exist for designing *arbitrarily* large sorting networks. Figure 2 shows two principles for constructing a sorting network for $N + 1$ inputs when an N-input network is given [26].

– Insertion — the $(N + 1)$st input is inserted into a proper place after the first N elements have been sorted.
– Selection — the largest input value can be selected before we proceed to sort the remaining ones.

We can see that the insertion principle corresponds to the *straight insertion* algorithm known from the theory of sorting. The selection principle is related to the *bubble sort* algorithm. Examples of sorting networks created using the two principles are shown in Figure 3. Observe that while physical positions of comparators are different, their logical positions are equivalent. Hence it is possible to re-arrange these comparators in order

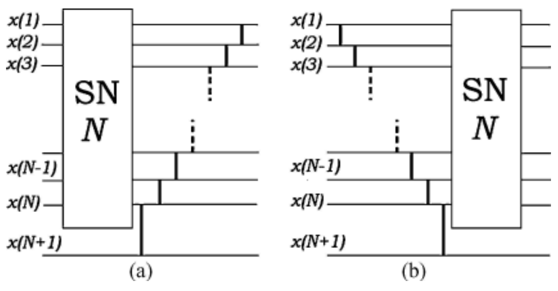

Figure 2 Making $(N+1)$-sorters from N-sorters: (a) insertion and (b) selection principle.

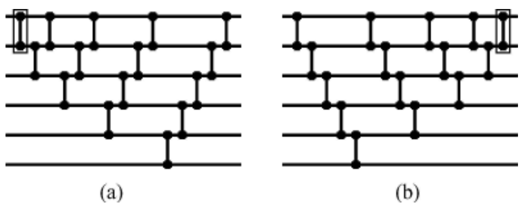

Figure 3 Examples of sorting networks created using: (a) insertion and (b) selection principle.

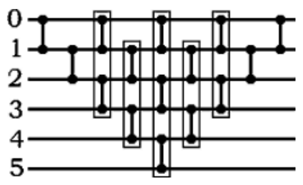

Figure 4 A sorting network with parallel layers (in rectangles).

to obtain a single sorting network (see Figure 4). The network contains the comparators that can be executed in parallel. Therefore, its delay can be reduced substantially. These comparators form the so-called *parallel layers*.

It is obvious that the sorting networks created using insertion or selection principle are much larger than those networks designed for a particular *N*. However, the method can be treated as a general design principle for building *arbitrarily* large sorting networks. In next sections, the principle will be rediscovered firstly and then improved by means of evolutionary techniques.

4. Development for sorting networks

The objective of this paper is to propose an application-specific development for evolutionary algorithm, which, consequently, will be able to produce innovative arbitrarily large sorting networks. Recall that the common evolutionary design of sorting networks deals with designing a *single sorting network* with a predefined number of inputs.

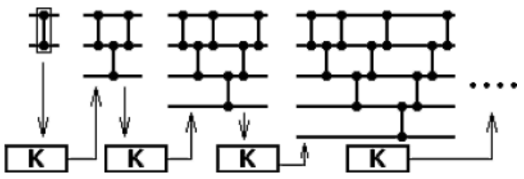

Figure 5 Designing larger sorting networks from smaller sorting networks by means of a constructor K.

4.1. *Basic concept*

The proposed algorithm is based on Sekanina's approach described in [35]. Unlike in [35] we deal with the delay of sorting networks. A genetic algorithm is used to design a program—constructor (consisting of application-specific instructions)—that is able to create a larger sorting network from a smaller one (the smallest one is called the embryo). Then the constructor is applied on its previous results in order to create a larger sorting network and so on. Algorithm 1 and Figure 5 demonstrate this idea.

Algorithm 1:

```
Set time t = 0;
Create initial population of programs P(t);
Create sorting networks using programs from P(t);
Evaluate sorting networks;
while (termination condition is false) do
{
  $t = t + 1$;
  $P(t)$ = create new population using $P(t - 1)$;
  Create sorting networks using programs from $P(t)$;
  Evaluate sorting networks;
}
```

The development is realized as follows. Consider that we have a 2-input sorting network (i.e. $N = 2$ as seen in Figure 5) and we are going to evolve a program (constructor) that will create a 3-input sorting network from the 2-input sorting network. The same program has to be able to create a 4-input sorting network from the 3-input sorting network and so on.

4.2. *Representation and the proposed developmental scheme*

Sorting networks are encoded as sequences of pairs of integers. For instance, as Figure 1 shows, the 3-input sorting network is represented by the sequence of pairs (0, 1)(1, 2)(0, 1) indicating the ordering of compare–swap operations over the inputs 0, 1 and 2. A constructor is a sequence of instructions, each of which is encoded as three integers—operational code, argument 1 and argument 2. The representation is similar to *linear* structures for genetic programming [3]. Two types of instruction are utilized: *copy* and *modify*. Table 2 introduces their semantics, variants, operational codes and parameters. The Modify instructions read the indices of inputs of a comparator and add the values of their arguments to them.

Table 2. Instruction set utilized in development. "*mod*" denotes the modulo operation.

Instruction	arg1	arg2	description
0: ModifyS	a	b	$c1 = (c1 + a)$: mod w, $c2 = (c2 + b)$ mod w, $cp = cp + 1$, $np = np + 1$
1: ModifyM	a	b	$c1 = (c1 + a)$ mod w, $c2 = (c2 + b)$ mod w, $cp = cp + 1$, $ep = ep + 1$, $np = np + 1$
2: CopyS	k	–	copy $w - k$ comparators, $cp = cp + 1$, $np = np + w - k$
3: CopyM	k	–	copy $w - k$ comparators, $cp = cp + 1$, $ep = ep + w - k$, $np = np + w - k$

Modulo-operation ensures that the created comparator remains inside the sorting network of a given number of inputs. This type of instructions may be considered as a "shift" of a comparator to another position preserving the ordering of comparators. The Copy instructions copy some comparators (beginning from the actual one) to the next instance. The number of comparators to be copied depends on the instruction argument and the number of inputs of the sorting network being created. The instruction ModifyS (resp. CopyS) differs from ModifyM (resp. CopyM) in handling the *ep* pointer. Note that we (as designers) designed these instructions for this particular task. Hence we call the approach an *application-specific development*.

A sequence representing sorting networks is implemented using a variable-length array. A sequence representing the constructor is implemented as a constant-length array. Its size is determined at the beginning of evolution using our previous experience [35]. This size is not optimized.

Let $c1$ and $c2$ (i.e. the pair $(c1,c2)$) denote indices of inputs of a comparator in embryo that is processed by an instruction from Table 2. Instructions utilize three pieces of information: (1) operation codes and (2) argument values given by GA, and (3) w, which is the number of inputs (width) of the currently constructed sorting network. This value must be inserted into the developmental process externally (from environment). Three pointers are utilized in order to indicate the current position in sequences:

– *ep*—pointer to the actually processed comparator of the source sorting network (embryo pointer),
– *np*—pointer to the first free position in the created sorting network, where the newly created comparator will be placed (next-position pointer), and
– *cp*—constructor pointer (actually executed instruction).

As Figure 6 shows, instructions of the constructor are sequentially executed processing the comparator pointed by the embryo pointer (*ep*). The comparators of the embryo are also processed sequentially. Before execution of the first instruction, an auxiliary variable (*e_end*) is initialized by the value of *np*. This auxiliary value marks the end of embryo and is invariable during actual application of the constructor. The process of construction terminates when either all instructions of the constructor are executed or the end of embryo is reached (i.e. $ep = e\_end$). After a single application of the constructor the obtained sorting network is evaluated. If we apply the constructor again, we obtain a larger sorting network and so on. In such case, the pointers *ep* and *np* possess their values resulted from the

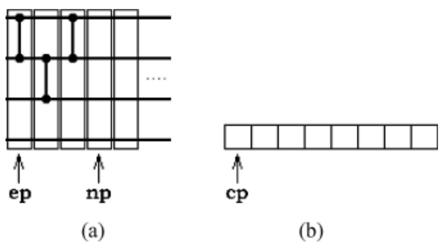

Figure 6 Initialization of the development: (a) growing sorting network and (b) chromosome, i.e. instructions in a constructor.

