Efficient Genetic Programming for Finding Good Generalizing Boolean Functions

Stefan Droste
Lehrstuhl Informatik II
Universität Dortmund
44221 Dortmund, Germany
droste@ls2.informatik.uni-dortmund.de

ABSTRACT

This paper shows how genetic programming (GP) can help in finding generalizing Boolean functions when only a small part of the function values are given. The selection pressure favours functions having as few subfunctions as possible while only using essential variables, so the resulting functions should have good generalization properties. For efficiency no S-expressions are used for representation, but a special case of directed acyclic graphs known as ordered binary decision diagrams (OBDDs), making it possible to learn the 20-multiplexer.

1 Introduction

When using genetic programming (GP) or other techniques that use a number of training examples, the main goal is not only to create a program that exactly reproduces these training examples, but also has good generalizing properties: for inputs not in the training set the program should output values that closely resemble the underlying function. Intuitively one would call a smaller program better generalizing than a larger one, if both fit the training examples with the same quality, because the smaller program cannot differentiate between as many cases as the larger one can. Hence, it must use more regularities in the training examples. If one accepts that smaller programs generalize better than larger ones, one has to search for small programs that fit the training examples well in order to find good generalizing programs.

In GP S-expressions are often used to represent programs (see "Koza (1992)" for a detailed introduction to GP). To correctly measure the size of a program, all unnecessary code has to be removed. As it can be very time-consuming to remove all unnecessary code in S-expressions, in this paper ordered binary decision diagrams (OBDDs) (see "Bryant (1986)") are used to represent programs. Although this structure can only represent programs with Boolean input and output, it allows efficient removal of unnecessary code and can represent many functions with polynomial size in the number of input variables, while allowing many important operations to be done in polynomial time in the size of the OBDD (see "Wegener (1994)" for a survey article on OBDDs). OBDDs are a special case of directed acyclic graphs, so that a cache can be used to ensure that the same node is stored only once (see "Handley (1994)"). "Yanagiya (1995)" used OBDDs to learn the 20-multiplexer with GP for the first time ever.

In this paper the case is studied, that the correct function values of an unknown Boolean function are only known for a small part of its possible inputs, which is often the case in practical applications. Then one wants to find an OBDD, which has the correct values on this small part of the inputs and a minimal number of nodes. As only OBDDs, where different nodes represent different subfunctions, are considered, an OBDD with the minimal number of nodes uses the minimal number of different subfunctions, while fitting the given function values. Therefore, one can call the resulting function the most generalizing function that fits the given values (with respect to a given ordering of the input variables).

The next section gives a short introduction to well known facts about OBDDs, the third section presents genetic operators for OBDDs and the fourth section experimental results, where the given function values represent the multiplexer and parity function, resp. The last section gives a conclusion of the paper.

2 Ordered Binary Decision Diagrams

Given n Boolean input variables $x_1, \ldots, x_n$, let π be an ordering of these variables, i.e. a bijective function $\pi : \{x_1, \ldots, x_n\} \to \{1, \ldots, n\}$. An OBDD for π is an acyclic directed graph with one source, where every node is labeled either by one of the variables or one of the Boolean constants 0 or 1 (in the last two cases the node is called a *sink*). Every sink node has index $n+1$ and no outgoing edges, while every non-sink node has index i, if its label is x_i, and exactly two outgoing edges, one labeled by 0 and the other by 1 (whose endpoints are called *0-successor* and *1-successor*, resp.). Furthermore, if there is an edge from a node labeled x_i to a node labeled x_j, then $\pi(x_i)$ must be really smaller than $\pi(x_j)$ (in the following it is always assumed, that $\pi(x_i) = i$ for all $i \in \{1, \dots, n\}$).

To evaluate the function f_O represented by a given OBDD O for an input $(a_1, \dots, a_n) \in \{0,1\}^n$, one starts at the source. At a node labeled x_i one chooses the a_i-successor. The value $f_O(a)$ equals the label of the finally reached sink. So the OBDD O in Figure 1 represents the Boolean function $f_O(x_1, x_2, x_3) = x_1\overline{x}_3 \vee \overline{x}_1 x_2 \overline{x}_3$.

