Evolving Evolutionary Algorithms Using Linear Genetic Programming

Mihai Oltean moltean@cs.ubbcluj.ro
Department of Computer Science, Babeş-Bolyai University, Kogalniceanu 1, Cluj-Napoca 3400, Romania

Abstract

A new model for evolving Evolutionary Algorithms is proposed in this paper. The model is based on the Linear Genetic Programming (LGP) technique. Every LGP chromosome encodes an EA which is used for solving a particular problem. Several Evolutionary Algorithms for function optimization, the Traveling Salesman Problem and the Quadratic Assignment Problem are evolved by using the considered model. Numerical experiments show that the evolved Evolutionary Algorithms perform similarly and sometimes even better than standard approaches for several well-known benchmarking problems.

Keywords

Genetic algorithms, genetic programming, linear genetic programming, evolving evolutionary algorithms

1 Introduction

Evolutionary Algorithms (EAs) (Goldberg, 1989; Holland, 1975) are new and powerful tools used for solving difficult real-world problems. They have been developed in order to solve some real-world problems that the classical (mathematical) methods failed to successfully tackle. Many of these unsolved problems are (or could be turned into) optimization problems. The solving of an optimization problem means finding solutions that maximize or minimize a criteria function (Goldberg, 1989; Holland, 1975; Yao et al., 1999).

Many Evolutionary Algorithms have been proposed for dealing with optimization problems. Many solution representations and search operators have been proposed and tested within a wide range of evolutionary models. There are several natural questions to be answered in all these evolutionary models:

What is the optimal population size?
What is the optimal individual representation?
What are the optimal probabilities for applying specific genetic operators?
What is the optimal number of generations before halting the evolution?

A breakthrough arose in 1995 when Wolpert and McReady unveiled their work on *No Free Lunch* (NFL) theorems for *Search* (Wolpert et al., 1995) and *Optimization* (Wolpert et al., 1997). The No Free Lunch theorems state that all the black-box algorithms have the same average performance over the entire set of optimization problems. (A black-box algorithm does not take into account any information about the problem or the particular instance being solved.) The magnitude of the NFL results stroke all the efforts for developing a universal black-box optimization algorithm capable of solving all the optimization problems in the best manner. Since we cannot build an EA able to solve best all problems we have to find other ways to construct algorithms that perform very well for some particular problems. One possibility (explored in this paper) is to let the evolution to discover the optimal structure and parameters for the evolutionary algorithm used for solving a particular problem.

In their attempt for solving problems, men delegated computers to develop algorithms capable of performing certain tasks. The most prominent effort in this direction is Genetic Programming (GP) (Koza, 1992; Koza, 1994), an evolutionary technique used for breeding a population of computer programs. Instead of evolving solutions for a particular problem instance, GP is mainly intended for discovering computer programs capable of solving particular classes of optimization problems. (This statement is only partially true since the discovery of computer programs may also be viewed as a technique for solving a particular problem input. For instance, the problem may be here: "Find a computer program that calculates the sum of the elements of an array of integers.").

There are many such approaches in literature concerning GP. Noticeable effort has been dedicated for evolving deterministic computer programs capable of solving specific problems such as symbolic regression (Koza, 1992; Koza, 1994), classification (Brameier et al., 2001a) etc.

Instead of evolving such deterministic computer programs we will evolve a full-featured evolutionary algorithm (i.e. the output of our main program will be an EA capable of performing a given task). Thus, we will work with EAs at two levels: the first (macro) level consists in a steady-state EA (Syswerda, 1989) which uses a fixed population size, a fixed mutation probability, a fixed crossover probability etc. The second (micro) level consists in the solutions encoded in a chromosome of the first level EA.

For the first (macro) level EA we use an evolutionary model similar to Linear Genetic Programming (LGP) (Brameier et al., 2001a; Brameier et al., 2001b; Brameier et al., 2002) which is very suitable for evolving computer programs that may be easily translated into an imperative language (like ***C*** or ***Pascal***).

The rules employed by the evolved EAs during of a generation are not preprogrammed. These rules are automatically discovered by the evolution. The evolved EA is a generational one (the generations do not overlap).

This research was motivated by the need of answering several important questions concerning Evolutionary Algorithms. The most important question is "Can Evolutionary Algorithms be automatically synthesized by using only the information about the problem being solved?" (Ross, 2002). And, if yes, which are the genetic operators that have to be used in conjunction with an EA (for a given problem)? Moreover, we are also interested to find the optimal (or near-optimal) sequence of genetic operations (selections, crossovers and mutations) to be performed during a generation of an Evolutionary Algorithm for a particular problem. For instance, in a standard GA the sequence is the following: selection, recombination and mutation. But, how do we know that scheme is the best for a particular problem (or problem instance)? We better let the evolution to find the answer for us.

Several attempts for evolving Evolutionary Algorithms were made in the past (Ross, 2002; Tavares et al., 2004). A non-generational EA was evolved (Oltean et al., 2003) by using the Multi Expression Programming (MEP) technique (Oltean et al., 2003; Oltean, 2003).

There are also several approaches that evolve genetic operators for solving difficult problems (Angeline, 1995; Angeline, 1996; Edmonds, 2001; Stephens et al., 1998; Teller, 1996). In his paper on Meta-Genetic Programming, Edmonds (Edmonds, 2001) used two populations: a standard GP population and a co-evolved population of operators that act on the main population. Note that all these approaches use a fixed evolutionary algorithm which is not changed during the search.

A recent paper of Spector and Robinson (Spector, 2002) describes a language called Push which supports a new, self-adaptive form of evolutionary computation called *autoconstructive evolution*. An experiment for symbolic regression problems was reported. The conclusion was that "Under most conditions the population quickly achieves reproductive competence and soon thereafter improves in fitness." (Spector 2002).

There are also several attempts for evolving heuristics for particular problems. In (Oltean et al., 2004a) the authors evolve an heuristic for the Traveling Salesman Problem. The obtained heuristic is a mathematical expression that takes as input some information about the already constructed path and outputs the next node of the path. It was shown (Oltean et al., 2004a) that the evolved heuristic performs better than other well-known heuristics (Nearest Neighbor Heuristic, Minimum Spanning Tree Heuristic (Cormen et al., 1990; Garey et al., 1979)) for the considered test problems.

The paper is organized as follows. The LGP technique is described in section 2. The model used for evolving EAs is presented in section 3. Several numerical experiments are performed in section 4. Three EAs for function optimization, the Traveling Salesman Problem and the Quadratic Assignment Problem are evolved in sections 4.1, 4.2 and 4.3. Further research directions are suggested in section 5.

2 Linear Genetic Programming Technique

In this section the *Linear Genetic Programming* (LGP) technique is described. LGP uses a linear chromosome representation and a special phenotype transcription model.

2.1 LGP Algorithm

In our experiments steady-state (Syswerda, 1989) is used as underlying mechanism for LGP.

The steady-state LGP algorithm starts with a randomly chosen population of individuals. The following steps are repeated until a termination condition is reached: Two parents are selected by using binary tournament and are recombined with a fixed crossover probability. Two offspring are obtained by the recombination of two parents. The offspring are mutated and the best of them replaces the worst individual in the current population (if the offspring is better than the worst individual in the current population).

2.2 Individual Representation

Linear Genetic Programming (LGP) (Banzhaf et al., 1998; Brameier et al., 2001a; Nordin, 1994) uses a specific linear representation of computer programs. Programs of an imperative language (like ***C***) are evolved instead of the tree-based GP expressions of a functional programming language (like ***LISP***).