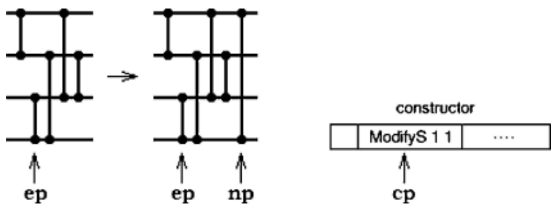

Figure 7 Example of invalid result of *Modify* instruction.

previous application; only *cp* and *e_end* are updated. Note that the sorting network obtained by repeated application of the constructor possesses all the comparators of its precursors.

The goal is to find such a constructor that will create valid sorting networks with the minimal number of comparators and/or delay. Because the delay of constructed sorting networks should be minimized, the following special condition has to be satisfied in order to execute a Modify instruction: the result of Modify instruction is valid only in case that $c1 < c2$ holds for the created comparator. Otherwise, the new comparator is not included in the sorting network and the instruction only updates the embryo pointer. Figure 7 shows an example of invalid result of Modify instruction. Pointer *ep* determines a comparator that will be used to create a comparator at position specified by *np*. However, the comparator is redundant. If accepted, the redundancy will propagate to larger sorting networks, which will be ineffective too.

4.3. An example of two steps of development

Figure 8 shows an example of two applications of a constructor. The horizontal sequence of numbers denotes the comparator positions. The vertical sequence of numbers denotes indices of inputs of sorting network. A rectangle surrounds the embryo. The vertical thin line separates the comparators created in the second application of the constructor. $ep1 = 0$ denotes the comparator pointed by embryo pointer, $np1 = 3$ denotes next-position pointer and $end1 = 3$ denotes the end of embryo before the first application of the constructor. Similarly, $ep2 = 3$ denotes the comparator pointed by embryo pointer, $np2 = 8$ denotes next-position pointer and $end2 = 8$ denotes the end of embryo before the second application of the constructor.

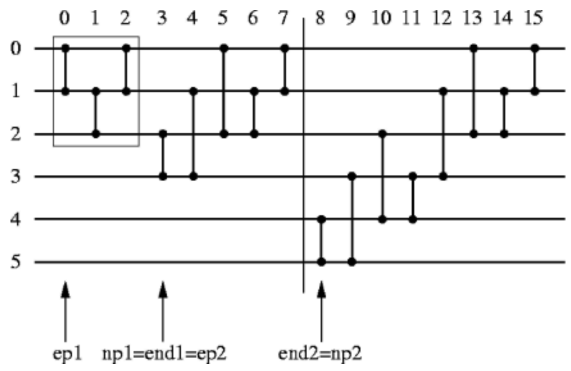

Figure 8 Example of the construction of sorting networks using constructor [ModifyS 2 2][ModifyS 1 2][ModifyM 0 1][ModifyS 2 1][CopyM 3 1][CopyM 2 4].

After execution of instructions [ModifyS 2 2] and [ModifyS 1 2], comparators (2,3) and (1,3) are created in positions 3 and 4 (using the comparator (0,1) at the position 0). The embryo pointer (*ep*) remains unchanged and $np = 5$. Execution of [ModifyM 0 1] results in creating comparator (0,2) at the position 5. Now, $ep = 1$ and $np = 6$. By applying [ModifyS 2 1] on comparator (1,2) we obtain a new comparator (3,3). However, such the comparator does not satisfy $c1 < c2$ condition and hence it will not be included in the sorting network. *ep* and *np* remain unchanged. [CopyM 3 1] instruction copies one comparator from the position 1 to the position pointed by $np = 6$ (since we are creating a 4-input sorting network and the first argument of CopyM instruction is 3, the 4–3 results in 1 comparator to be copied — see Table 2). The instruction updates the pointers, so now $ep = 2$ and $np = 7$. The [CopyM 2 4] should copy two comparators. Since there is only one comparator before the end of embryo, only one comparator will be copied and the pointers will be updated to $ep = 3$ and $np = 8$. Because the end of embryo was reached and all the instructions of the constructor were executed, the first application is finished.

The *ep* and *np* pointers now possess the values of *ep*2 and *np*2 and this is the starting configuration for the second application of the constructor. Execution of instructions proceeds in the same manner. Comparators will be created in positions 8–15. Note that during the second application of the constructor the result of [ModifyS 2 1] is valid and the comparator (3,4) will be created in position 11 from (1,3) in position 4. Since we are now creating a 6-input sorting network, [CopyM 3 1] copies three comparators from the positions 4, 5 and 6. The last instruction [CopyM 2 4] copies one comparator from the position 7 before the end of the second embryo and the second application of the constructor is finished. The next applications would construct the 8-, 10-, 12-input sorting networks and so on.

4.4. *Genetic algorithm*

A steady state genetic algorithm and a simple genetic algorithm implemented using Galib [40] have been utilized. The GA operates with constant-length chromosomes (programs) represented by triplets of positive integers. Initial population is generated randomly. The probabilities of uniform crossover and mutation and other parameters will be given together with the results in Section 5. The mutation operator is applied on all offspring.

We would like to evolve arbitrarily large sorting networks. However, because of problems with the scalability of fitness evaluation, only several instances of the growing sorting network can be evaluated in the fitness calculation process. Assume that we start with a 3-input sorting network. In our case a candidate constructor is used to build the 4-input, 5-input, 6-input and 7-input sorting networks from the 3-input embryo. The fitness value is calculated as follows:

$$\mathit{fitness} = f(4) + f(5) + f(6) + f(7),$$

where *f*(*j*) is the fitness value for a *j*-input sorting network. This value is calculated using the zero–one principle as the number of input sequences of zeroes and ones sorted correctly. Hence $2^4 + 2^5 + 2^6 + 2^7 = 240$ represents the best possible value that we could obtain. At the end of evolution we have to test whether the evolved constructor is *general*, i.e. whether it generates infinitely large sorting networks which sort all possible input sequences. If a constructor is able to create a sorting network for a sufficiently high N ($N = 28$ in our case) then we consider the constructor as general.

The proposed developmental scheme can fully be defined using the following parameters: $w1$, *w_max*, *dw* and *ew*, which will be utilized to characterize the results in Section 5. Let $w1$ denote the number of inputs of the smallest sorting network that is constructed from *ew*-input embryo in the fitness calculation process (i.e. the sorting network created by the first application of constructor). Similarly *w_max* denotes the largest sorting network constructed during fitness evaluation. Let *dw* be a difference between the number of inputs of neighboring networks created by a constructor. In this paper, *dw* is 1 or 2. Finally, it is useful to define one more parameter, *de*, $de = w1 - ew$. The following parameters summarize the mentioned example: $w1 = 4$, $w\_max = 7$, $dw = 1$, and $ew = 3$.

5. Experimental results

This section summarizes the experiments that we performed. Each experiment required setting up parameters of genetic algorithm (the probability of crossover and mutation, population size, the number of generations, etc.) and parameters of development ($w1$, *w_max*, *dw*, and *ew*). The quality of resulting sorting networks depends on both sets of parameters. We measured the *number of general constructors* (NGC) obtained out of 100 independent runs.

The produced sorting networks will be characterized in terms of comparators count and delay. Each constructor will be labeled by its length (the number of instructions), size of utilized embryo and identification. Moreover, we recognized that very interesting sorting networks are produced in the case that only even-input (or odd-input) networks are required. Hence constructors were evolved for the even, odd, and even and odd[1] number of inputs in growing sorting networks, which is also included in the label as seen in Table 3.