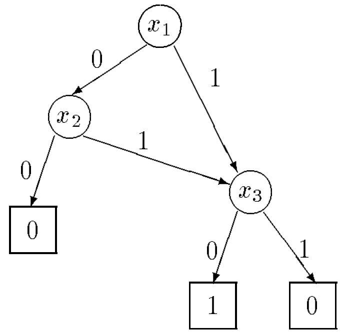

Figure 1: An example OBDD O

If a function $f : \{0,1\}^n \rightarrow \{0,1\}$ is given, an OBDD O_f that represents f can be constructed by building the complete decision tree (O_f stands for an appropriate OBDD when the function f is given, and f_O stands for the function represented by a given OBDD O).

The size $|O|$ of an OBDD O is the number of its non-sink nodes. An OBDD is called *reduced*, if it has no node with identical 0-successor and 1-successor and contains no isomorphic subgraphs. It is well known, that for a given function f the reduced OBDD is uniquely determined by starting with an arbitrary OBDD representing the function and using two reduction rules: the *deletion rule* states that a node v can be deleted, if its 0- and 1-successor are identical, where all incoming edges to v are redirected to this successor. The *merging rule* states that a node v can be deleted, if there is another node w with the same label and the same 0- and 1-successor, where all incoming edges to node v are redirected to node w. If neither the deletion nor the merging rule can be applied on an OBDD, this OBDD is reduced and for given f and π the reduced OBDD O_f is unique up to isomorphism (see "Wegener (1994)").

For a set $\{i_1, \dots, i_k\} \subseteq \{1, \dots, n\}$ of indices and $(a_1, \dots, a_k) \in \{0,1\}^k$ the function $f_{|x_{i_1}=a_1, \dots, x_{i_k}=a_k} : \{0,1\}^{n-k} \rightarrow \{0,1\}$ is defined as the restriction of f, where for every $j \in \{1, \dots, k\}$ the variable x_{i_j} is set to a_j. A function $f : \{0,1\}^n \rightarrow \{0,1\}$ *depends essentially* on x_i, if $f_{|x_i=0} \neq f_{|x_i=1}$. One can show that a reduced OBDD O_f contains exactly as many nodes labeled x_i as there are different subfunctions $f_{|x_1=a_1, \dots, x_{i-1}=a_{i-1}}$ depending essentially on x_i (for all $(a_1, \dots, a_{i-1}) \in \{0,1\}^{i-1}$).

If f is only partially defined, i.e. $f : \{0,1\}^n \rightarrow \{0,1,*\}$, O_f is defined as the set of all reduced OBDDs O, where $f_O(x) = f(x)$ for all $x \in \{0,1\}^n$ with $f(x) \neq *$. A set T of vectors $(x_1^i, \dots, x_n^i, y^i) \in \{0,1\}^{n+1}$ (called *table*) defines a partially defined function $f_T : \{0,1\}^n \rightarrow \{0,1,*\}$ by $f_T(x_1^i, \dots, x_n^i) = y_i$ for all $i \in \{1, \dots, |T|\}$, while $f_T(x) = *$ for all other $x \in \{0,1\}^n$.

Let O_{f_T} be the set of reduced OBDDs fitting the table T, which contains exactly $2^{2^n - |T|}$ OBDDs. The *minimal consistent OBDD* problem is the problem of finding the OBDD with the minimal number of nodes in the set O_{f_T} of reduced OBDDs for a given table T. This problem is known to be NP-hard (see "Sauerhoff and Wegener (1996)") and many heuristics for it are known (e.g. see "Shiple et al. (1994)"). In this paper it is shown how GP can (and in some sense has to) approximately solve the minimal consistent OBDD problem when T has small size, in order to find Boolean functions f_O with good generalization properties.

3 GP using OBDDs

The now described GP system uses only OBDDs that fit the given table T, therefore being similar to strongly typed GP. "Montana (1995)" showed that strongly typed GP can be very effective by restricting the space of used programs, i.e. the search space. This concept is also used here in slight variation:

The OBDDs of the initial generation are created in such a way that they fit the table, but otherwise are random. Every OBDD of a later generation, created by crossover or mutation, is explicitly checked, if it fits the table. If not, it is replaced by its parental OBDD, i.e. a reproduction takes place. This differs from most strongly typed GP systems, where the genetic operators implicitly guarantee the syntactic correctness of the new programs. This checking costs time $O(n \cdot |T|)$ per OBDD, but reduces the size of the search space by a factor of $2^{|T|}$. Subsection 4.3 compares this approach with GP without restriction by experimental results.