An LGP individual is represented by a variable-length sequence of simple ***C*** language instructions. Instructions operate on one or two indexed variables (registers) r or on constants c from predefined sets. The result is assigned to a destination register, e.g. $r_i = r_j * c$.

An example of an LGP program is the following:

```
void LGP_Program(double v[8])
{
...
v[0] = v[5] + 73;
v[7] = v[4] - 59;
v[4] = v[2] *v[1];
v[2] = v[5] + v[4];
v[6] = v[1] * 25;
v[6] = v[4] - 4;
v[1] = sin(v[6]);
v[3] = v[5] * v[5];
v[7] = v[6] * 2;
v[5] = [7] + 115;
v[1] = sin(v[7]);
}
```

A linear genetic program can be turned into a functional representation by successive replacements of variables starting with the last effective instruction (Brameier et al., 2001a).

Variation operators are crossover and mutation. By crossover continuous sequences of instructions are selected and exchanged between parents (Brameier et al., 2001a). Two cutting points are randomly chosen in each parent and the sequences of instructions between them are exchanged. As an immediate effect, the length of the obtained offspring might be different from the parents.

Two types of mutations are used: micro mutation and macro mutation (Brameier et al., 2001a). By micro mutation an operand or an operator of an instruction is changed. Macro mutation inserts or deletes a random instruction.

3 LGP for Evolving Evolutionary Algorithms

In order to use LGP for evolving EAs we have to modify the structure of an LGP chromosome and define a set of function symbols.

3.1 Individual Representation for Evolving EAs

Instead of working with registers, our LGP program will modify an array of individuals (the population). We denote by *Pop* the array of individuals (the population) which will be modified by an LGP program.

The set of function symbols will consist in genetic operators that may appear into an evolutionary algorithm. There are usually 3 types of genetic operators that may appear into an EA. These genetic operators are:

Select - selects the best solution among several already existing solutions,

Crossover - recombines two existing solutions,

Mutate - varies an existing solution.

These operators will act as function symbols that may appear into an LGP chromosome. Thus, each simple ***C*** instruction that appeared into a standard LGP chromosome will be replaced by a more complex instruction containing genetic operators. More specifically, we have three major types of instructions in the modified LGP chromosomes. These instructions are:

```
Pop[k] = Select (Pop[i], Pop[j]); // Select the best individual from those stored in
                                  // Pop[i] and Pop[j] and keep the result in position k.

Pop[k] = Crossover (Pop[i], Pop[j]); // Crossover the individuals stored in
                                  // Pop[i] and Pop[j] and keep the result in position k.

Pop[k] = Mutate (Pop[i]); // Mutate the individual stored in
                                  // position i and keep the result in position k.
```

Remarks:

(i) The *Crossover* operator always generates a single offspring from two parents in our model. Crossover operators generating two offspring may be designed to fit our evolutionary model as well.

(ii) The *Select* operator acts as a binary tournament selection. The better of two individuals is always accepted as the result of the selection.

(iii) *Crossover* and *Mutate* operators are problem dependent. For instance, if we want to evolve an EA (with binary representation) for function optimization we may use the set of genetic operators having the following functionality: *Crossover* – recombines two parents using one cut point crossover, *Mutate* – one point mutation. If we want to evolve an EA for solving the TSP problem (Merz et al., 1997) we may use DPX as a crossover operator and 2-opt as a mutation operator (Krasnogor, 2002).

An LGP chromosome C, storing an evolutionary algorithm is the following:

```
void LGP_Program(Chromosome  Pop[8]) // a population with 8 individuals
{
...
   Pop[0] = Mutate(Pop[5]);
   Pop[7] = Select(Pop[3], Pop[6]);
   Pop[4] = Mutate(Pop[2]);
   Pop[2] = Crossover(Pop[0], Pop[2]);
   Pop[6] = Mutate(Pop[1]);
   Pop[2] = Select(Pop[4], Pop[3]);
   Pop[1] = Mutate(Pop[6]);
   Pop[3] = Crossover(Pop[5], Pop[1]);
...
}
```

These statements will be considered to be genetic operations executed during an EA generation. Since our purpose is to evolve a generational EA we have to add a wrapper loop around the genetic operations that are executed during an EA generation. More than that, each EA starts with a random population of individuals. Thus, the LGP program must contain some instructions that initialize the initial population.

The obtained LGP chromosome is given below:

```
void LGP_Program(Chromosome  Pop[8]) // a population with of 8 individuals
{
    Randomly_initialize_the_population();
    for (int k = 0; k < MaxGenerations; k++){ // repeat for a number of generations
        Pop[0] = Mutate(Pop[5]);
        Pop[7] = Select(Pop[3], Pop[6]);
        Pop[4] = Mutate(Pop[2]);
        Pop[2] = Crossover(Pop[0], Pop[2]);
        Pop[6] = Mutate(Pop[1]);
        Pop[2] = Select(Pop[4], Pop[3]);
        Pop[1] = Mutate(Pop[6]);
        Pop[3] = Crossover(Pop[5], Pop[1]);
        }
}
```

Remark: The initialization function and the **for** cycle will not be affected by the genetic operators. These parts are kept unchanged during the search process.

3.2 Fitness Assignment

We deal with EAs at two different levels: a micro level representing the evolutionary algorithm encoded into an LGP chromosome and a macro level GA, which evolves LGP individuals. Macro level GA execution is bounded by known rules for GAs (see (Goldberg, 1989)).

In order to compute the fitness of a LGP individual we have to compute the quality of the EA encoded in that chromosome. For this purpose the EA encoded into a LGP chromosome is run on the particular problem being solved.

Roughly speaking the fitness of an LGP individual equals the fitness of the best solution generated by the evolutionary algorithm encoded into that LGP chromosome. But since the EA encoded into a LGP chromosome uses pseudo-random numbers it is very likely that successive runs of the same EA will generate completely different solutions. This stability problem is handled in a standard manner: the EA encoded into an LGP chromosome is executed (run) more times (500 runs are in fact executed in all the experiments performed for evolving EAs for function optimization and 25 runs for evolving EAs for TSP and QAP) and the fitness of a LGP chromosome is the average of the fitness of the EA encoded in that chromosome over all the runs.

The optimization type (minimization/maximization) of the macro level EA is the same as the optimization type of the micro level EA. In our experiments we have employed a minimization relation (finding the minimum of a function and finding the shortest TSP path and finding the minimal quadratic assignment).

Remark. In standard LGP one of the registers is chosen as the program output. This register is not changed during the search process. In our approach the register storing the best value (best fitness) in all the generations is chosen to represent the chromosome. Thus, every LGP chromosome stores multiple solutions of a problem in the same manner as Multi Expression Programming does (Oltean et al., 2003; Oltean, 2003; Oltean et al., 2004b).

3.3 The Model Used for Evolving EAs

For evolving EAs we use the steady state algorithm described in section 2.1. The problem set is divided into two sets, suggestively called training set, and test set. In our experiments the training set consists in a difficult test problem. The test set consists in some other well-known benchmarking problems (Burkard et al., 1991; Reinelt, 1991; Yao et al., 1999).

4 Numerical Experiments

In this section several numerical experiments for evolving EAs are performed. Two evolutionary algorithms for function optimization, the TSP and the QAP problems are evolved. For assessing the performance of the evolved EAs, several numerical experiments with a standard Genetic Algorithm for function optimization, for TSP and for QAP are also performed and the results are compared.

