Designing Digital Circuits for the Knapsack Problem

Mihai Oltean[1], Crina Groşan[1], and Mihaela Oltean[2]

[1] Department of Computer Science,
Faculty of Mathematics and Computer Science,
Babeş-Bolyai University, Kogălniceanu 1
Cluj-Napoca, 3400, Romania.
{moltean, cgrosan}@cs.ubbcluj.ro
[2] David Prodan College, Cugir, 2566, Romania.
olteanmihaelaelena@yahoo.com

Abstract. Multi Expression Programming (MEP) is a Genetic Programming variant that uses linear chromosomes for solution encoding. A unique feature of MEP is its ability of encoding multiple solutions of a problem in a single chromosome. In this paper we use Multi Expression Programming for evolving digital circuits for a well-known NP-Complete problem: the knapsack (subset sum) problem. Numerical experiments show that Multi Expression Programming performs well on the considered test problems.

1 Introduction

The problem of evolving digital circuits has been deeply analyzed in the recent past [4]. A considerable effort has been spent on evolving very efficient (regarding the number of gates) digital circuits. J. Miller, one of the pioneers in the field of the evolvable digital circuits, used a special technique called Cartesian Genetic Programming (CGP) [4] for evolving digital circuits. CGP architecture consists of a network of gates (placed in a grid structure) and a set of wires connecting them. The results [4] show that CGP is able to evolve digital circuits competitive to those designed by human experts.

In this paper, we use Multi Expression Programming (MEP)[3] [5] for evolving digital circuits. MEP is a Genetic Programming (GP) [3] variant that uses linear chromosomes of fixed length. A unique feature of MEP is its ability of storing multiple solutions of a problem in a single chromosome. Note that this feature does not increase the complexity of the MEP decoding process when compared to other techniques that store a single solution in a chromosome.

In this paper we present the way in which MEP may be efficiently applied for evolving digital circuits. We describe the way in which multiple digital circuits may be stored in a single MEP chromosome and the way in which the fitness

[3] MEP source code is available from www.mep.cs.ubbcluj.ro.

of this chromosome may be computed by traversing the MEP chromosome only once.

In this paper MEP is used for evolving digital circuits for a well-known NP-Complete [2] problem: the knapsack (subset sum) problem. Since this problem is NP-Complete we cannot realistically expect to find a polynomial-time algorithm for it. Instead, we have to speed-up the existing techniques in order to reduce the time needed to obtain a solution. A possibility for speeding-up the algorithms for this problem is to implement them in assembly language. This could lead sometimes to improvements of over two orders of magnitude. Another possibility is to design and build a special hardware dedicated to that problem. This approach could lead to significant improvements of the running time. Due to this reason we have chosen to design, by the means of evolution, digital circuits for several instances of the knapsack problem.

The knapsack problem may also be used as benchmarking problem for the evolutionary techniques which design electronic circuits. The main advantage of the knapsack problem is its scalability: increasing the number of inputs leads to more and more complicated circuits. The results show that MEP performs very well for all the considered test problems.

The paper is organized as follows. In section 2, the problem of designing digital circuits and the knapsack problem are shortly described. The Multi Expression Programming technique is presented in section 3. Several numerical experiments are performed in section 4.

2 Problem Statement and Representation

The problem that we are trying to solve in this paper may be briefly stated as follows:

Find a digital circuit that implements a function given by its truth table.

The gates that are usually used in the design of digital circuits along with their description are given in Table 1.

Table 1. Function set (gates) used in numerical experiments. These functions are taken from [4]

#	Function	#	Function
0	$a \cdot b$	5	$a \oplus \bar{b}$
1	$a \cdot \bar{b}$	6	$a + b$
2	$\bar{a} \cdot b$	7	$a + \bar{b}$
3	$\bar{a} \cdot \bar{b}$	8	$\bar{a} + b$
4	$a \oplus b$	9	$\bar{a} + \bar{b}$

The knapsack problem (or the subset sum problem) may be stated as follows:

Let M be a set of numbers and a target sum k. Is there a subset $S \subseteq M$ which has the sum k?

The knapsack problem is a well-known NP-Complete problem [2]. No polynomial-time algorithm is known for this problem.

Instead of designing a heuristic for this problem we will try to evolve digital circuits which will provide the answer for a given input.

In the experiments performed in this paper the set M consists of several integer numbers from the set of consecutive integers starting with 1. For instance if the base set is {1, 2, 3, 4, 5, 6, 7} then M may be {2, 5, 6}. We will try to evolve a digital circuit that is able to provide the correct answer for all subsets M of the base set.

The input for this problem is a sequence of bits. A value of 1 in position k means that the integer number k belongs to the set M, otherwise the number k does not belong to the set M.

For instance consider the consecutive integer numbers starting with 1 and ending with 7. The string 0100110 encodes the set $M = \{2, 5, 6\}$. The numbers 1, 3, 4 and 7 do not belong to M since the corresponding positions are 0. The possible subsets of M instance have the sum 2, 5, 6, 7, 8, 11 or 13. In our approach, the target sum is fixed and we are asking if is there a subset of given sum.