Three tables will summarize each experiment. The first table lists the best constructors. The second table gives the number of compare–swap components and the number of redundant comparators (in parentheses). Delay and the number of parallel layers in parentheses (that are available after removal of redundant comparators) are given in the third table.

Table 3. Definition of labels for constructors in the form *gX-Yzzz_ID*.

Symbols	Description
X	Constructor length (the number of instructions)
Y	Embryo width (the number of inputs)
zzz	Odd/even/all (possible inputs)
ID	Identification

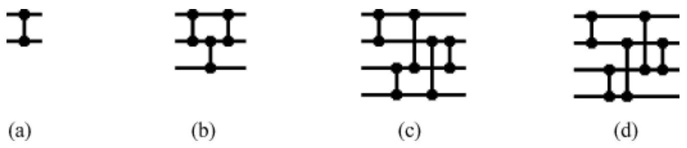
Figure 9. Embryos tested: (a) 2-input, (b) 3-input, (c) 4-input, (d) 4-input — another type.

The best solution is typed *italic*. We experimented with various types of embryo. Figure 9 shows the embryos that we utilized.

5.1. Evolving sorting networks

In the first set of experiments, the sorting networks with the even as well as odd number of inputs were evolved from a three-input embryo. It corresponds to setting: $ew = 3$, $de = 1$ and $dw = 1$. We used a simple GA, operating with 60 individuals, with the probability of crossover $p_c = 0.75$ and the probability of mutation $p_m = 0.08$. Results are summarized in Tables 4–6.

The evolved constructors are very simple and of the same quality as the conventional approach produces. In fact the conventional straight insertion algorithm has been rediscovered (see Figure 10). Some other examples are given in Figure 11. We were not able to improve the principle of construction in this way. Hence we have tried to change parameters of the development and GA as the next section illustrates.

Table 4. Examples of general constructors evolved for a 3-input embryo.

Constructor	Instructions	NGC
g3-3all	*[ModifyS 2 2] [ModifyS 1 1] [CopyM 3 2]*	
g3-3all_2	[ModifyS 1 1] [ModifyM 2 2] [CopyM 0 2]	100
g3-3all_3	[ModifyS 2 2] [ModifyS 1 1] [CopyM 0 3]	
g4-3all	[ModifyS 3 2] [ModifyS 2 2] [ModifyS 1 1] [CopyM 3 3]	100
g4-3all_2	[ModifyM 0 0] [ModifyS 1 1] [ModifyS 1 0] [CopyM 0 2]	

Parameters: $ew = 3$, $de = 1$, $dw = 1$.

Table 5. The number of comparators of sorting networks for constructors from Table 4. The number of redundant comparators is given in parentheses.

N	4	5	6	7	8	9	10	11	12	13	14	15
conv.	6	10	15	21	28	36	45	55	66	78	91	105
g3-3all	*6*	*10*	*15*	*21*	*28*	*36*	*45*	*55*	*66*	*78*	*91*	*105*
g3-3all_2	7 (1)	12 (2)	18 (3)	25 (4)	33 (5)	42 (6)	52 (7)	63 (8)	75 (9)	88 (10)	102 (11)	117 (12)
g3-3all_3	8 (2)	14 (4)	21 (6)	29 (8)	38 (10)	48 (12)	59 (14)	71 (16)	84 (18)	98 (20)	113 (22)	129 (24)
g4-3all	6	10	15	21	28	36	45	55	66	78	91	105
g4-3all_2	7 (1)	12 (2)	18 (3)	25 (4)	33 (5)	42 (6)	52 (7)	63 (8)	75 (9)	88 (10)	102 (11)	117 (12)
N	16	17	18	19	20	21	22	23	24	25	26	27
conv.	120	136	153	171	190	210	231	253	276	300	325	351
g3-3all	*120*	*136*	*153*	*171*	*190*	*210*	*231*	*253*	*276*	*300*	*325*	*351*
g3-3all_2	133 (13)	150 (14)	168 (15)	187 (16)	207 (17)	228 (18)	250 (19)	273 (20)	297 (21)	322 (22)	348 (23)	375 (24)
g3-3all_3	146 (26)	164 (28)	183 (30)	203 (32)	224 (34)	246 (36)	269 (38)	293 (40)	318 (42)	344 (44)	371 (46)	399 (48)
g4-3all	120	136	153	171	190	210	231	253	276	300	325	351
g4-3all_2	133 (13)	150 (14)	168 (15)	187 (16)	207 (17)	228 (18)	250 (19)	273 (20)	297 (21)	322 (22)	348 (23)	375 (24)

Table 6. Delay of sorting networks from Table 4. Parentheses show delay after removal of redundant comparators.

N	4	5	6	7	8	9	10	11	12	13	14	15
conv.	5	7	9	11	13	15	17	19	21	23	25	27
g3-3all	*5*	*7*	*9*	*11*	*13*	*15*	*17*	*19*	*21*	*23*	*25*	*27*
g3-3all_2	7 (5)	11 (7)	15 (9)	19 (11)	23 (13)	27 (15)	31 (17)	35 (19)	39 (21)	43 (23)	47 (25)	51 (27)
g3-3all_3	7 (5)	11 (7)	15 (9)	19 (11)	23 (13)	27 (15)	31 (17)	35 (19)	39 (21)	43 (23)	47 (25)	51 (27)
g4-3all	5	7	9	11	13	15	17	19	21	23	25	27
g4-3all_2	6 (5)	9 (7)	12 (9)	15 (11)	18 (13)	21 (15)	24 (17)	27 (19)	30 (21)	33 (23)	36 (25)	39 (27)
N	16	17	18	19	20	21	22	23	24	25	26	27
conv.	29	31	33	35	37	39	41	43	45	47	49	51
g3-3all	*29*	*31*	*33*	*35*	*37*	*39*	*41*	*43*	*45*	*47*	*49*	*51*
g3-3all_2	55 (29)	59 (31)	63 (33)	67 (35)	71 (37)	75 (39)	79 (41)	83 (43)	87 (45)	91 (47)	95 (49)	99 (51)
g3-3all_3	55 (29)	59 (31)	63 (33)	67 (35)	71 (37)	75 (39)	79 (41)	83 (43)	87 (45)	91 (47)	95 (49)	99 (51)
g4-3all	29	31	33	35	37	39	41	43	45	47	49	51
g4-3all_2	42 (29)	45 (31)	48 (33)	51 (35)	54 (37)	57 (39)	60 (41)	63 (43)	66 (45)	69 (47)	72 (49)	75 (51)

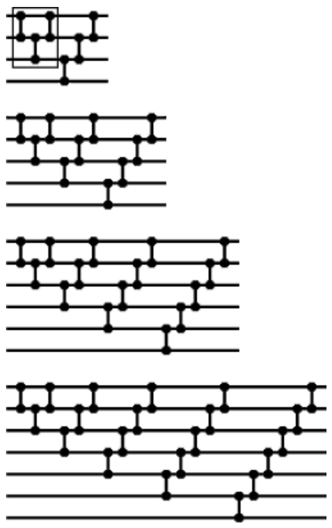

Figure 10 The insertion principle rediscovered using instructions: [ModifyS 2 2] [ModifyS 1 1] [CopyM 3 2] or [ModifyS 3 2] [ModifyS 2 2] [ModifyS 1 1] [CopyM 3 3].

5.2. *Evolving odd-input sorting networks*

The constructed sorting networks were restricted to the odd number of inputs. Surprisingly, the most interesting odd-input sorting networks were generated by using an even-input embryo. We chose a 4-input embryo, $ew = 4$, and parameters $de = 1$ and $dw = 2$. After some experiments, the best results were produced by a steady-state genetic algorithm with $p_c = 0.74$ and $p_m = 0.1$. Population consists of 400 individuals with overlapping 12 individuals. Table 7 shows chromosomes of some evolved constructors.[2] As Table 8 indicates, we were able to reduce the number of comparators substantially in this set of experiments. Delays are given in Table 9.