3.1 Representation of Programs

In this paper reduced OBDDs are used for representation of programs, because the fitness of a program is based on the size of its corresponding reduced OBDD and OBDDs allow very efficient representation of Boolean functions for GP (see "Yanagiya (1994)").

To automatically build reduced OBDDs without explicitly using the reduction rules, the following well known technique is used: in a hash table all nodes are stored by their label and their 0- and 1-successor. Every time a node shall be constructed, it is checked, if its 0- and 1-successor are identical. If so, this successor is returned without constructing a new node (deletion rule). Otherwise, the hash table is checked, if a node with the same label, 0-, and 1-successor already exists. If so, this node is returned without constructing a new node (merging rule); otherwise a new node is constructed, inserted in the hash table, and returned.

By constructing a node only after its 0- and 1-successor it is guaranteed, that the constructed OBDD is reduced. So only reduced OBDDs are used, lowering storage requirements for GP enormously. The hash table is used for all OBDDs, so equal nodes of different OBDDs are in memory only once. For all following estimations it is assumed that every access to the hash table can be done in time $O(1)$.

3.2 The initial OBDDs

The OBDDs of the initial generation are created in such a way, that they fit the table T. This is done by using a function `CheckTable`(x) taking as input a vector $x \in \{0,1,*\}^n$, that outputs the set of values y with $(x', y) \in T$, where x' fits x, i.e. $x'_i = x_i$ for all $i \in \{1, \ldots, n\}$ with $x_i \neq *$. `CheckTable` can be trivially implemented to work in time $O(n \cdot |T|)$. Notice, that any path from the source of an OBDD to one of its nodes v uniquely determines a vector $x \in \{0,1,*\}^n$ by starting with $x = (*, \ldots, *)$ and setting $x_i = a_i$, if the a_i-successor is taken at a node labeled x_i.

Using `CheckTable` an OBDD is build in a depth-first-manner using another function `BuildOBDD`(l, x), that creates a subOBDD, whose source has index $l \in \{1, \ldots, n+1\}$ and starts on the path $x \in \{0,1,*\}^n$. If l is $n+1$, the source has to be a sink: if `CheckTable`(x) is empty, then either the 0- or the 1-sink is returned with probability 1/2. Otherwise, `CheckTable`(x) contains exactly one element (see below): if it is 0, the 0-sink is returned, otherwise the 1-sink.

If l is at most n, the path x is copied into x^0 and x^1, where x^0_l is set to 0 and x^1_l to 1. If `CheckTable`(x) contains two elements, a new node with label x_l is created (and returned) in the way described in subsection 3.1, whose 0- and 1-successor are given by `BuildOBDD`$(l+1, x^0)$ and `BuildOBDD`$(l+1, x^1)$, resp. If `CheckTable`(x) contains at most one element, values $\Delta^0, \Delta^1 \in \{1, \ldots, n+1-l\}$ are chosen randomly and a new node with label x_l is created (and returned), whose 0- and 1-successor are given by `BuildOBDD`$(l+\Delta^0, x^0)$ and `BuildOBDD`$(l+\Delta^1, x^1)$, resp.

Calling `BuildOBDD`$(1, (*, \ldots, *))$ returns the source of an OBDD that fits the table, but contains random elements, too. As long as the actual node is relevant for two inputs $x, x' \in \{0,1\}^n$ with different outputs regarding to T, there is no randomness in the construction. But as soon as this is not the case anymore for a node labeled x_l, randomness is used. By choosing $P(\Delta = i) = (1-p)^{i-1} \cdot p$ for $i \in \{1, \ldots, n-l\}$ and $P(\Delta = n+1-l) = 1 - \sum_{i=1}^{n-l} P(\Delta = i)$ it is guaranteed that every node with a label greater than x_l has the same probability p of occurring in any path starting from this node labeled with x_l. In all following experiments p was 1/4.