The number of training instances for this problem depends on the number of consecutive integers used as base for M. If we use numbers 1, 2 and 3, we have $2^3 = 8$ training instances. If we use numbers 1, 2, 3, 4, 5, 6 and 7, we have $2^7 = 128$ training instances. In this case, whichever subset M of {1,...,7} will be presented to the evolved circuit we have to obtain a binary answer whether the target sum k may or not be obtained from a subset of M.

3 Multi Expression Programming

In this section, *Multi Expression Programming* (MEP) [5] is briefly described.

3.1 MEP Representation

MEP genes are represented by substrings of a variable length. The number of genes per chromosome is constant. This number defines the length of the chromosome. Each gene encodes a terminal or a function symbol. A gene encoding a function includes pointers towards the function arguments. Function arguments always have indices of lower values than the position of that function in the chromosome.

This representation is similar to the way in which ***C*** and ***Pascal*** compilers translate mathematical expressions into machine code [1].

The proposed representation ensures that no cycle arises while the chromosome is decoded (phenotypically transcripted). According to the proposed representation scheme the first symbol of the chromosome must be a terminal symbol. In this way only syntactically correct programs (MEP individuals) are

obtained.

Example

A representation where the numbers on the left positions stand for gene labels is employed here. Labels do not belong to the chromosome, they are being provided only for explanation purposes.

For this example we use the set of functions $F = \{+, *\}$, and the set of terminals $T = \{a, b, c, d\}$. An example of chromosome using the sets F and T is given below:

1: a
2: b
3: + 1, 2
4: c
5: d
6: + 4, 5
7: * 3, 6

3.2 Decoding MEP Chromosomes and Fitness Assignment Process

In this section it is described the way in which MEP individuals are translated into computer programs and the way in which the fitness of these programs is computed.

This translation is achieved by reading the chromosome top-down. A terminal symbol specifies a simple expression. A function symbol specifies a complex expression obtained by connecting the operands specified by the argument positions with the current function symbol.

For instance, genes 1, 2, 4 and 5 in the previous example encode simple expressions formed by a single terminal symbol. These expressions are:

$E_1 = a$,
$E_2 = b$,
$E_4 = c$,
$E_5 = d$,

Gene 3 indicates the operation + on the operands located at positions 1 and 2 of the chromosome. Therefore gene 3 encodes the expression:

$E_3 = a + b$.

Gene 6 indicates the operation + on the operands located at positions 4 and 5. Therefore gene 6 encodes the expression:

$E_6 = c + d$.

Gene 7 indicates the operation * on the operands located at position 3 and 6. Therefore gene 7 encodes the expression:

$E_7 = (a + b) * (c + d)$.

E_7 is the expression encoded by the whole chromosome.

There is neither practical nor theoretical evidence that one of these expressions is better than the others. Moreover, Wolpert and McReady [7] proved that we cannot use the search algorithm's behavior so far for a particular test function to predict its future behavior on that function. This is why each MEP chromosome is allowed to encode a number of expressions equal to the chromosome length. Each of these expressions is considered as being a potential solution of the problem.

The value of these expressions may be computed by reading the chromosome top down. Partial results are computed by dynamic programming and are stored in a conventional manner.

As MEP chromosome encodes more than one problem solution, it is interesting to see how the fitness is assigned.

Usually the chromosome fitness is defined as the fitness of the best expression encoded by that chromosome.

For instance, if we want to solve symbolic regression problems the fitness of each sub-expression E_i may be computed using the formula:

$$f(E_i) = \sum_{k=1}^{n} |o_{k,i} - w_k|, \tag{1}$$

where $o_{k,i}$ is the obtained result by the expression E_i for the fitness case k and w_k is the targeted result for the fitness case k. In this case the fitness needs to be minimized.

The fitness of an individual is set to be equal to the lowest fitness of the expressions encoded in chromosome:

$$f(C) = \min_i f(E_i). \tag{2}$$

When we have to deal with other problems we compute the fitness of each sub-expression encoded in the MEP chromosome and the fitness of the entire individual is given by the fitness of the best expression encoded in that chromosome.

3.3 Search Operators

Search operators used within MEP algorithm are crossover and mutation. Considered search operators preserve the chromosome structure. All offspring are syntactically correct expressions.

Crossover

By crossover two parents are selected and are recombined. For instance, within the uniform recombination the offspring genes are taken randomly from one parent or another.

Example

Let us consider the two parents C_1 and C_2 given in Table 2. The two offspring O_1 and O_2 are obtained by uniform recombination as shown in Table 2.