If the number of comparators is measured then the best-evolved sorting network is given in Figure 12. In case of minimizing the delay, the best solution is shown in Figure 13. However, all the sorting networks contain redundant comparators which make their delay unnecessarily long. After their removal we can obtain the quality (delay) of the conventional solution.

5.3. *Evolving even-input sorting networks*

In the previous section we discovered better constructors than the conventional approach offers for the odd-input sorting networks. This section deals with discovered even-input sorting networks that are better than conventional ones.

In contrast to the previous section, various types of embryos have been confirmed as useful for constructing novel sorting networks. We applied a simple genetic algorithm with $p_c = 0.7$, $p_m = 0.023$ and population size 60. Tables 10–12 summarize the results for the two-input embryo.

Table 7. Constructors of odd-input sorting networks for a four-input embryo.

Constructor	Instructions	NGC
g8-4odd	[0 2 2] [0 2 3] [1 3 3] [0 1 1] [0 4 0] [3 2 3] [3 0 4] [3 1 3]	41
g8-4odd_2	[0 2 2] [0 2 3] [1 3 3] [0 1 1] [3 4 2] [3 2 2] [3 2 2] [3 4 4]	
g8-4odd_3	[0 2 2] [0 2 3] [0 3 3] [1 2 0] [0 1 1] [3 0 4] [0 3 3] [3 3 3]	
g8-4odd_4	*[0 2 2] [0 3 3] [0 2 2] [0 1 1] [0 2 2] [3 0 0] [3 3 2] [3 0 0]*	
g7-4odd	[0 2 2] [0 3 2] [1 2 3] [0 3 2] [0 1 1] [3 0 1] [3 0 3]	62
g7-4odd_2	[0 2 2] [1 2 3] [0 3 2] [0 1 1] [0 2 1] [3 0 2] [3 4 0]	
g7-4odd_3	[0 2 2] [1 2 3] [0 1 1] [0 3 2] [0 1 1] [3 3 2] [3 2 3]	
g6-4odd	[0 2 2] [0 2 3] [0 3 3] [1 1 2] [3 2 1] [3 3 3]	80
g6-4odd_2	*[0 2 2] [1 2 3] [0 3 2] [0 1 1] [3 1 3] [3 3 4]*	

Parameters: $ew = 4$, $de = 1$, $dw = 2$.

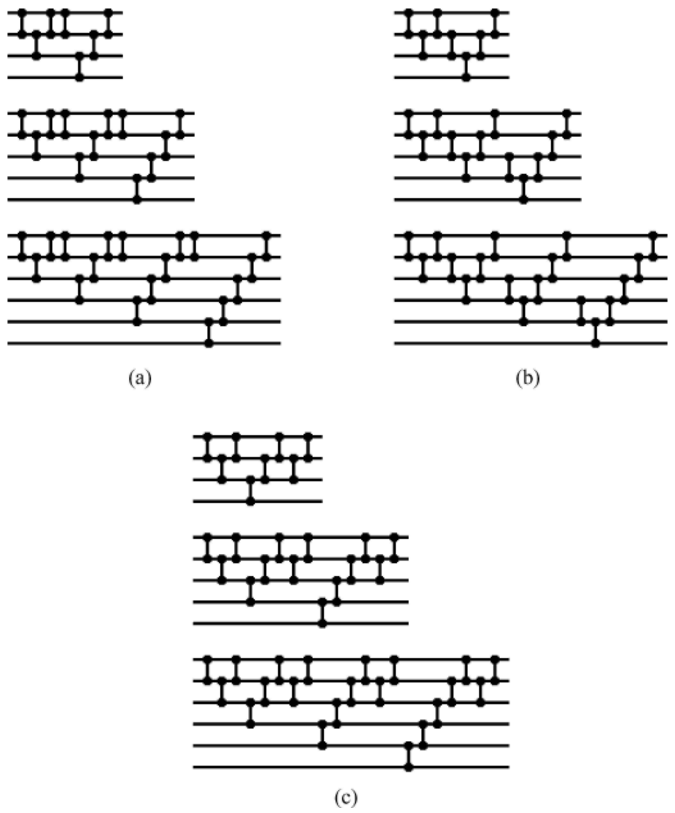

Figure 11 Examples of growing sorting networks created using constructors: (a) g4-3all_2, (b) g3-3all_2, (c) g3-3all_3.

Table 8. The number of comparators for odd-input sorting networks created using constructors from Table 7.

N	5	7	9	11	13	15	17	19	21	23	25	27
conv.	10	21	36	55	78	105	136	171	210	253	300	351
g8-4odd	14 (5)	26 (8)	41 (11)	59 (14)	80 (17)	104 (20)	131 (23)	161 (26)	194 (29)	230 (32)	269 (35)	311 (38)
g8-4odd_2	13 (4)	24 (6)	38 (8)	55 (10)	75 (12)	98 (14)	124 (16)	153 (18)	185 (20)	220 (22)	258 (24)	299 (26)
g8-4odd_3	13 (4)	24 (6)	39 (9)	58 (13)	81 (18)	108 (24)	139 (31)	174 (39)	213 (48)	256 (58)	303 (69)	354 (81)
g8-4odd_4	15 (6)	30 (10)	50 (15)	75 (21)	105 (28)	140 (36)	180 (45)	225 (55)	275 (66)	330 (78)	390 (91)	455 (105)
g7-4odd	12 (3)	22 (4)	35 (5)	51 (6)	70 (7)	92 (8)	117 (9)	145 (10)	176 (11)	210 (12)	247 (13)	287 (14)
g7-4odd_2	12 (3)	23 (5)	38 (8)	57 (12)	80 (17)	107 (23)	138 (30)	173 (38)	212 (47)	255 (57)	302 (68)	353 (80)
g7-4odd_3	13 (4)	25 (7)	41 (11)	61 (16)	85 (22)	113 (29)	145 (37)	181 (46)	221 (56)	265 (67)	313 (79)	365 (92)
g6-4odd	13 (4)	24 (6)	38 (8)	55 (10)	75 (12)	98 (14)	124 (16)	153 (18)	185 (20)	220 (22)	258 (24)	299 (26)
g6-4odd_2	*12 (3)*	*22 (4)*	*35 (5)*	*51 (6)*	*70 (7)*	*92 (8)*	*117 (9)*	*145 (10)*	*176 (11)*	*210 (12)*	*247 (13)*	*287 (14)*

Table 9. Delay of odd-input sorting networks created using constructors from Table 7.

N	5	7	9	11	13	15	17	19	21	23	25	27
conv.	7	11	15	19	23	27	31	35	39	43	47	51
g8-4odd	11 (6)	18 (12)	25 (17)	32 (22)	39 (27)	46 (32)	53 (37)	60 (42)	67 (47)	74 (52)	81 (57)	88 (62)
g8-4odd_2	10 (6)	16 (12)	22 (17)	28 (22)	34 (27)	40 (32)	46 (37)	52 (42)	58 (47)	64 (52)	70 (57)	76 (62)
g8-4odd_3	10 (6)	16 (12)	22 (17)	28 (22)	37 (27)	45 (32)	53 (37)	61 (42)	70 (47)	80 (52)	90 (57)	100 (62)
g8-4odd_4	*11 (6)*	*19 (11)*	*27 (15)*	*35 (19)*	*43 (23)*	*51 (27)*	*59 (31)*	*67 (35)*	*75 (39)*	*83 (43)*	*91 (47)*	*99 (51)*
g7-4odd	9 (6)	14 (12)	19 (17)	24 (22)	29 (27)	34 (32)	39 (37)	44 (42)	49 (47)	54 (52)	59 (57)	64 (62)
g7-4odd_2	9 (6)	16 (12)	23 (17)	30 (22)	37 (27)	44 (32)	51 (37)	58 (42)	65 (47)	72 (52)	79 (57)	86 (62)
g7-4odd_3	10 (6)	17 (12)	24 (16)	31 (20)	38 (24)	45 (28)	52 (32)	59 (36)	66 (40)	73 (44)	80 (48)	87 (52)
g6-4odd	10 (6)	16 (12)	22 (17)	28 (22)	34 (27)	40 (32)	46 (37)	52 (42)	58 (47)	65 (52)	70 (57)	76 (62)
g6-4odd_2	9 (6)	14 (12)	19 (17)	24 (22)	29 (27)	34 (32)	39 (37)	44 (42)	49 (47)	54 (52)	59 (57)	64 (62)

Table 10. Constructors of even-input sorting networks utilizing a two-input embryo.