3.3 Crossover

When using OBDDs one has to be careful about the crossover points to ensure that the resulting structure is still an OBDD, i.e. respects the variable ordering $x_1, \ldots, x_n$. To guarantee that the resulting structure is an OBDD, the following method is used here: in the first parental OBDD O_1 the crossover node v_1 is chosen by random, where every node and the two sinks have the same probability $1/(|O_1| + 2)$. In the second OBDD a crossover node v_2 is selected with equal probability from all nodes having an index, which is at least that of v_1 (similar to strongly typed GP, see "Montana (1995)"). Then the subOBDD starting at v_1 in the first OBDD O_1 is exchanged by the subOBDD starting at v_2. This exchange makes an update of the nodes on the chosen path in O_1 necessary. This is equivalent to tree modification in directed acyclic graphs (see "Ehrenburg (1996)").

The example in Figure 2 shows two parental OBDDs, where node v_1 is chosen in the left OBDD O_1 on the rightmost path $x = (1, *, *)$. Then node v_2 is chosen as the crossover point of the second OBDD O_2 from its two nodes with label x_3 and the two sinks.

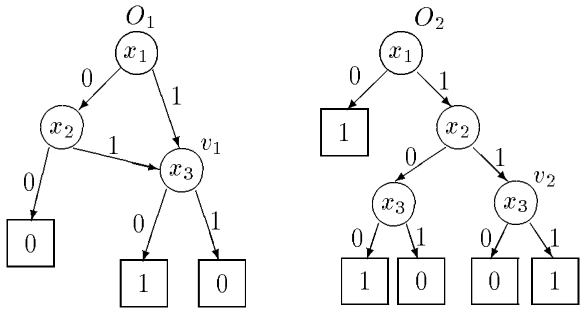

Figure 2: Two parental OBDDs O_1 and O_2

Now the first crossover fragment starting at v_1 is exchanged by the crossover fragment starting at v_2. In O_1 are two incoming edges to node v_1, but as only the nodes on the path x are updated, the 1-successor of the x_2-node remains node v_1. This leads to the left offspring OBDD O_1^* shown in Figure 3. But now the number of nodes in OBDD O_1^* is higher than in O_1, as there is a shared subOBDD in O_1, but not in O_1^*. So it can be advantageous, if all nodes of the OBDD are updated, i.e. all edges to the old crossover node are redirected to the new one. To do so, every edge on a path leading to the node v_1 is redirected to the corresponding node on the path, where v_1 is exchanged with v_2.

This is done by first doing a path update, where every old node on this path (included node v_1, if it is no sink) gets a reference pointer to the new node that substitutes it. Then a depth-first search in O_1^* is done, where every edge to an old node is replaced by an edge to the referenced new node (a node with a replaced successor becomes an old node, too, by its new successor). The right OBDD in Figure 3 is the result of the depth-first search, where every edge to an old node is replaced by an edge to the referenced node.

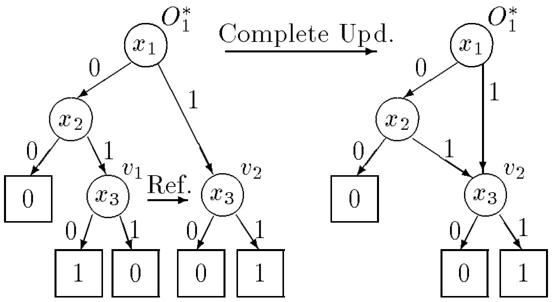

Figure 3: Path and complete update

After the exchange of the subOBDDs and path update with probability 1/2 a complete update of the OBDD is done, as there are also situations where only path update is advantageous. After the crossover the new OBDD O_1^* is tested, if it fits the table T; if not, it is replaced by its parental OBDD O_1, i.e. in this case crossover degenerates to reproduction. This crossover of two OBDDs O_1 and O_2 is done in time $O(|O_1| + |O_2| + n \cdot |T|)$.

3.4 Mutation

In this paper mutation is trying to apply the deletion rule to an OBDD O. So the mutation of an OBDD O works as follows: first, a node v (and a path x leading to v) is selected with probability $1/(|O|+2)$ from all of its nodes and its two sinks. If v is a sink, then v is replaced by a random subOBDD generated in the same way as described in subsection 3.2 (when using an empty table T). The index of the source of this subOBDD has to be greater than the indices of all predecessors of v in O (if there are no predecessors, then the index has to be at least 1).