4.1 Evolving EAs for Function Optimization

In this section an Evolutionary Algorithm for function optimization is evolved.

4.1.1 Test Functions

Ten test problems $f_1 - f_{10}$ (given in Table 1) are used in order to asses the performance of the evolved EA. Functions $f_1 - f_6$ are unimodal test function. Functions $f_7 - f_{10}$ are highly multimodal (the number of the local minima increases exponentially with the problem dimension (Yao et al., 1999)).

Table 1: Test functions used in our experimental study. The parameter n is the space dimension (n = 5 in our numerical experiments) and f_{min} is the minimum value of the function.

Test function	Domain	f_{min}
$f_1(x) = \sum_{i=1}^{n} (i \cdot x_i^2)$.	[-10, 10]^n	0
$f_2(x) = \sum_{i=1}^{n} x_i^2$.	[-100, 100]^n	0
$f_3(x) = \sum_{i=1}^{n} \lvert x_i \rvert + \prod_{i=1}^{n} \lvert x_i \rvert$.	[-10, 10]^n	0
$f_4(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j^2 \right)$.	[-100, 100]^n	0
$f_5(x) = \max\{x_i, 1 \leq i \leq n\}$.	[-100, 100]^n	0
$f_6(x) = \sum_{i=1}^{n-1} 100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2$.	[-30, 30]^n	0
$f_7(x) = 10 \cdot n + \sum_{i=1}^{n} (x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i))$	[-5, 5]^n	0
$f_8(x) = -a \cdot e^{-b\sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}} - e^{\frac{\sum \cos(c \cdot x_i)}{n}} + a + e$.	[-32, 32]^n a = 20, b = 0.2, $c = 2\pi$.	0
$f_9(x) = \frac{1}{4000} \cdot \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1$.	[-500, 500]^n	0
$f_{10}(x) = \sum_{i=1}^{n} (-x_i \cdot \sin(\sqrt{\lvert x_i \rvert}))$	[-500, 500]^n	-$n*$ 418.98

4.1.2 Experimental Results

In this section we evolve an EA for function optimization and then we asses the performance of the evolved EA. A comparison with standard GA is performed farther in this section.

For evolving an EA we use f_1 as the training problem.

An important issue concerns the solutions evolved by the EAs encoded into an LGP chromosome and the specific genetic operators used for this purpose. The solutions evolved by the EA encoded into LGP chromosomes are represented using real values (Goldberg, 1989). Thus, each chromosome of the evolved EA is a fixed-length array of real values. By initialization, a point within the definition domain is randomly generated. Convex crossover with $\alpha = {}^1\!/_2$ and Gaussian mutation with $\sigma = 0.5$ are used (Goldberg, 1989).

A short description of real encoding and the corresponding genetic operators is given in Table 2.

Table 2: A short description of real encoding.

Function to be optimized	f:[*MinX*, *MaxX*]$^n \rightarrow \Re$
Individual representation	$x = (x_1, x_2, \ldots, x_n)$.
Convex Recombination with $\alpha = 0.5$	parent 1 – $x = (x_1, x_2, \ldots, x_n)$. parent 2 – $y = (y_1, y_2, \ldots, y_n)$. the offspring – $o = (\frac{x_1+y_1}{2}, \frac{x_2+y_2}{2}, \ldots, \frac{x_n+y_n}{2})$.
Gaussian Mutation	the parent – $x = (x_1, x_2, \ldots, x_n)$. the offspring – $o = (x_1 + G(0,\sigma), x_2 + G(0,\sigma), \ldots, x_n + G(0,\sigma))$, where G is a function that generates real values with Gaussian distribution.

Experiment 1

In this experiment an Evolutionary Algorithm for function optimization is evolved.

There is a wide range of Evolutionary Algorithms that can be evolved by using the technique described above. Since the evolved EA has to be compared with another algorithm (such as standard GA or ES), the parameters of the evolved EA should be similar to the parameters of the algorithm used for comparison.

For instance, standard GA uses a primary population of N individuals and an additional population (the new population) that stores the offspring obtained by crossover and mutation. Thus, the memory requirement for a standard GA is 2 * N. In each generation there will be 2 * N Selections, N Crossovers and N Mutations (we assume here that only one offspring is obtained by the crossover of two parents). Thus, the number of genetic operators (*Crossovers*, *Mutations* and *Selections*) in a standard GA is 4 * N. We do not take into account the complexity of the genetic operators, since in most of the cases this complexity is different from operator to operator. The standard GA algorithm is given below:

Standard GA algorithm

```
S_1. Randomly create the initial population P(0)
S_2. for t = 1 to Max_Generations do
```

```
S_3.      P'(t) = $\phi$;
S_4.    for k = 1 to |P(t)| do
S_5.         $p_1$ = Select(P(t)); // select an individual from the population
S_6.         $p_2$ = Select(P(t)); // select the second individual
S_7.         Crossover ($p_1$, $p_2$, offsp); // crossover the parents $p_1$ and $p_2$
             // an offspring offspr is obtained
S_8.         Mutation (offspr); // mutate the offspring offspr
S_9.          Add offspf to P'(t); //move offspr in the new population
S_10.    endfor
S_11.    P(t+1) = P'(t);
S_12. endfor
```

The best solution generated over all the generations is the output of the program.

Rewritten as an LGP program, the Standard GA is given below. The individuals of the standard (main) population are indexed from 0 to *PopSize* - 1 and the individuals of the new population are indexed from *PopSize* up to 2 * *PopSize* - 1.

```
void LGP_Program(Chromosome Pop[2 * PopSize])
//an array containing of 2 * PopSize individuals
{
Randomly_initialize_the_population();
for (int k = 0; k < MaxGenerations; k++){ // repeat for a number of generations
    // create the new population
    p1 = Select(Pop[1], Pop[6]);
    p2 = Select(Pop[3], Pop[2]);
    o = Crossover(p1, p2);
    Pop[PopSize] = Mutate(o);

    p1 = Select(Pop[3], Pop[6]);
    p2 = Select(Pop[7], Pop[1]);
    o = Crossover(p1, p2);
    Pop[PopSize + 1] = Mutate(o);

    p1 = Select(Pop[2], Pop[1]);
    p2 = Select(Pop[4], Pop[7]);
    o = Crossover(p1, p2);
    Pop[PopSize + 2] = Mutate(o);
    ...
    p1 = Select(Pop[1], Pop[5]);
    p2 = Select(Pop[7], Pop[3]);
    o = Crossover(p1, p2);
    Pop[2 * PopSize - 1] = Mutate(o);

    // pop(t + 1) = new_pop (t)
    // copy the individuals from new_pop to the next population

    Pop[0] = Pop[PopSize];
    Pop[1] = Pop[PopSize + 1];
    Pop[2] = Pop[PopSize + 2];
    ...
    Pop[PopSize - 1] = Pop[2 * PopSize - 1];
  }
  }
```

The parameters of the standard GA are given in Table 3.

Table 3: The parameters of a standard GA for Experiment 1.

Parameter	**Value**
Population size	20 (+ 20 individuals in the new pop)
Individual encoding	fixed-length array of real values
Number of generations	100
Crossover probability	1
Crossover type	Convex Crossover with α = 0.5
Mutation	Gaussian mutation with σ = 0.01
Mutation probability	1
Selection	Binary Tournament

We will evolve an EA that uses the same memory requirements and the same number of genetic operations as the standard GA described above.