Constructor	Instructions	NGC
g9-2even	[0 2 2] [0 1 2] [0 0 1] [1 1 1] [0 4 4] [3 3 2] [3 1 1] [1 1 2] [2 1 0]	14
g8-2even	[0 2 2] [0 0 1] [0 1 2] [1 1 1] [3 0 2] [0 1 3] [3 0 0] [3 2 3]	25
g8-2even_2	[0 2 2] [0 1 2] [0 0 1] [1 1 1] [0 4 4] [3 0 1] [3 4 1] [1 2 3]	
g6-2even	[0 2 2] [0 1 1] [0 0 2] [0 2 2] [3 0 4] [3 0 0]	73
g6-2even_2	*[0 2 2] [0 1 2] [0 0 1] [1 1 1] [3 1 2] [3 1 1]*	

(Parameters: *ew* = 2, *de* = 2, *dw* = 2.)

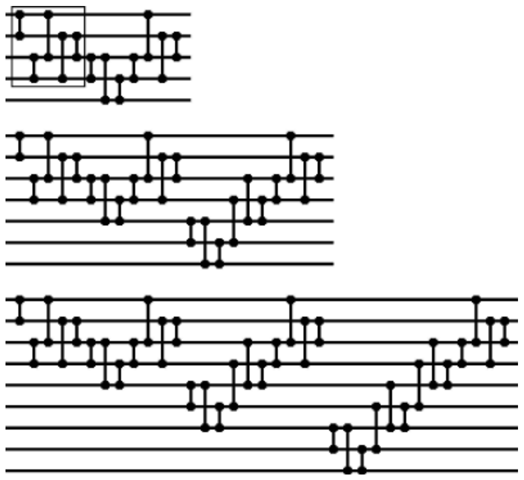

Figure 12 Comparator-efficient odd-input sorting networks created by means of the constructor *g6-4odd_2*. The embryo is marked.

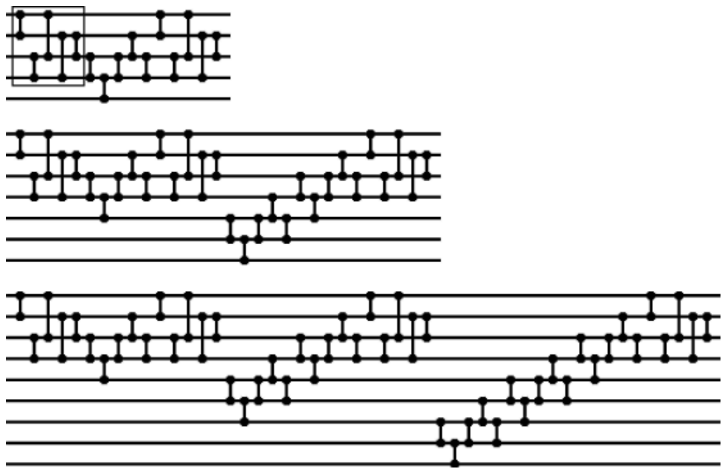

Figure 13 Delay-efficient odd-input sorting networks created by means of the constructor *g8-4odd_4*.

Table 11. The number of comparators of even-input sorting networks created from a two-input embryo using constructors given in Table 10.

N	4	6	8	10	12	14	16	18	20	22	24	26	28
conv.	6	15	28	45	66	91	120	153	190	231	276	325	378
g9-2even													
g8-2even_2	*5*	*12*	*22*	*35*	*51*	*70*	*92*	*117*	*145*	*176*	*210*	*247*	*287*
g6-2even_2													
g8-2even	5	12	22	35	51	71	95	123	155	191	231	275	323
	(0)	(0)	(0)	(0)	(0)	(1)	(3)	(6)	(10)	(15)	(21)	(28)	(36)
g6-2even	6	15	28	45	66	91	120	153	190	231	276	325	378

Table 12. Delay of even-input sorting networks created from a two-input embryo using constructors given in Table 10.

N	4	6	8	10	12	14	16	18	20	22	24	26	28
conv.	5	9	13	17	21	25	29	33	37	41	45	49	53
g9-2even													
g8-2even_2	*3*	*7*	*11*	*15*	*19*	*23*	*27*	*31*	*35*	*39*	*43*	*47*	*51*
g6-2even_2													
g8-2even	3	7	11	15	19	23	28	34	39	45	51	57	63
	(3)	*(7)*	*(11)*	*(15)*	*(19)*	*(23)*	*(27)*	*(31)*	*(35)*	*(39)*	*(43)*	*(47)*	*(51)*
g6-2even	3	7	11	15	19	23	27	31	35	39	43	47	51

As Figure 14 shows, the optimal 4-input embryo was created from a 2-input embryo after the first step of development.

The *g8-4even_2* is one of the best constructors we have ever evolved. This constructor uses a four-input embryo and produces sorting networks with a better comparator count and delay than the best conventional solution. However, it contains redundant comparators that have to be removed. Examples of constructors evolved from the 4-input embryo (including *g8-4even_2*) are given in Table 13. Other parameters are summarized in Tables 14 and 15. Sorting networks created using the best constructors are shown in Figures 15 and 16.

Table 13. Constructors of even-input sorting networks utilizing a four-input embryo.

Constructor	Instructions	NGC
g8-4even	[1 4 1] [0 0 1] [0 2 2] [0 0 1] [1 1 2] [0 3 2] [3 3 0] [3 3 2]	
g8-4even_2	*[1 4 4] [1 2 1] [0 4 3] [0 2 2] [0 3 3] [3 4 1] [0 2 2] [3 1 3]*	41
g8-4even_3	[0 2 2] [0 4 4] [0 3 4] [1 2 3] [1 2 0] [3 4 2] [0 2 2] [3 4 4]	
g7-4even	[1 4 4] [1 2 2] [0 2 2] [0 3 3] [0 3 2] [3 2 0] [3 3 3]	46

Parameters: *ew*=4, *de*=2, *dw*=2.

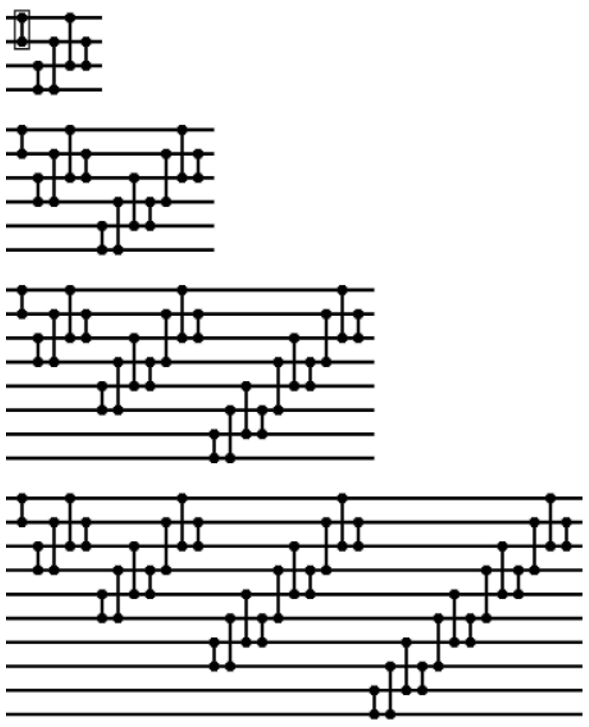

Figure 14 The most comparator-efficient as well as delay-efficient even-input sorting networks created from a two-input embryo using constructors g9-2even, g8-2even_2 or g6-2even_2.