Table 2. MEP uniform recombination

Parents		*Offspring*	
C_1	C_2	O_1	O_2
1: ***b***	1: a	1: a	1: ***b***
2: *** 1, 1**	2: b	2: *** 1, 1**	2: b
3: **+ 2, 1**	3: + 1, 2	3: **+ 2, 1**	3: + 1, 2
4: ***a***	4: c	4: c	4: ***a***
5: *** 3, 2**	5: d	5: *** 3, 2**	5: d
6: ***a***	6: + 4, 5	6: + 4, 5	6: ***a***
7: **- 1, 4**	7: * 3, 6	7: **- 1, 4**	7: * 3, 6

Mutation

Each symbol (terminal, function of function pointer) in the chromosome may be target of mutation operator. By mutation some symbols in the chromosome are changed. To preserve the consistency of the chromosome its first gene must encode a terminal symbol.

Example

Consider the chromosome C given in Table 3. If the boldfaced symbols are selected for mutation an offspring O is obtained as shown in Table 3.

3.4 MEP Algorithm

In this paper we use a steady-state [6] as underlying mechanism for Multi Expression Programming. The algorithm starts by creating a random population of individuals. The following steps are repeated until a stop condition is reached. Two parents are selected using a selection procedure. The parents are recombined in order to obtain two offspring. The offspring are considered for mutation. The best offspring replaces the worst individual in the current population if the offspring is better than the worst individual.

The algorithm returns as its answer the best expression evolved along a fixed number of generations.

Table 3. MEP mutation

C	O
1: a	1: a
2: * 1, 1	2: * 1, 1
3: ***b***	3: **+ 1, 2**
4: * 2, 2	4: * 2, 2
5: b	5: b
6: + **3**, 5	6: + **1**, 5
7: a	7: a

4 Numerical Experiments

In this section several numerical experiments for evolving digital circuits for the knapsack problem are performed. The general parameters of the MEP algorithm are given in Table 4. Since different instances of the problem being solved will have different degrees of difficulty we will use different population sizes, number of genes in a chromosome and number of generations for each instance. Particular parameters are given in Table 5.

Table 4. General parameters of the MEP algorithm for evolving digital circuits

Parameter	**Value**
Crossover probability	0.9
Crossover type	Uniform
Mutations	5 / chromosome
Function set	Gates 0 to 9 (see Table 1)
Terminal set	Problem inputs
Selection	Binary Tournament

Experimental results are given in Table 6. We are interested in computing the number of successful runs and the number of gates in the shortest evolved circuit.

Table 6 shows that MEP successfully found at least a solution for the considered test problems. The difficulty of evolving a digital circuit for this problem increases with the number of inputs of the problem. Only 20 individuals are required to obtain 39 solutions (out of 100 runs) for the instance with 4 inputs. In return, 1000 individuals (50 times more) are required to obtain 10 perfect solutions (out of 100 independent runs) for the instance with 7 inputs. Also the size of the evolved circuits increases with the number of problem inputs. However, due to the reduced number of runs we cannot be sure that we have obtained the optimal circuits. Additional experiments are required in this respect.

Table 5. Particular parameters of the MEP algorithm for different instances of the knapsack problem. In the second column the base set of numbers is given for each instance. In the third column the target sum is given.

#	Set of numbers	Sum	Number of fitness cases	Population size	Number of genes	Number of generations
1	$\{1 \ldots 4\}$	5	16	20	10	51
2	$\{1 \ldots 5\}$	7	32	100	30	101
3	$\{1 \ldots 6\}$	10	64	500	50	101
4	$\{1 \ldots 7\}$	14	128	1000	100	201

Table 6. Results obtained by MEP for the considered test problems. 100 independent runs have been performed for all problems

#	Set of numbers	Sum	Successful runs	Number of gates in the shortest circuit
1	$\{1 \ldots 4\}$	5	39 out of 100	3
2	$\{1 \ldots 5\}$	7	31 out of 100	5
3	$\{1 \ldots 6\}$	10	10 out of 100	11
4	$\{1 \ldots 7\}$	14	7 out of 100	21

Due to the NP-Completeness of the problem it is expected that the number of gates in the shortest circuit to increase exponentially with the number of inputs.

References

1. Aho A., Sethi R., and Ullman J.: Compilers: Principles, Techniques, and Tools, Addison Wesley, (1986)
2. Garey M.R., Johnson D.S.: Computers and Intractability: A Guide to NP-completeness, Freeman & Co, San Francisco, (1979)
3. Koza J. R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, (1992)
4. Miller J. F., Job D. and Vassilev V.K.: Principles in the Evolutionary Design of Digital Circuits - Part I, Genetic Programming and Evolvable Machines, Vol. 1(1), Kluwer Academic Publishers, (1999) 7-35
5. Oltean M.: Solving Even-Parity Problems using Multi Expression Programming, in Proceedings of the the 7^{th} Joint Conference on Information Sciences, Research Triangle Park, North Carolina, Edited by Ken Chen (et. al), (2003) 315-318
6. Syswerda G.: Uniform Crossover in Genetic Algorithms, In Proceedings of the 3^{rd} International Conference on Genetic Algorithms, J.D. Schaffer (eds), Morgan Kaufmann Publishers, CA, (1989) 2-9
7. Wolpert D.H. and McReady W.G.: No Free Lunch Theorems for Search, Technical Report, SFI-TR-05-010, Santa Fe Institute, (1995)