Reports

Machine Learning and Inference Laboratory

Validating Learnable Evolution Model on Selected Optimization and Design Problems

Guido Cervone
Kenneth A. Kaufman
Ryszard S. Michalski

MLI 03-1
P 03-2
June, 2003

School of Computational Sciences

George Mason University

VALIDATING LEARNABLE EVOLUTION MODEL ON SELECTED OPTIMIZATION AND DESIGN PROBLEMS

Guido Cervone, Kenneth A. Kaufman and Ryszard S. Michalski

Machine Learning and Inference Laboratory
George Mason University
Fairfax, VA 22030-4444

{cervone, kaufman, michalski}@gmu.edu
http://www.mli.gmu.edu

Abstract

The recently introduced Learnable Evolution Model (LEM) represents a form of non-Darwinian evolutionary computation that is guided by a learning system. Specifically, LEM "genetically engineers" new populations via hypothesis formation and instantiation. Initial experiments with a preliminary implementation of LEM were highly encouraging, but tentative. This paper presents results from a new study in which LEM was systematically tested on a range of optimization problems and a complex real world design task. The study involved LEM2, a new implementation oriented toward function optimization, and ISHED, an implementation oriented toward engineering design. In all cases of function optimization, LEM2 strongly outperformed tested evolutionary algorithms in terms of the *evolution length*, measured by the number of fitness function evaluations needed to reach the desired solution. This evolutionary speedup also translated to an *execution speedup* whenever the fitness evaluation time was above a small threshold (a fraction of a second). The most important result of the study was that the advantage of LEM2 over the tested Darwinian-style evolutionary methods in terms of evolution length grew rapidly with the growth of the complexity of the optimized function. Experiments with ISHED on problems of optimizing heat exchangers (evaporators) produced designs that matched or exceeded designs produced by human experts. The obtained very strong results from the application of the LEM methodology to two diverse domains suggest that it may be useful also in other application domains, especially, those in which the fitness function evaluation is time-consuming or complex.

Keywords: machine learning, evolutionary computation, function optimization, learnable evolution model, engineering design, multistrategy learning.

Acknowledgments

The authors thank Dr. Piotr Domanski from the National Institute of Standards and Technology for his assistance during the development of the ISHED system, and for the computer simulator that was used for evaluating the performance of evaporator designs. Thanks go also to Janejira Kalsmith and Dr. Elizabeth Marchut Michalski for comments, support, and patience that helped to bring this paper to a successful conclusion. Domenico Napoletani of the Department of Mathematics at the University of Maryland helped in plotting graphs of the functions used in experiments.

This research has been conducted in the Machine Learning and Inference Laboratory at George Mason University. The Laboratory's research has been supported in part by the National Science Foundation under Grants No. IIS-0097476 and IIS-9906858, and in part by the UMBC/LUCITE #32 grant. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the UMBC/LUCITE program or the National Science Foundation.

1 INTRODUCTION

The recently introduced Learnable Evolution Model (LEM) represents a new, non-Darwinian approach to evolutionary computation. The main novelty of LEM is that employs machine learning to guide the process of generating new populations. Specifically, new populations are generated by *hypotheses generation* and *instantiation* rather than by mutation and/or recombination, as in conventional Darwinian-type evolutionary algorithms (Michalski, 1998, Cervone, 1999, Michalski 2000). Initial results from testing of the LEM methodology were highly promising, but involved a preliminary LEM implementation, LEM1, and were limited to a few simple optimization problems.

This paper presents results from a systematic testing of two new LEM implementations: LEM2—oriented toward function optimization, and LEMd-ISHED—oriented toward a class of engineering design problems. In function optimization experiments, LEM2 was compared with selected conventional algorithms both in terms of the *evolution length* (the number of fitness function evaluations or births needed to achieve a desirable solution) and the *evolution time* (the computation time to achieve the solution).