Remark. We have performed several comparisons between the evolved EA and the standard GA. These comparisons are mainly based on two facts:

(i) the memory requirements (i.e. the population size) and the number of genetic operators used during the search process.

(ii) the number of function evaluations. This comparison cannot be easily performed in our model since we cannot control the number of function evaluations (this number is decided by evolution). The total number of genetic operators (crossovers + mutations + selections) is the only parameter that can be controlled in our model. However, in order to perform a comparison based on the number of function evaluations we will adopt the following strategy: we will count the number function evaluations/generation performed by our evolved EA and we will use for comparison purposes another standard evolutionary algorithm (like GA) that performs the same number of function evaluations/generation. For instance, if our evolved EA performs 53 function evaluations/generation we will use a population of 53 individuals for the standard GA (knowing that the GA described in section 4.1.2 creates in each generation a number of new individuals equal to the population size).

The parameters of the LGP algorithm are given in Table 4.
The parameters of the evolved EA are given in Table 5.
The results of this experiment are depicted in Figure 1.
The effectiveness of our approach can be seen in Figure 1. The LGP technique is able to evolve an EA for solving optimization problems. The quality of the evolved EA (LGP chromosome) improves as the search process advances.

Table 4: The parameters of the LGP algorithm used for Experiment 1.

Parameter	**Value**
Population size	500
Code Length	80 instructions
Number of generations	100
Crossover probability	0.7
Crossover type	Uniform Crossover
Mutation	5 mutations per chromosome
Function set	$F = \{Select, Crossover, Mutate\}$

Table 5: The parameters of the evolved EA for function optimization.

Parameter	**Value**
Individual representation	fixed-length array of real values.
Population size	40
Number of generations	100
Crossover probability	1
Crossover type	Convex Crossover with $\alpha = 0.5$
Mutation	Gaussian mutation with $\sigma = 0.01$
Mutation probability	1
Selection	Binary Tournament

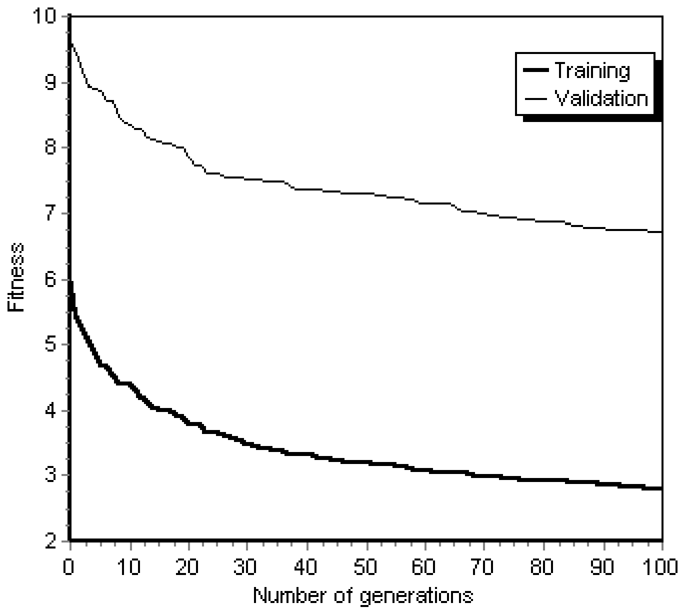

Figure 1: The relationship between the fitness of the best LGP individual in each generation and the number of generations. Results are averaged over 25 runs.

Experiment 2

This experiment serves our purpose of comparing the evolved EA with the standard Genetic Algorithm described in Experiment 1. The parameters used by the evolved EA are given in Table 5 and the parameters used by standard GA are given in Table 3. The results of the comparison are given in Table 6.

Table 6: The results obtained by applying the Evolved EA and the Standard GA for the considered test functions. StdDev stands for the standard deviation. The results are averaged over 500 runs.

Test function	**Evolved EA 40 individuals**		**Standard GA 20 individuals in the standard population + 20 individuals in the new population**	
	Mean	StdDev	Mean	StdDev
f_1	1.06	1.81	13.52	13.68
f_2	104.00	115.00	733.40	645.80
f_3	1.01	0.88	3.95	2.29
f_4	149.02	163.37	756.61	701.52
f_5	6.27	3.52	17.03	7.73
f_6	2440.30	5112.72	113665.57	307109.69
f_7	2.65	1.75	6.16	4.10
f_8	5.08	2.34	10.39	2.90
f_9	1.09	8.07	5.34	4.07
f_{10}	-959.00	182.00	-860.39	202.19

Table 6 shows that the Evolved EA significantly outperforms the standard GA on all the considered test problems.

The next experiment serves our purpose of comparing the Evolved EA with a Genetic Algorithm that performs the same number of function evaluations. Having this in view we count how many new individuals are created during a generation of the evolved EA. Thus, GA will use a main population of 56 individuals and a secondary population of 56 individuals. Note that this will provide significant advantage of the standard GA over the Evolved EA. However, we use this larger population because, in this case, the algorithms (the Standard GA and the Evolved EA) share an important parameter: they perform the same number of function evaluations. The results are presented in Table 7.

The results in Table 7 show that the Evolved EA is better than the standard GA in 3 cases (out of 10) and have the same average performance for 2 functions. However, in this case the standard GA has considerable advantage over the Evolved EA.

In order to determine whether the differences (given in Table 6) between the Evolved EA and the standard GA are statistically significant we use a t-test with 95% confidence. Before applying the t-test, an F-test has been used for determining whether the compared data have the same variance. The P-values of a two-tailed t-test are given in Table 8.

Table 8 shows that the difference between the Evolved EA and the standard GA is

Table 7: The results of applying the Evolved EA and the Standard GA for the considered test functions. StdDev stands for standard deviation. Results are averaged over 500 runs.

Test function	**Evolved EA 40 individuals**		**Standard GA 56 individuals in the standard population + 56 individuals in the new population**	
	Mean	StdDev	Mean	StdDev
f_1	1.06	1.81	1.12	1.98
f_2	104.00	115.00	90.10	108.02
f_3	1.01	0.88	1.10	0.94
f_4	149.02	163.37	111.09	128.01
f_5	6.27	3.52	5.86	3.20
f_6	2440.30	5112.72	2661.83	7592.10
f_7	2.65	1.75	2.32	1.60
f_8	5.08	2.34	5.08	2.24
f_9	1.09	8.07	1.09	7.53
f_{10}	-959.00	182.00	-1010.00	177.00

Table 8: The results of the t-Test and F-Test.

Function	F-Test	t-Test
f_1	0.04	0.58
f_2	0.17	0.05
f_3	0.13	0.15
f_4	7E-8	5E-5
f_5	0.03	0.05
f_6	1E-18	0.67
f_7	0.04	1E-5
f_8	0.32	0.99
f_9	0.12	0.94
f_{10}	0.47	2E-6

statistically significant ($P < 0.05$) for 3 test problems.

Experiment 3

We are also interested in analyzing the relationship between the number of generations of the evolved EA and the quality of the solutions obtained by applying the evolved EA for the considered test functions. The parameters of the Evolved EA (EEA) are given in Table 5 and the parameters of the Standard GA (SGA) are given in Table 3. In order to provide a comparison based on the number of function evaluations we use a main population of 56 individuals for the Genetic Algorithm.