If v is a non-sink node with index i, then let x^0 and x^1 be copies of x with $x_i^0 = 0$ and $x_i^1 = 1$. Now it is checked, if v can be deleted, because one of its successors is not relevant for fitting the table T, by checking the sets `CheckTable`(x^0) and `CheckTable`(x^1). If one of the sets is empty, the corresponding successor is not relevant, so the node v is replaced by the other successor (corresponding to the deletion rule applied to v). If both sets contain at least one element, v is replaced by a random subOBDD.

The replacement of v makes it necessary to update at least the nodes on the path x. Again, a complete update is done with probability 1/2. The explicit checking of the table by `CheckTable` prevents that most of the mutated OBDDs do not fit the table and leads to a high creation rate of random subOBDDs when the OBDDs are rather small. Neglecting the time needed to create a random OBDD, the whole mutation of an OBDD O is done in time $O(|O|+n \cdot |T|)$. In all experiments mutation probability was 0.1 and crossover probability 0.9.

3.5 Fitness

The standardized fitness of an OBDD O is simply the number $|O|$ of its non-sink nodes, which can be evaluated by a depth-first search in time $O(|O|)$. The adjusted fitness $a(i,t)$ of individual i in generation t is computed as follows: let $s^+(t)$ be the maximal standardized fitness in generation t and $s^-(t)$ be the minimal standardized fitness in generation t, then $a(i,t)$ is computed as

$$a(i,t) = \frac{1}{s(i,t) - (2 \cdot s^-(t) - s^+(t)) + \varepsilon(t)}.$$

Using this formula with $\varepsilon(t) = 0$, a best-of-generation individual i with $s(i,t) = s^-(t)$ has a 100% higher adjusted fitness than a worst-of-generation individual j with $s(j,t) = s^+(t)$. To allow escape from local minima, $\varepsilon(t)$ is chosen as $10/(\overline{s}(t) - s^-(t) + 0.001)$, where $\overline{s}(t)$ is the average standardized fitness in generation t.

4 Experimental results

The purpose of this section is twofold: first, it will be shown by experiment, that the described GP system can find small OBDDs when only few function values are given, thus finding generalizing functions. Second, it will be shown that the explicit restriction to OBDDs that fit the table T leads to much better results when using the described crossover and mutation operator.

The GP system was written from scratch and all experiments were done on a PC with a 33MHz Intel 80486DX CPU and 8MB main memory. The input of the GP system was the table $T = \{(x_1^i, \ldots, x_n^i, y^i) \in \{0,1\}^{n+1} \mid i \in \{1, \ldots, |T|\}\}$, so that a minimal OBDD in O_{f_T} should be found. In all following examples T was generated by a known function $f : \{0,1\}^n \to \{0,1\}$, i.e. $y^i = f(x_1^i, \ldots, x_n^i)$ for all $i \in \{1, \ldots, |T|\}$. This guarantees that the number of nodes of the minimal OBDD in O_{f_T} is at most $|O_f|$.

4.1 The multiplexer function

The first type of function used to generate the table T is the n-multiplexer function $mux_n : \{0,1\}^n \to \{0,1\}$, where $n = k + 2^k$ for some $k \geq 1$. It is defined by $mux_n(a_0, \ldots, a_{k-1}, d_0, \ldots, d_{2^k-1}) = d_i$, where i has the binary coding $a_0 \ldots a_{k-1}$. The size of the reduced OBDD for the $k+2^k$-multiplexer is $2^{k+1}-1$ for the used variable ordering $a_0, \ldots, a_{k-1}, d_0, \ldots, d_{2^k-1}$.

The table T used for a $k+2^k$-multiplexer has the size $4 \cdot 2^k$: if $a_0 \ldots a_{k-1}$ is the binary coding of $i \in \{0, \ldots, 2^k - 1\}$, then the following four elements are inserted in T:
$(a_0, \ldots, a_{k-1}, 0, \ldots, 0, 0), (a_0, \ldots, a_{k-1}, 1, \ldots, 1, 1)$,
$(a_0, \ldots, a_{k-1}, 0, \ldots, 0, \overbrace{1}^{k+1+i}, 0, \ldots, 0, 1)$, and
$(a_0, \ldots, a_{k-1}, 1, \ldots, 1, \overbrace{0}^{k+1+i}, 1, \ldots, 1, 0)$.