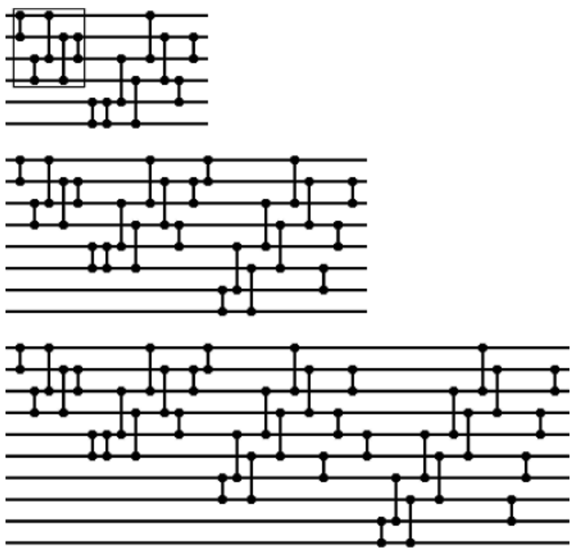

Figure 15 Efficient even-input sorting networks created using the constructor g8-4even_2.

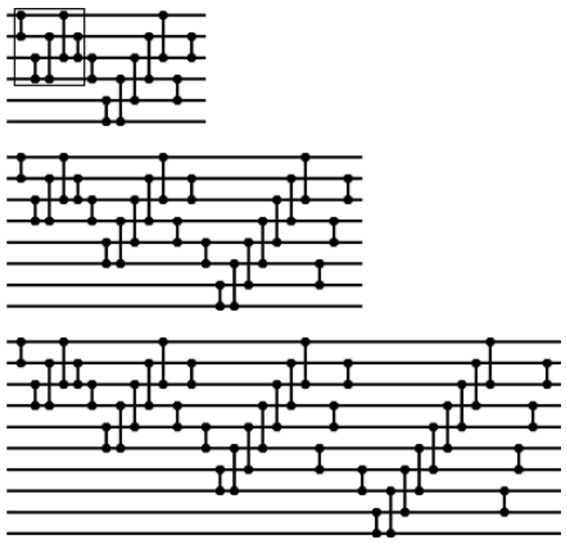

Figure 16 Efficient even-input sorting networks created using the constructor g8-4even_3.

Table 14. The number of comparators of even-input sorting networks created using a four-input embryo by means of constructors given in Table 13.

N	6	8	10	12	14	16	18	20	22	24	26	28
conventional	15	28	45	66	91	120	153	190	231	276	325	378
g8-4even	13	24	38	55	75	98	124	153	185	220	258	299
g8-4even_2	*13 (1)*	*24 (2)*	*38 (3)*	*55 (4)*	*75 (5)*	*98 (6)*	*124 (7)*	*153 (8)*	*185 (9)*	*220 (10)*	*258 (11)*	*299 (12)*
g8-4even_3	13 (1)	24 (2)	38 (3)	55 (4)	75 (5)	98 (6)	124 (7)	153 (8)	185 (9)	220 (10)	258 (11)	299 (12)
g7-4even	13 (1)	24 (2)	38 (3)	55 (4)	75 (5)	98 (6)	124 (7)	153 (8)	185 (9)	220 (10)	258 (11)	299 (12)

Table 15. Delay of even-input sorting networks created using a four-input embryo by means of constructors given in Table 13.

N	6	8	10	12	14	16	18	20	22	24	26	28
conventional	9	13	17	21	25	29	33	37	41	45	49	53
g8-4even	9	15	21	27	33	39	45	51	57	63	69	75
g8-4even_2	*6 (6)*	*9 (9)*	*14 (12)*	*19 (15)*	*23 (18)*	*26 (21)*	*31 (24)*	*36 (27)*	*41 (30)*	*46 (33)*	*51 (36)*	*56 (39)*
g8-4even_3	7 (6)	12 (9)	17 (12)	22 (15)	27 (18)	32 (21)	37 (24)	42 (27)	47 (30)	52 (33)	57 (36)	62 (39)
g7-4even	7 (7)	11 (11)	16 (15)	20 (19)	24 (23)	28 (27)	33 (31)	37 (35)	41 (39)	45 (43)	49 (47)	53 (51)

We evolved two interesting constructors by using a three-input embryo. They are not as good as the constructors utilizing a four-input embryo. However, they still produce better results than the conventional approach (see Tables 16–18). Examples of sorting networks are given in Figure 17.

Table 16. Constructors of even-input sorting networks utilizing a three-input embryo.

Constructor	**Instructions**	**NGC**
g6-3even	*[0 2 2] [0 1 2] [1 0 1] [0 2 1] [3 3 1] [3 2 4]*	59
g6-3even_2	[0 2 2] [0 1 2] [0 0 1] [1 1 1] [3 4 4] [3 0 1]	

Parameters: *ew*=3, *de*=1, *dw*=2.

Table 17. The number of comparators of even-input sorting networks created using a three-input embryo by means of constructors given in Table 16.

N	4	6	8	10	12	14	16	18	20	22	24	26	28
Conventional	6	15	28	45	66	91	120	153	190	231	276	325	378
g6-3even	*8 (2)*	*16 (3)*	*27 (4)*	*41 (5)*	*58 (6)*	*78 (7)*	*101 (8)*	*127 (9)*	*156 (10)*	*188 (11)*	*223 (12)*	*261 (13)*	*302 (14)*
g6-3even_2	9 *(3)*	18 *(5)*	30 *(7)*	45 *(9)*	63 *(11)*	84 *(13)*	108 *(15)*	135 *(17)*	165 *(19)*	198 *(21)*	234 *(23)*	273 *(25)*	315 *(27)*

Table 18. Delay of even-input sorting networks created using a three-input embryo by means of constructors given in Table 16.

N	4	6	8	10	12	14	16	18	20	22	24	26	28
Conventional	5	9	13	17	21	25	29	33	37	41	45	49	53
g6-3even	*6 (5)*	*10 (9)*	*14 (13)*	*18 (17)*	*22 (21)*	*26 (25)*	*30 (29)*	*34 (33)*	*38 (37)*	*42 (41)*	*46 (45)*	*50 (49)*	*54 (53)*
g6-3even_2	7 *(5)*	12 *(9)*	17 *(13)*	22 *(17)*	27 *(21)*	32 *(25)*	37 *(29)*	42 *(33)*	47 *(37)*	52 *(41)*	57 *(45)*	62 *(49)*	67 *(53)*

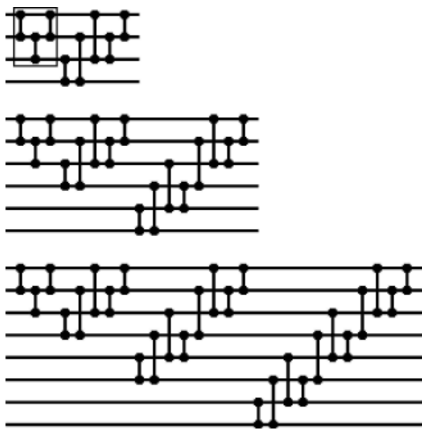

Figure 17 Even-input sorting networks created using the constructor g6-3even.

5.4. Improving odd-input sorting networks

The presented evolutionary approach produced sorting networks with better implementation cost (the number of comparators) than the conventional approach for even-input as well as odd-input sorting networks. Delay of even-input sorting networks was also improved.