The results of this experiment are depicted in Figure 2 (the unimodal test functions) and in Figure 3 (the multimodal test functions).

Figures 2 and 3 show that the Evolved EA is scalable regarding the number of generations. For all test functions ($f_1 - f_{10}$) we can see a continuous improvement tendency during the search process.

4.2 Evolving EAs for TSP

In this section, an Evolutionary Algorithm for solving the Traveling Salesman Problem (Cormen et al., 1990; Garey et al., 1979) is evolved. First of all, the TSP problem is described and then an EA is evolved and its performance assessed by running it on several well-known instances in TSPLIB (Reinelt, 1991).

4.2.1 The Traveling Salesman Problem

The TSP may be stated as follows.

Consider a set $C = \{c_0, c_1, \ldots, c_{N-1}\}$ of cities, and a distance $d(c_i, c_j) \in \Re^+$ for each pair $c_i, c_j \in C$. The tour $<c_{\pi(0)}, c_{\pi(1)}, \ldots, c_{\pi(N-1)}>$ of all cities in C having minimum length is needed (Cormen et al., 1990; Garey et al., 1979).

The TSP is NP-complete (Garey et al., 1979). No polynomial time algorithm for solving this problem is known. Evolutionary Algorithms have been extensively used for solving this problem (Freisleben et al., 1996; Krasnogor, 2002; Merz et al., 1997).

Experiment 5

In this experiment, an EA for the TSP problem is evolved.

A TSP path will be represented as a permutation of cities (Freisleben et al., 1996; Merz et al., 1997) and it is initialized by using the Nearest Neighbor heuristic (Cormen et al., 1990; Garey et al., 1979). The genetic operators used by the Evolved EA are DPX as crossover and 2-Exchange (Krasnogor, 2002) as mutation. These operators are briefly described in what follows.

The DPX recombination operator copies into offspring all the common edges of the parents. Then it completes the offspring to achieve a valid tour with links that do not belong to the parents, in such a way that the distance between the parents in the newly created offspring is preserved. This completion may be done by using the nearest neighbor information (Freisleben et al., 1996; Merz et al., 1997).

Mutation is performed by applying 2-Exchange operator. The 2-*Exchange* operator breaks the tour by 2 edges and then rebuilds the path by adding 2 new edges (see (Krasnogor, 2002)).

The parameters used by the LGP algorithm are given in Table 9.

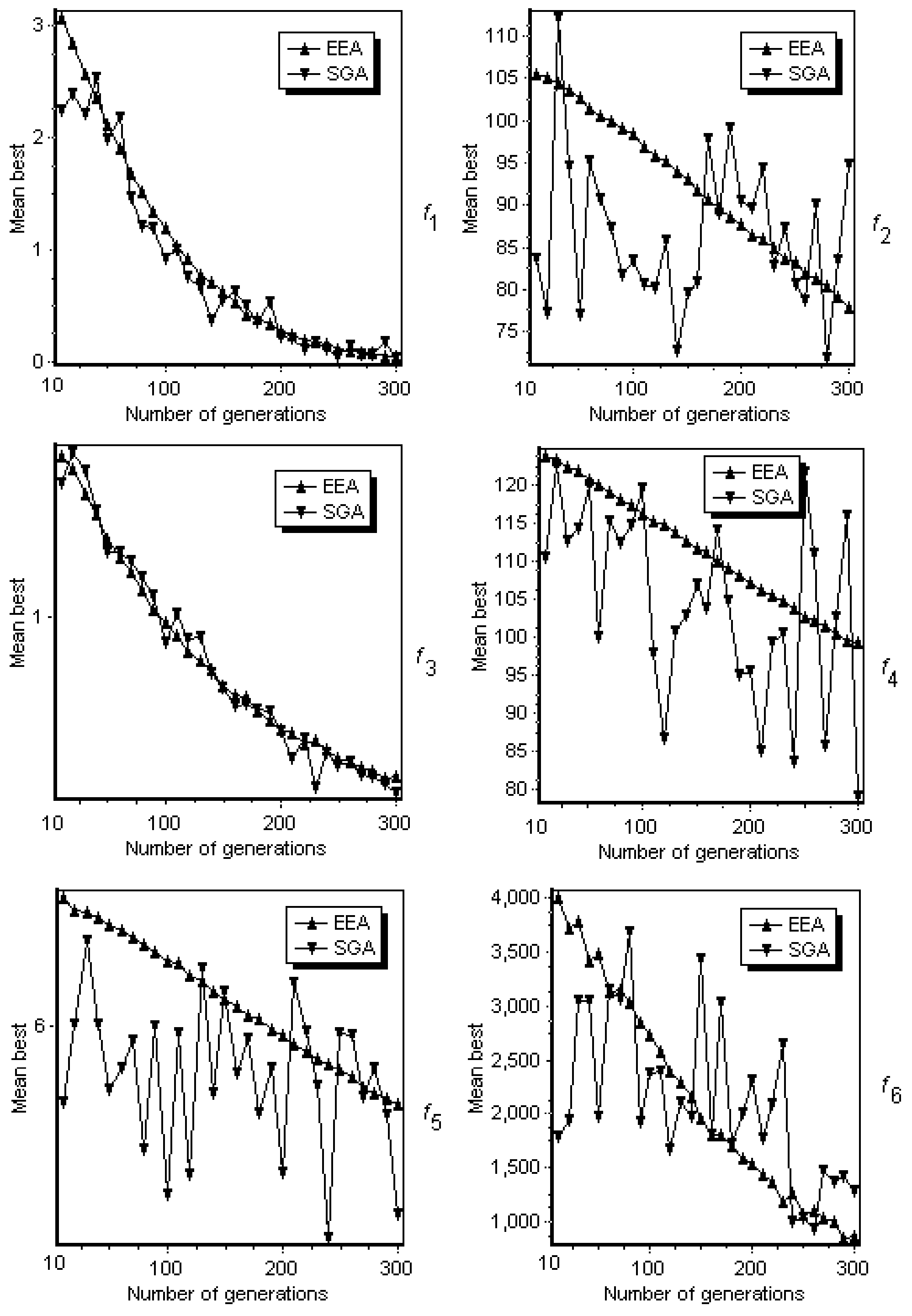

Figure 2: The relationship between the number of generations and the quality of the solutions obtained by the Evolved EA (EEA) and by the Standard GA (SGA) for the unimodal test functions $f_1 - f_6$. The number of generations varies between 10 and 300. Results are averaged over 500 runs.