First, the GP system was tested for the 11-multiplexer. The number of individuals was 50, the number of generations 200, and the number of runs 100. Table 1 shows the size of the best-of-run OBDD, the time needed to find the best-of-run OBDD, and the generation, in which the best-of-run OBDD was found, over the 100 runs. In 61 of the 65 runs, where the size of the best-of-run OBDD was 15, the exact 11-multiplexer was found.

Table 1: 11-multiplexer

	OBDD Size	Time (sec)	Generation
Minimum	15	72	97
Average	15.9	117	160
Maximum	26	149	200

The same experiment was repeated with the 20-multiplexer: now T consisted of 64 elements, and the number of generations was raised to 2000. The number of individuals was still 50, the number of runs 10. The results over the 10 runs are shown in Table 2. In all 5 runs, where the size of the best-of-run OBDD was 31, the exact 20-multiplexer was found.

Table 2: 20-multiplexer

	OBDD Size	Time (sec)	Generation
Minimum	31	3843	1270
Average	32.1	4229	1558
Maximum	36	4993	1988

So the GP system was able to find the OBDDs of both multiplexers with relatively small time effort, although only 32 and 64 function values were given.

4.2 The parity function

The other function used to test the GP system was the n-parity function $par_n : \{0,1\}^n \rightarrow \{0,1\}$. The n-parity function par_n computes one, if and only if the number of ones in the input is even. While the table T for the multiplexer function consists of carefully chosen elements, in practical applications the given input values will not reflect the underlying function so well, but will be rather randomly chosen samples. To test the GP system for tables of such kind, the input values x of the table T for the parity function are chosen with equal probability from $\{0,1\}^n$. Then x is expanded by $par_n(x)$, so that $(x, par_n(x)) \in \{0,1\}^{n+1}$ is inserted into T.

Experiments with the n-parity function were done for $n = 11$ and 20 with tables of size 32 and 64, resp. As the size of the reduced OBDD for the n-parity function is $2n - 1$, the size of the optimal OBDD in O_{f_T} can be at most 21 and 39, resp. The first experiment was done with the 11-parity function: the number of individuals was 50, the number of generations 200, and the number of runs 100. The results over the 100 runs are shown in Table 3.

Table 3: 11-parity

	OBDD Size	Time (sec)	Generation
Minimum	9	25	41
Average	14.9	60	119
Maximum	21	102	199

For the 20-parity function the size of the table T was 64 (whose elements were chosen randomly as described above), the number of individuals was 50, the number of generations 1000, and the number of runs 20. The results over the 20 runs are shown in Table 4:

Table 4: 20-parity

	OBDD Size	Time (sec)	Generation
Minimum	20	554	316
Average	25.2	844	645
Maximum	31	1112	987

In both experiments the GP system found OBDDs being smaller than the parity-OBDD, showing that these OBDDs found regularities in the given tables (which were too small to reflect the parity function completely). Furthermore, the small size of the OBDDs shows that the data were not just simply "memorized".

4.3 Restricting the search space

The GP system described in this paper excludes all OBDDs which do not fit the table T by explicitly generating fitting OBDDs in the initial generation and replacing new OBDDs, which do not fit the table. This decreases the size of the search space by a factor of $2^{|T|}$, but also gives rise to the question, if this restriction can make it harder to find good solutions.

Therefore, the experiments for the 11-multiplexer described in subsection 4.1 were repeated in two versions to justify this restriction: in the first version all OBDDs were allowed, i.e. the random OBDDs of the initial generation were completely random OBDDs (constructed as described in subsection 3.2 for an empty table T), and no test was done after crossover or mutation. The standardized fitness $s(i,t)$ of individual i in generation t was computed as $s(i,t) = size(i,t) + 100 \cdot (|T| - hits(i,t))$ and its adjusted fitness $a(i,t)$ by the standard formula $a(i,t) = 1/(1 + s(i,t))$.

100 runs were done with 50 individuals over 200 generations. Graph 1 in Figure 4 shows the development of the standardized fitness of the best-of-generation OBDD on average over the 100 runs.