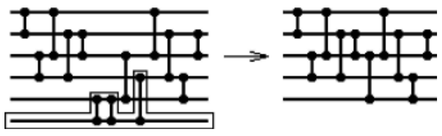

Figure 18 Creating delay efficient odd-input sorting networks from even-input sorting networks by removing the bottom line of comparators. The original six-input sorting network: (0,1) (2,3) (0,2) (1,3) (1,2) (4,5) (4,5) (2,4) (3,5) (0,2) (1,3) (3,4) (1,2). The new five-input sorting network: (0,1) (2,3) (0,2) (1,3) (1,2) (2,4) (0,2) (1,3) (3,4) (1,2).

However, in case of odd-input sorting networks, none of the presented constructors is better than a conventional one in terms of delay.

We have discovered that the best-known constructor for even-input sorting networks (*g8-4even_2*) can be utilized to improve delay in case of odd-input networks. Figure 18 shows that by removing the bottom line together with "connected" comparators, the odd-input sorting network is established. We verified the improvement of created sorting networks for $N \leq 29$.

5.5. *Computational effort*

More than 10,000 independent runs of evolutionary algorithm were performed. The number of generations needed for gaining a solution varies from about 150 to many thousands. We have found out the limit 10,000 generations to be sufficient to get some solutions in a reasonable time. If the evolution does not terminate successfully within this limit, the evolutionary process is stopped.

Consider even-input sorting networks constructed from a 2-input embryo. In this case, 58% of independent runs of evolutionary process terminated successfully. The average number of generations is 2053. Figure 19 shows a typical example of the progress of average fitness of the population along with the rise of fitness value of the best individual during evolution. This experiment worked with a simple genetic algorithm, the crossover probability 0.7, the mutation probability 0.023 and for population size of 60 individuals. The fitness function considered four developmental steps, i.e. the maximum fitness value was $f_{\max} = f(4) + f(6) + f(8) + f(10) = 2^4 + 2^6 + 2^8 + 2^{10} = 1376$, where $f(n)$ is the number of all possible sequences of zeroes and ones of n-input sorting network.

5.6. *Summary of results for each category*

Sorting networks with complete inputs: It is easy to evolve a general constructor in this category. We rediscovered the principle of the straight insertion algorithm. However, sorting of large data sets is not efficient in this way because many comparators are required.

Odd-input sorting networks: Some constructors were evolved that produce smaller sorting networks in terms of a comparator count than the conventional insertion and selection method can offer. However, it works only using a four-input embryonic network. The next improvement can be done by removing redundant comparators that are often generated by the constructors. We were not able to improve delay in this category — the best constructor has reached the quality of a conventional method. Surprisingly, it is possible to modify the

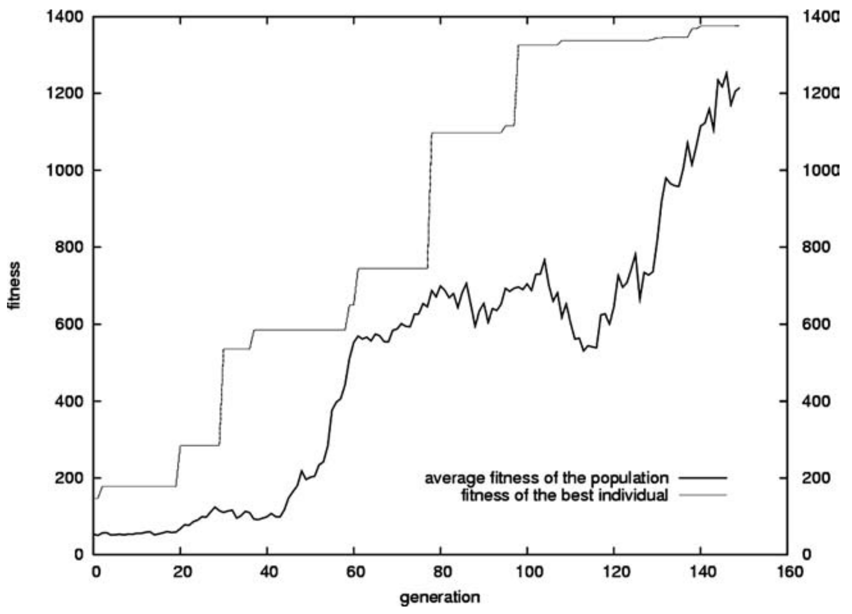

Figure 19 The best and average fitness value in a typical run of a simple GA for the following settings: $ew = 2$, $de = 2$, $dw = 2$, $p_c = 0.7$, $p_m = 0.023$, 60 individuals in population, 4 developmental steps for fitness calculation ($f_{\max} = 1376$).

best even-input sorting networks in order to obtain odd-input sorting networks whose delay is shorter than delay of conventional networks.

Even-input sorting networks: In this category various types of embryos have generated interesting results. The usage of the two-input embryo has led to a substantial reduction of the number of comparators and a small reduction of delay. The constructors evolved from a two-input embryo did not produce redundant comparators. On the other hand, the constructors g8-4even_2 and g8-4even_3, evolved using a four-input embryo, minimize the number of comparators as well as delay substantially. However, first, it is necessary to remove redundant comparators from the created networks. These constructors represent the main contribution of this paper.

6. Discussion

We clearly demonstrated that the proposed evolutionary method combined with development has improved the conventional design principle not only for a single instance, but for all instances of our problem. In order to illustrate the quality of evolved sorting networks, Table 19 compares the best evolved sorting networks with common sequential sorting algorithms — *BubbleSort* and *QuickSort*. We measured the mean number of comparisons for ten thousands randomly generated input sequences of length N. Under this criterion, the evolved sorting networks exhibit the best results.

All candidate constructors were evaluated using the zero–one principle; however, only for a limited number of inputs. We found this approach very efficient because about 50%

Table 19. Confrontation of evolved sorting networks (generated by means of constructors *g8-4even_2* or *g8-4even_3*) with conventional sorting algorithms *BubbleSort* and *QuickSort*. The table shows the mean number of comparisons required for sorting *N* elements.

N	3	4	5	6	7	8	9	10	11	12	13	14	15
Bsort	3	6	10	15	21	28	36	45	55	66	78	91	105
Qsort	9	15	24	33	42	52	63	74	85	96	108	120	132
SN	–	5	–	12	–	22	–	35	–	51	–	70	–
N	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
Bsort	120	136	153	171	190	210	231	253	276	300	325	351	378
Qsort	144	157	170	183	196	209	222	236	249	263	277	290	305
SN	92	–	117	–	145	–	176	–	210	–	247	–	287

of them are considered as "general" (see NGC parameter in the previous tables). Although we use the word "general" it is obvious that the evolved constructors have not to be really general—the verification method we applied (i.e. the evaluation of a constructor up to a sufficiently high *N*) is not a proof. Furthermore, the size of constructors was not optimized. Next research will be devoted to prove that the constructors are really *general* and minimal.

The main feature of the proposed developmental system for genetic algorithm is that a lot of problem-domain knowledge (such as the definition and use of copy and modify instructions) has been presented in its inductive bias. We do believe that the idea of evolving constructors for infinitely growing objects is generally applicable. However, it is difficult to define an embryo and appropriate domain knowledge for a particular problem. It seems that designing an efficient developmental system is as difficult as designing an efficient genetic algorithm for a given problem.

Except the instructions that we had to design for this particular application manually and that GA had to put them together to make a constructor, the proposed developmental scheme has utilized another information — the size of the currently constructed network *N*. This information is not a part of our artificial genetic code. Therefore, we can understand it as a property of environment, which surrounds a growing sorting network. It is obvious that as positional information is crucial for biological development [1], no correct sorting network can be created without a correct *N*. In real biological systems the interplay between a cell and its environment is very complex. In our system the interplay practically does not exist. A growing sorting network, for example, does not influence the value of *N* at all. We are planning to develop more complex models of development for this application in order to investigate whether the obtained results can be improved.