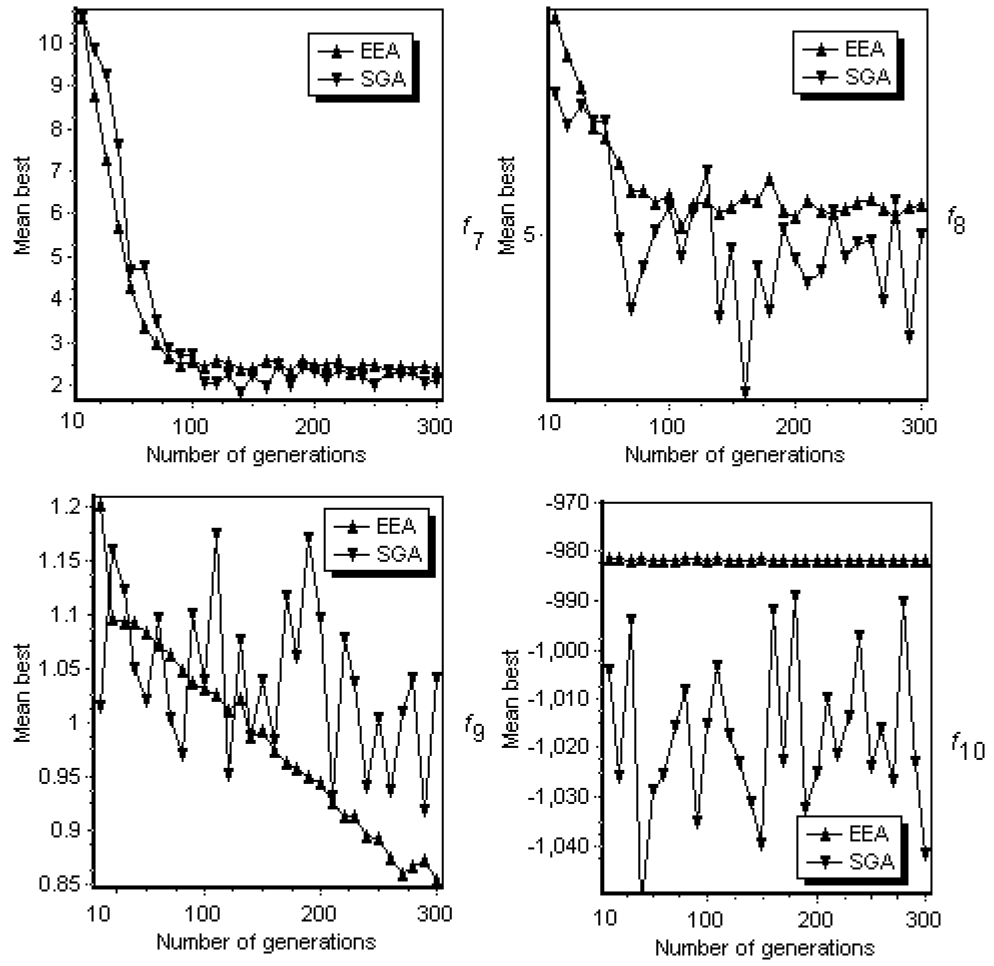

Figure 3: The relationship between the number of generations and the quality of the solutions obtained by the Evolved EA (EEA) and by the Standard GA (SGA) for the multimodal test functions $f_7 - f_{10}$. The number of generations varies between 10 and 300. Results are averaged over 100 runs.

Table 9: The parameters of the LGP algorithm used for Experiment 5.

Parameter	**Value**
Population size	500
Code Length	80 instructions
Number of generations	50
Crossover probability	0.7
Crossover type	Uniform Crossover
Mutation	5 mutations per chromosome
Function set	$F = \{Select, Crossover, Mutate\}$

The parameters of the Evolved EA are given in Table 10.

Table 10: The parameters of the evolved EA for TSP.

Parameter	**Value**
Population size	40
Number of generations	100
Crossover probability	1
Crossover type	DPX
Mutation	2-Exchange
Selection	Binary Tournament

For the training and testing stages of our algorithm we use several problems from the TSPLIB (Reinelt, 1991). The *att48* problem (containing 48 nodes) is used for training purposes. Some other 25 well-known TSP instances are used as the test set.

25 runs for evolving EAs were performed. The time needed for a run was about a day on a PIII -600 MHz computer. An EA yielding a very good performance was evolved in each run. One of these EAs was tested against other 26 difficult instances from TSPLIB.

The results of the Evolved EA along with the results obtained by using the GA described in section 4.1.2 are given in Table 11. Again we count the number of the newly created individuals in a generation of the evolved EA. Thus the standard GA will use a main population of 55 individuals and a secondary population of 55 individuals. In this way both algorithms will perform the same number of function evaluations.

From Table 11 it can be seen that the Evolved EA performs better than the standard GA for all the considered test problems. The difference Δ ranges from 0.38 % (for the problem *bier127*) up to 5.79 % (for the problem *ch130*).

One can see that the standard GA performs very poorly compared with other implementations found in literature (Krasnogor, 2002; Merz et al., 1997). This is due to the weak (non-elitist) evolutionary scheme employed in this experiment. The performance of the GA can be improved by preserving the best individual found so far. We also could use the Lin-Kernighan heuristic (Freisleben et al., 1996; Merz et al., 1997) for genereting very initial solutions and thus improving the search. However, this is beyond the purpose of this research. Our main aim was to evolve an Evolutionary Algorithm and then to compare it with some similar (in terms of the number of genetic operations performed, the memory requirements and the number of function evaluations) EA structures.

4.3 Evolving EAs for the Quadratic Assignment Problem

In this section an evolutionary algorithm for the Quadratic Assignment Problem is evolved.

4.3.1 The Quadratic Assignment Problem

In the quadratic assignment problem (QAP), n facilities have to be assigned to n locations at the minimum cost. Given the set $\Pi(n)$ of all the permutations of $\{1, 2, 3, \ldots n\}$ and two nxn matrices A=(a_{ij}) and B=(b_{ij}) the task is to minimize the quantity

Table 11: The results of the standard GA and Evolved EA for 27 instances from TSPLIB. *Mean* stands for the mean over all runs and *StdDev* stands for the standard deviation. The difference Δ is in percent and it is computed considering the values of the Evolved EA as a baseline. Results are averaged over 100 runs.

Problem	**Standard GA**		**Evolved EA**		Δ
	Mean	*StdDev*	*Mean*	*StdDev*	
a280	3143.64	20.91	3051.35	39.34	3.02
att48	37173.41	656.13	36011.50	650.19	3.22
berlin52	8202.10	83.5758	7989.63	114.98	2.65
bier127	127401.70	1119.56	126914.50	1295.47	0.38
ch130	7124.14	86.98	6734.12	114.05	5.79
ch150	7089.56	17.68	6950.81	97.36	1.99
d198	17578.45	200.50	17127.13	220.12	2.63
d493	40435.86	408.1137	39631.29	407.5115	2.03
d657	59638.29	503.0018	58026.19	591.9782	2.77
eil101	741.91	5.12	728.58	7.92	1.82
eil51	468.91	5.06	461.25	4.22	1.66
eil76	604.31	8.08	587.57	6.82	2.84
fl417	14535.32	223.36	14288.14	198.99	1.72
gil262	2799.96	26.56	2721.58	37.55	2.87
kroA100	24496.34	235.40	23780.42	435.99	3.01
kroA150	31690.82	374.15	30247.58	461.91	4.77
kroA200	34647.90	278.16	33613.78	664.49	3.07
kroB100	24805.07	281.13	23623.80	320.46	5.00
kroB150	30714.54	425.03	29628.97	465.14	3.66
kroC100	23328.12	336.89	22185.22	402.76	5.15
kroD100	24716.68	195.48	24192.46	282.24	2.16
kroE100	24930.71	202.08	24184.65	470.68	3.08
lin105	16937.03	104.73	16324.81	432.41	3.75
lin318	49813.96	454.12	49496.42	590.41	0.64
p654	42827.31	522.99	40853.26	1004.75	4.83
pcb442	59509.64	251.36	58638.23	485.51	1.48
pr107	46996.24	362.04	46175.80	240.38	1.77

$$C(\pi) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \cdot b_{\pi(i)\pi(j)}. \qquad \pi \in \Pi(n).$$

Matrix A can be interpreted as a distance matrix, i.e. a_{ij} denotes the distance between location i and location j, and B is referred to as the flow matrix, i.e. b_{kl} represents the flow of materials from facility k to facility l. The QAP belongs to the class of NP-hard problems (Garey et al., 1979).