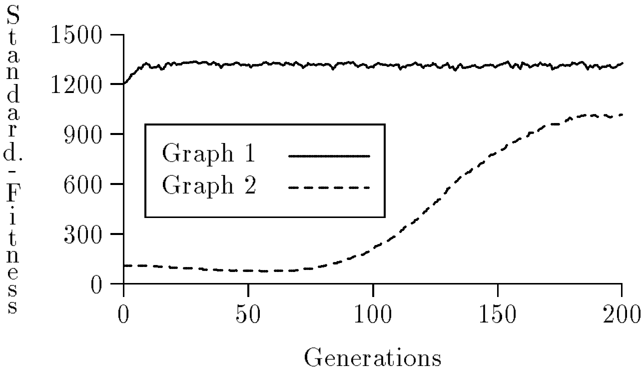

Figure 4: 11-multiplexer without restriction

One can see that the GP system made absolutely no progress in finding OBDDs which fit the table well. The best individual of all 100 runs had a standardized fitness of 605, i.e. for six elements of the table its value did not agree with the table T (its size was 5). Hence, the used mutation and crossover operators seem not to be suited for a GP system that operates with reduced OBDDs, when no restrictions are used.

Then this experiment was repeated with restricted initial programs, i.e. with explicit initial generation of OBDDs fitting the table as described in subsection 3.2, while new OBDDs from crossover and mutation were not tested. This was done with the same parameters, i.e. with 50 individuals and 200 generations over 100 runs. The results are shown in graph 2 in Figure 4.

In only 19 of these 100 runs, the 11-multiplexer OBDD with 15 nodes was found, showing that the initial generation of OBDDs that fit the table is very important for convergence to a good solution. But in many runs there was a point where the best-of-generation OBDD did not completely fit the table anymore and the following best-of-generation OBDDs became even worse (therefore being responsible for the rise shown in graph 2)

The two experiments show, that the initial generation of OBDDs which fit the table is very important for finding good OBDDs. Selection pressure is not sufficient to generate OBDDs which fit the table, when their predecessors do not have this property. The explicit testing of new OBDDs allows GP to be run over many generations while avoiding OBDDs which do not fit the table.

5 Conclusion

If one seeks for good generalizing functions, it can be useful to seek for functions having as few subfunctions as possible while fitting the fitness cases. In this paper it is shown, how GP can help in finding such generalizing Boolean functions. The usage of OBDDs, efficient data structures for representation, and efficient genetic operators for OBDDs are presented, lowering memory and time needs and making it possible to find the 20-multiplexer when only 64 fitness cases are given. Experimental results for the parity function show that small OBDDs can be found, even if the fitness cases are chosen arbitrarily. Furthermore, it is shown that it can be advantageous to restrict the search space of GP by explicitly forbidding some programs, in accordance with the results of strongly typed GP.

Acknowledgments

Hereby I thank Thomas Jansen and Ingo Wegener for their help while preparing the paper and the referees for their helpful comments. This research was supported by the DFG as part of the Collaborative Research Center “Computational Intelligence” (531).

Bibliography

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers.* Volume 35. Pages 677-691.

Ehrenburg, H. 1996. Improved directed acyclic graph evaluation and the combine operator in genetic programming. In *Proceedings of the Genetic Programming Conference GP-96.* Pages 285-290.

Handley, S. 1994. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of IEEE World Congress on Computational Intelligence.* Pages 154-159.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Montana, D.J. 1995. Strongly typed genetic programming. *Journal of Evolutionary Computation.* Vol. 3. Number 2. Pages 199-230.

Sauerhoff, M. and Wegener, I. 1996. On the complexity of minimizing the OBDD size for incompletely specified functions. In *IEEE Transactions on CAD.* Volume 15. Number 11.

Shiple, T.R., Hojati, R., Sangiovanni-Vincentelli, A.L., and Brayton, R.K. 1994. Heuristic minimization of BDDs using don't cares. In *Proceedings of the 31st Conference on Design Automation.* Pages 225-231.

Wegener, I. 1994. Efficient data structures for Boolean functions. *Discrete Mathematics.* Volume 136. Pages 347-372.

Yanagiya, M. 1995. Efficient genetic programming based on binary decision diagrams. In *Proceedings of International IEEE Conference on Evolutionary Computation.* Pages 234-239.