Among important new features and improvement, LEM2 includes an *adaptive anchoring discretization* method, called ANCHOR, for adaptively discretizing continuous variables (Michalski and Cervone, 2001). In engineering design experiments, LEMd-ISHED was applied to a problem of designing optimized evaporators in air conditioning units. In the name of the system, LEMd stands for the LEM method tailored for design problems, and ISHED stand name of the specific system, specifically, "Intelligent System for Heat Exchanger Design." Henceforth, for simplicity, we will use an abbreviation, ISHED, whenever we talk about this specific system.

To make this paper self-contained, we start with a very brief review of the Learnable Evolution Model.

2 A REVIEW OF THE LEARNABLE EVOLUTION MODEL

The Learnable Evolution Model (LEM) represents a fundamentally different approach to evolutionary computation than conventional Darwinian-type evolutionary algorithms. In Darwinian-type algorithms, new individuals are generated through various forms of mutation and/or recombination operators. Such operators are easy to execute and applicable to a wide range of problems. They are, however, semi-random, and take into consideration neither the experience of individuals in a given population (as in Lamarckian-type evolution), nor the experience of populations through the history of the evolution process. As a consequence, Darwinian-type evolutionary algorithms tend to be inefficient, that this diminishes their effectiveness in complex real-world problems.

The novel idea introduced in LEM is that an evolutionary computation can be guided by hypotheses created by a machine learning program that identify areas in the search space that most likely include the sought optimum, or optima. Such hypotheses are created on the basis of the current and, optionally, also past populations of individuals. A general form of LEM may also include periods of conducting a Darwinian form of evolution, when it appears to be useful. Such a form is implemented in the *duoLEM* version, which integrates the intrinsic

LEM mode of operation employing machine learning, called *Machine Learning mode*, and a conventional evolutionary mode, called *Darwinian Evolution mode* (more details on this topic are presented later).

In Machine Learning mode, at each step of evolution, a population is divided into three groups of individuals: High-performing individuals (*H-group*) that score high on the fitness function, Low-performing individuals (*L-group*) that score low on the fitness function, and the rest. The creation of these groups can be done using various methods (Michalski, 2000).

The population from which these groups are selected may be the current population, or a combination of the current and past populations. The selected H-group and L-group are supplied to a learning program that creates general hypotheses distinguishing between these groups. The hypotheses are then instantiated in various ways to produce new, candidate individuals. The candidate individuals compete in terms of their fitness with previously generated individuals for the inclusion in the new population.

As one can see from the above, the generation of new individuals in LEM may take into consideration not only properties of individuals, but also properties of populations of individuals, and even the history of evolution. Thus, LEM uses much more information in generating individuals than Darwinian-type algorithms. Initial experiments have shown that guiding evolutionary processes by hypotheses generated on the basis of whole populations can lead to a dramatic evolutionary speedup (Michalski, 1998; Cervone, 1999; Michalski, 2000).

A general flow diagram of the LEM methodology in presented in Figure 1. A detailed description of individual modules and various aspects of the methodology can be found in (Michalski, 2000). Here we will briefly characterize is most important features. LEM can be run in two different versions: uniLEM and duoLEM.

In uniLEM, the *Machine Learning mode*, described above, is the sole method for generating new populations. In duoLEM, Machine Learning mode is integrated with Darwinian Evolutionary mode that executes some conventional evolutionary algorithm (in which new individuals are generated by a form of mutation and/or recombination operators).

In executing duoLEM, one mode runs until a *mode termination criterion* is met, and then control is switched to the other mode. The mode termination criterion is satisfied where there is insufficient improvement of the fitness function after a certain number of populations, or the allocated computational resources are exhausted.

The main reason for implementing duoLEM is that operators of hypothesis creation and instantiation are more costly computationally than conventional evolutionary operators of mutation and recombination, but are more powerful in selecting individuals. By allowing the interchangeable execution of both Darwinian and Machine Learning modes, duoLEM can utilize the best features of both of them, and also facilitates comparative studies of both approaches.

The following sections describe the LEM2 implementation, its application to a range of function optimization problems, the ISHED implementation, and its application to the optimization of heat exchanger designs.