Experiment 12

In this experiment, an Evolutionary Algorithm for the QAP problem is evolved.

Every QAP solution is a permutation π encoded as a vector of facilities, so that the value j of the i^{th} component in the vector indicates that the facility j is assigned to location i ($\pi(i) = j$).

The initial population contains randomly generated individuals. The crossover operator is DPX (Merz et al., 2000). Mutation is performed by swapping two randomly chosen facilities (Merz et al., 2000).

The parameters used by the LGP algorithm are given in Table 12.

Table 12: The parameters of the LGP algorithm used for evolving an Evolutionary Algorithm for the Quadratic Assignment Problem.

Parameter	**Value**
Population size	500
Code Length	80 instructions
Number of generations	50
Crossover probability	0.7
Crossover type	Uniform Crossover
Mutation	5 mutations per chromosome
Function set	$F = \{$*Select*, *Crossover*, *Mutate*$\}$

The parameters of the Evolved EA for QAP are given in Table 13.

Table 13: The parameters of the evolved EA for the QAP.

Parameter	**Value**
Population size	40
Number of generations	100
Crossover probability	1
Crossover type	DPX
Mutation	2-Exchange
Selection	Binary Tournament

For the training and testing stages of our algorithm we use several problems from the QAPLIB (Burkard et al., 1991). The *tai10a* problem (containing 10 facilities) is used for training purposes. Some other 26 well-known QAP instances are used as the test set.

25 runs for evolving EAs were performed. In each run an EA yielding a very good performance has been evolved. One of the evolved EAs was tested against other 26 difficult instances from QAPLIB. The results of the Evolved EA along with the results obtained with the GA described in section 4.1.2 are given in Table 14. Since the evolved EA creates 42 individuals at each generation we will use for the standard GA a main population of 42 individuals and an additional population (the new population) with 42 individuals.

Table 14: The results of the standard GA and of the Evolved EA for 27 instances from QAPLIB. *Mean* stands for the mean over all runs and *StdDev* stands for the standard deviation. The difference Δ is shown as a percentage and it is computed considering the values of the Evolved EA as a baseline. Results are averaged over 100 runs.

Problem	**Standard GA**		**Evolved EA**		Δ
	Mean	*StdDev*	*Mean*	*StdDev*	
bur26a	5496026.50	12150.28	5470251.89	14547.86	0.47
chr12a	13841.42	1291.66	12288.16	1545.37	12.64
chr15a	18781.02	1820.54	15280.78	1775.68	22.90
chr25a	9224.26	600.94	7514.98	731.41	22.74
esc16a	71.66	2.44	68.56	1.10	4.52
had12	1682.10	9.40	1666.42	9.42	0.94
had20	7111.54	46.19	7044.46	56.35	0.95
kra30a	107165.20	1730.81	103566.40	2083.24	3.47
kra32	21384.06	368.69	20727.38	374.24	3.16
lipa50a	63348	48.02	63306.37	42.38	0.06
nug30	6902.26	87.82	6774.00	85.40	1.89
rou20	798047.00	7923.00	774706.80	8947.67	3.01
scr20	141237.72	4226.88	128670.50	6012.24	9.76
sko42	17684.18	159.31	17569.20	171.43	0.65
sko49	25968.34	195.09	25895.76	186.31	0.28
ste36a	13562.82	377.62	13022.90	457.48	4.14
ste36b	31875.52	2095.00	29276.02	2254.04	8.87
ste36c	11282157.00	320870.20	10899290.06	432078.30	3.51
tai20a	785232.10	6882.52	759370.20	7808.40	3.40
tai25a	1282398.50	7938.85	1256943.80	9985.62	2.02
tai30a	2010495.90	14351.86	1978437.90	14664.52	1.62
tai35a	2688498.90	17643.60	2649634.68	19598.61	1.46
tai50a	5485928.90	29697.00	5461181.02	28383.97	0.45
tai60a	7977368.30	35081.48	7960123.48	38001.33	0.21
tho30	172923.82	2326.60	168152.84	2722.36	2.83
tho40	281015.00	3890.10	277275.46	3555.32	1.34
wil50	51751.84	250.69	51740.46	269.39	0.02

Table 14 shows that the evolved EA performs better than the standard GA for all the considered QAP instances. The difference Δ ranges from 0.02 (for the $wil50$ problem) up to 22.90 (for the $chr15a$ problem).

We could use some local search (Merz et al., 2000) techniques in order to improve the quality of the solutions, but this is again beyond the purpose of our research.

5 Further Work

Some other questions should be answered about the Evolved Evolutionary Algorithms. Some of them are:

- Are there patterns in the source code of the Evolved EAs? i.e. should we expect that the best algorithm for a given problem contain a patterned sequence of instructions? When a standard GA is concerned the sequence is the following: selection, recombination and mutations. Such a sequence has been given in section 4.1.2. But, how do we know what the optimal sequence of instructions for a given problem is?
- Are all the instructions effective? It is possible that a genetic operation be useless (i.e. two consecutive crossovers operating on the same two parents). Brameier and Banzhaf (Brameier et al., 2001a) used an algorithm that removes the introns from the LGP chromosomes. Unfortunately, this choice proved to be not very efficient in practice since some useless genetic material should be kept in order to provide a minimum of genetic diversity.
- Are all the genetic operators suitable for the particular problem being solved? A careful analysis regarding the genetic operators used should be performed in order to obtain the best results. The usefulness / useless of the genetic operators employed by the GP has already been subject to long debates. Due to the NFL theorems (Wolpert et al., 1997) we know that we cannot have "the best" genetic operator that performs the best for all the problems. However, this is not our case, since our purpose is to find Evolutionary Algorithms for particular classes of problems.
- What is the optimal number of genetic instructions performed during a generation of the Evolved EA? In the experiments performed in this paper we used fixed length LGP chromosomes. In this way we forced a certain number of genetic operations to be performed during a generation of the Evolved EA. Further numerical experiments will be performed by using variable length LGP chromosomes, hoping that this representation will find the optimal number of genetic instructions that have to be performed during a generation.

Another approach to the problem of evolving EAs could be based on Automatically Defined Functions (Koza, 1994). Instead of evolving an entire EA we will try to evolve a small pattern (sequence of instructions) that will be repeatedly used to generate new individuals. Most of the known evolutionary schemes use this form of evolution. For instance the pattern employed by a Genetic Algorithm is:

p_1 = Select (Pop[3], Pop[7]); // two individuals randomly chosen

p_2 = Select (Pop[5], Pop[1]); // another two individuals randomly chosen

c = Crossover(p_1, p_2);

c = Mutate(c);

An advantage of this approach is its reduced complexity: the size of the pattern is considerably smaller than the size of the entire EA.

In order to evolve high high quality EAs and assess their performance an extended set of training problems should be used. This set should include problems from different fields such as: function optimization, symbolic regression, TSP, classification etc. Further efforts will be dedicated to the training of such algorithm which should to have an increased generalization ability.

For obtaining more powerful Evolutionary Algorithms an extended set of operators will be used. This set will include operators that compute the fitness of the best/worst individual in the population. In this case the evolved EA will have the "elitism" feature which will allow us to compare it with more complex evolutionary schemes like steady-state (Syswerda, 1989).