Further research should be devoted to specifying a hardware reconfigurable platform and applications that could benefit from growing sorting networks (or similar growing digital circuits). A circuit should grow when it is required and get smaller when is not needed. It will also be interesting to explore fault tolerance of growing sorting networks.

7. Conclusions

In this paper we described a method for constructing efficient larger sorting networks from smaller ones. First we rediscovered a conventional principle of straight insertion algorithm by means of genetic algorithm endowed with an application-specific development. Later, very efficient principles (programs) have been discovered by the same technique allowing us to reduce the size and delay of constructed odd/even-input sorting networks.

The reported research represents the rare case in which a new scalable principle is discovered by an evolutionary algorithm. In most cases, evolutionary algorithms are being used to find a single suitable solution.

We do believe that application-specific evolutionary algorithms endowed with application-specific developmental systems will allow designers to discover novel design principles for constructing some other arbitrarily large systems in near future.

Acknowledgments

The research was performed with the Grant Agency of the Czech Republic under No. 102/03/P004 *Evolvable hardware based application design methods*.

Notes

1. Odd and even is denoted as "all" in labels.
2. Operation codes are given instead of symbolic names according to Table 2.

References

1. B. Alberts, et al., Essential Cell Biology—An Introduction to the Molecular Biology of the Cell, Garland Publishing: New York, 1998.
2. T. Bäck, Evolutionary Algorithms in Theory and Practice, Oxford University Press: New York, Oxford, 1996.
3. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, Genetic Programming—An Introduction. Morgan Kaufmann Publishers: San Francisco, CA, 1998.
4. P. Bentley, (ed.), Evolutionary Design by Computers, Morgan Kaufmann Publisher, 1999.
5. P. Bentley, "Fractal proteins," Genetic Programming and Evolvable Machines, vol. 5, no. 1, pp. 71–101, 2004.
6. P. Bentley and D. W. Corne, (eds.), Creative Evolutionary Systems, Morgan Kaufmann, 2001.
7. E. J. W. Boers and H. Kuiper, "Biological Metaphors and the Design of Artificial Neural Networks. Master Thesis," Departments of Computer Science and Experimental and Theoretical Psychology, Leiden University, 1992.
8. S. Choi and B. Moon, "A hybrid genetic search for the sorting network problem with evolving parallel layers," in Genetic and Evolutionary Computation Conference, San Francisco, 2001, pp. 258–265.
9. S. Choi and B. Moon, "More effective genetic search for the sorting network problem," in Genetic and Evolutionary Computation Conference, New York, 2002, pp. 335–342.
10. S. Choi and B. Moon, "Isomorphism, normalization, and a genetic algorithm for sorting network optimization," in Genetic and Evolutionary Computation Conference, New York, 2002, pp. 327–334.
11. R. Dawkins, The Blind Watchmaker. Penguin Books: London, 1991.
12. H. de Garis, 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 Computer Society Press, 1999, pp. 102–110.
13. T. Gordon and P. Bentley, "On evolvable hardware. In Soft Computing in Industrial Electronics, Ovaska, S. and Sztandera, L. (eds.), Physica-Verlag: Heidelberg 2001, pp. 279–323.

14. T. Gordon and P. Bentley, "Towards development in evolvable hardware," in Proc. of the 4th NASA/DoD Conference on Evolvable Hardware, A. Stoica, et al. (eds.), Alexandria, Virginia, USA, IEEE Computer Society: Los Alamitos, 2002, pp. 241–250.
15. F. Gruau, "Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm," PhD thesis, l'Universite Claude Bernard Lyon I, 1994, p. 159.
16. P. Haddow and G. Tufte, "Bridging the genotype–phenotype mapping for digital FPGAs," in Proc. of the 3rd NASA/DoD Workshop on Evolvable Hardware, Long Beach, CA, USA, 2001, IEEE Computer Society, Los Alamitos, 2001, pp. 109–115.
17. P. Haddow, G. Tufte and P. van Remortel, "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, 2001, pp. 128–139.
18. M. L. Harrison and J. A. Foster, Co-evolving faults to improve the fault-tolerance of sorting networks," in Proc. of the 7th European conference on Genetic Programming, LNCS 3003, Springer Verlag: Berlin, 2004, pp. 57–66.
19. T. Higuchi, et al., "Evolving hardware with genetic learning: A first step towards building a darwin machine," in Proc. of the 2nd International Conference on Simulated Adaptive Behaviour, MIT Press: Cambridge MA 1993, pp. 417–424.
20. W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure: Physica D," vol. 42, pp. 228–234, 1990.
21. G. S. Hornby and J. B. Pollack, "The advantages of generative grammatical encodings for physical design," in. Proc. of the 2001 Congress on Evolutionary Computation CEC2001, IEEE Computer Society Press: pp. 600–607, 2001.
22. L. Huelsbergen, "Finding general solutions to the parity problem by evolving machine-language representations," in Proc. of Conf. on Genetic Programming, 1998, pp. 158–166.
23. K. Imamura, J. A. Foster and A. W. Krings, "The test vector problem and limitations to evolving digital circuits," in: Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, IEEE Computer Society Press: pp. 75–79, 2000.
24. H. Juillé,"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, pp. 351–358.
25. H. Kitano, "Designing neural networks using genetic algorithms with graph generation system," Complex Systems, 4, pp. 461–476, 1990.
26. D. E. Knuth, The Art of Computer Programming: Sorting and Searching, 2nd edition, Addison Wesley, 1998.
27. J. R. Koza, et al., Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers: San Francisco, CA, 1999.
28. S. Kumar, "Investigating Computational Models of Development for the Construction of Shape and Form. PhD thesis," University of London, UK, 2004.
29. J. Masner, J. Cavalieri, J. Frenzel and J. Foster, "Size versus robustness in evolved sorting networks: Is Bigger Better?" in Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, IEEE Computer Press, 2000, pp. 81–90.
30. J. Miller, D. Job and V. Vassilev, "Principles in the evolutionary design of digital circuits—Part I. genetic programming and evolvable machines," vol. 1, no. 1, pp. 8–35, 2000.
31. J. Miller and P. Thomson, "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, pp. 93–104.
32. M. Murakawa, et al., "Evolvable hardware at function level. In: Proc. of the Parallel Problem Solving from Nature Conference. LNCS 1141, Springer Verlag, 1996, pp. 62–71.
33. C. L. Nehaniv, "Evolvability," Biosystems. vol. 69, no. 2-3, pp. 77–81, 2003.
34. L. Sekanina, "Evolvable Components: From Theory to Hardware Implementations. Natural Computing Series, Springer Verlag: Berlin, 2003.
35. L. Sekanina, "Evolving constructors for infinitely growing sorting networks and medians," in Proc. of the Conference on Current Trends in Theory and Practice of Computer Science SOFSEM 2004. LNCS 2932, Springer Verlag, 2004, pp. 314–323.
36. M. J. Streeter, M. A. Keane and J. R. Koza, "Routine duplication of post-2000 patented inventions by means of genetic programming," in Proc. of the 5th European Conference on Genetic Programming. Kinsale, Ireland,

2002, LNCS 2278, Springer: Berlin, 2002, pp. 26–36.
37. G. Tempesti, 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, pp. 141–152.
38. J. Torresen, “A scalable approach to evolvable hardware,” Genetic Programming and Evolvable Machines. vol. 3, no. 3, pp. 259–282, 2002.
39. G. Wagner and L. Altenberg, Complex adaptations and the evolution of evolvability. evolution, vol. 50, no. 3, pp. 967–976, 1996.
40. M. Wall, GAlib: A C++ Library of Genetic Algorithm Components, version 2.4. Massachusetts Institute of Technology, 1996, http://lancet.mit.edu/ga/dist/galibdoc.pdf