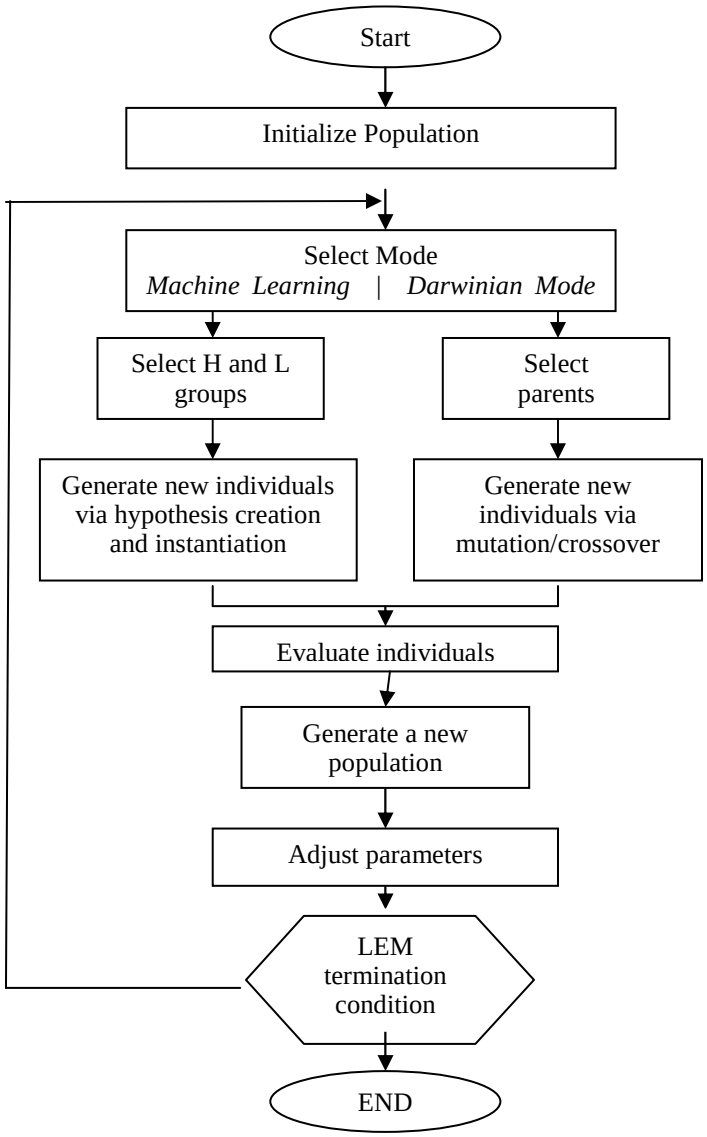


Figure 1. A flowchart of LEM methodology.

3 THE LEM2 SYSTEM

3.1 Overall description

LEM2 is the second implementation of the LEM methodology, oriented toward function optimization. It represents a significant improvement over LEM1, the first, rudimentary implementation (Michalski and Zhang, 1999).

LEM1 employed an earlier, AQ15c, machine learning program in Machine Learning mode (Wnek et al. 1995), and two simple evolutionary algorithms GA1 and GA2, in Darwinian Evolution mode. GA1 and GA2 use a deterministic selection mechanism and a real-valued representation of the variables. The main differences between the two are that GA1 generates new individuals only through a uniform Gaussian mutation operator, while GA2

also uses a uniform crossover operator. Continuous variables are discretized into a fixed number of units. LEM1 was applied to simple function optimization problems (Michalski and Zhang, 2000), and to designing non-linear digital filters (Coletti et al. 1999).

LEM2 employs the AQ18 rule learning program (Kaufman and Michalski, 2000a), a more advanced version of AQ learning than AQ15, and has a number of new features and improvements over LEM1. These include:

A. A new method for discretizing continuous variables. The method, called *Adaptive Anchoring Discretizatio*n, briefly ANCHOR (Michalski and Cervone, 2000), progressively and adaptively discretizes subranges of the attribute domains in the process of evolution.