In our experiments only populations of fixed size have been used. Another extension of the proposed approach will take into account the scalability of the population size.

Further numerical experiments will analyze the relationship between the LGP parameters (such as *Population Size*, *Chromosome Length*, *Mutation Probability* etc.) and the ability of the evolved EA to find optimal solutions.

6 Conclusions

In this paper, Linear Genetic Programming has been used for evolving Evolutionary Algorithms. A detailed description of the proposed approach has been given allowing researchers to apply the method for evolving Evolutionary Algorithms that could be used for solving problems in their fields of interest.

The proposed model has been used for evolving Evolutionary Algorithms for function optimization, the Traveling Salesman Problem and the Quadratic Assignment Problem. Numerical experiments emphasize the robustness and the efficacy of this approach. The evolved Evolutionary Algorithms perform similar and sometimes even better than some standard approaches in the literature.

Acknowledgments

The author likes to thanks anonymous reviewers for their usefull sugestions.

The source code for evolving Evolutionary Algorithms and all the evolved EAs described in this paper are available at www.eea.cs.ubbcluj.ro.

References

Angeline, P. J., (1995), Adaptive and Self-Adaptive Evolutionary Computations, *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163, IEEE Press: New York.

Angeline, P. J., (1996), Two Self-Adaptive Crossover Operators for Genetic Programming. In Angeline, P. and Kinnear, K. E., editors, *Advances in Genetic Programming II*, pages 89–110, MIT Press, Cambridge, MA.

Back, T. (1992), Self-Adaptation in Genetic Algorithms, In *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*, pages 263–271, MIT Press, Cambridge, MA.

Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D. (1998). *Genetic Programming -*

An Introduction On the automatic evolution of computer programs and its applications, dpunkt/Morgan Kaufmann, Heidelberg/San Francisco.

Brameier, M. and Banzhaf, W. (2001a). A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining, *IEEE Transactions on Evolutionary Computation*, 5:17–26, IEEE Press, NY.

Brameier, M. and Banzhaf, W. (2001b). Evolving Teams of Predictors with Linear Genetic Programming, *Genetic Programming and Evolvable Machines*, 2:381–407, Kluwer.

Brameier, M. and Banzhaf, W. (2002). Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming, In E. Lutton, J. Foster, J. Miller, C. Ryan and A. Tettamanzi Editors, *European Conference on Genetic Programming IV*, Springer Verlag, Berlin, pages 38–50, 2002.

Burkard, R. E. and Rendl, F. (1991), QAPLIB-A Quadratic Assignment Problem Libray, *European Journal of Operational Research*, 115–119.

Cormen, T. H., Leiserson, C. E. and Rivest, R. R. (1990). *Introduction to Algorithms*, MIT Press.

Edmonds, B. (2001). Meta-Genetic Programming: Co-evolving the Operators of Variation. *Electrik on AI*, 9:13–29.

Freisleben, B. and Merz, P. (1996). A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems, In *IEEE International Conference on Evolutionary Computation 1996*, pages 616–621, IEEE Press.

Garey, M. R., and Johnson, D. S. (1979). *Computers and Intractability: A Guide to NP-Completeness*, Freeman & Co, San Francisco, CA.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

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

Koza, J. R. (1994). *Genetic Programming II, Automatic Discovery of Reusable Subprograms*, MIT Press, Cambridge, MA.

Krasnogor, N. (2002) Studies on the Theory and Design Space of Memetic Algorithms, PhD Thesis, University of the West of England, Bristol, 2002.

Merz, P. and Freisleben B. (1997). Genetic Local Search for the TSP: New Results, In *IEEE International Conference on Evolutionary Computation*, pages 616–621.

Merz, P. and Freisleben, B. (2000), Fitness Landscape Analysis and Memetic Algorithms for the Quadratic Assignment Problem, *IEEE Transaction On Evolutionary Computation*, 4:337–352, IEEE Press, NY.

Nordin, P. (1994). A Compiling Genetic Programming System that Directly Manipulates the Machine-Code. In Kinnear K.E. editors, *Advances in Genetic Programming I*, pages 311–331, MIT Press, Cambridge, MA.

Oltean, M. and Groşan, C. (2003). Evolving Evolutionary Algorithms using Multi Expression Programming. In Banzhaf, W. (et al.) editors, *European Conference on Artificial Life VII*, LNAI 2801, pages 651–658, Springer-Verlag, Berlin, Germany.

Oltean, M. (2003). Solving Even-parity problems with Multi Expression Programming. In Chen K. (et al.) editors, *The* 5^{th} *International Workshop on Frontiers in Evolutionary Algorithm*, pages 315–318.

Oltean, M. and Dumitrescu, D. (2004a). Evolving TSP heuristics with Multi Expression Programming, In Encoding Multiple Solutions in a Linear GP Chromosome. In Bubak, M., van Albada, G. D., Sloot, P., and Dongarra, J., *International Conference on Computational Sciences*, Vol II, pages 670–673, Springer-Verlag, Berlin.

Oltean, M., Groşan, C. and Oltean, M., (2004b). Encoding Multiple Solutions in a Linear GP Chromosome. In Bubak, M., van Albada, G. D., Sloot, P., and Dongarra, J., *International Conference on Computational Sciences, E-HARD Workshop*, Vol III, pages 1281–1288, Springer-Verlag, Berlin.

Reinelt, G. (1991). TSPLIB - A Traveling Salesman Problem Library, ORSA, *Journal of Computing*, 4:376–384.

Ross, B. (2002), Searching for Search Algorithms: Experiments in Meta-search, Technical Report: CS-02-23, Brock University, Ontario, Canada.

Spector, L. and Robinson, A. (2002). Genetic Programming and Autoconstructive Evolution with the Push Programming Language. Genetic Programming and Evolvable Machines, 1:7–40, Kluwer.

Stephens, C. R., Olmedo, I. G., Vargas, J. M. and Waelbroeck, H. (1998), Self-Adaptation in Evolving Systems, *Artificial Life* 4:183–201.

Syswerda, G. (1989). Uniform Crossover in Genetic Algorithms. In Schaffer, J. D. editors, *The* 3^{rd} *International Conference on Genetic Algorithms*, pages 2–9, Morgan Kaufmann Publishers, San Mateo, CA.

Tavares, J., Machado, P., Cardoso A., Pereira F. B., Costa E. (2004). On the Evolution of Evolutionary Algorithms. In Keijzer, M. (et al.) editors, European Conference on Genetic Programming, pages 389–398, Springer-Verlag, Berlin.

Teller, A. (1996). Evolving Programmers: the Co-evolution of Intelligent Recombination Operators. In Angeline, P. and Kinnear, K. E., editors, *Advances in Genetic Programming II*, pages 45–68, MIT Press.

Yao, X., Liu, Y. and Lin, G. (1999). Evolutionary Programming Made Faster. *IEEE Transaction on Evolutionary Computation*, 2:82–102, IEEE Press, NY.

Wolpert, D. H. and McReady, W. G. (1995). No Free Lunch Theorems for Search, Technical Report SFI-TR-05-010, Santa Fe Institute.

Wolpert, D. H. and McReady, W. G. (1997). No Free Lunch Theorems for Optimization. *IEEE Transaction on Evolutionary Computation*, 1:67–82, IEEE Press, NY.