B. New individuals are generated by instantiating multiple rules, rather than only the strongest rule in the ruleset generated by the learning program. This feature allows the system to explore in parallel several subareas of the search space, and thus is particularly important in the case of multi-modal landscapes.

C. The number of new individuals generated from a single rule is not fixed, but is proportional to the *rule fitness*, defined as the sum of fitnesses of examples covered by the rule. This feature is called the *fitness-proportional instantiation*.

D. In addition to the *population-based* method for selecting the H-group and L-group, LEM2 can also use the *fitness-based* method.

E. The cost of variables (defining individuals) is adjusted dynamically during the evolution process. Each time a variable is included in a set of rules (ruleset) generated by the learning program, its cost is increased. This way, the system gives preference to rules incorporating variables not included in the previously generated ruleset. This feature has proven to be useful in optimizing functions with a very large number of variables.

F. The uniLEM version has been implemented and integrated with Darwinian Evolution mode to form duoLEM.

G. A simple version of the *Start-Over* operation has been implemented for the uniLEM version. Specifically, when the fitness profile function is flat for a certain number of generations, new individuals are created randomly, and inserted into the current population.

H. *Population lookback* and *description lookback*, parameters controlling the selection of the H- and L-groups at each step of evolution, have been implemented (Michalski, 2000).

3.2 Implementation of LEM2

This section explains how individual modules of the LEM methodology, shown in Figure 1, have been implemented in LEM2. The *Initialize Population* module creates individuals randomly. When LEM2 runs uniLEM, the *Select Mode* module always chooses Machine Learning mode, and when it runs duoLEM, it starts in Machine Learning mode (default), then alternates between Machine Learning and Darwinian modes, switching to another mode

when a *mode-termination condition* is satisfied. This is indicated by the lack of improvement of the fitness function for a number of generations specified by the *learn-probe* parameter in Machine Learning mode, and the *dar-probe* parameter in Darwinian mode. The toggling between the two modes continues until the *LEM termination condition* is satisfied (no improvement after *LEM-probe* generations, reaching the allocated limit on the number of evaluations or on the evolution time).

In *Machine Learning mode*, LEM2 supports two methods of selecting H- and L- groups: *Fitness-based* and *Population-Based*. In the Fitness-based method, the H-group and L-group consist of individuals whose fitnesses are above the *High Fitness Threshold* (HFT) and below the *Low Fitness threshold* (LFT), respectively. In the Population-based method, the H-group and L-group consist of portions of the population defined by the *High Population Threshold* (HPT) and the *Low Population Threshold* (LPT), respectively. HPT defines the percentage of the highest performing individuals, and LPT the percentage of the lowest performing individuals to be selected for the H- and L-group, respectively. HFT, LFT, HPT and LPT are controllable parameters.

The *Generate new individuals via hypothesis creation and instantiation* module employs the AQ18 learning system for generating hypotheses distinguishing between H- and L-groups. This choice of this learning system was due to its several features important for LEM (see below) that not available in other machine learning programs. Since AQ-type learning was described in a number of publications (e.g., Michalski, 1973, Cohen and Feigenbaum, 1982), we will describe it here only very briefly, stressing characteristics that are particularly relevant to LEM.

LEM2 employs a recent version of AQ learning, specifically, AQ18, described in detail in (Kaufman and Michalski, 2000a). Given examples representing different concepts, the AQ learning method induces general hypotheses characterizing these concepts and optimizing a multi-component hypothesis quality criterion. The concept examples can be in the form of attribute-value vectors, or structural descriptions (Michalski, 1983). The learned hypotheses are expressed in the form of *attributional rules* whose conditions may include internal disjunctions of attribute values, ranges of values, and other constructs, unlike conventional decision rules, whose conditions are all limited to the <attribute-relation-value> form (Michalski, 2001). Using more expressive conditions makes the representation language not only more powerful, but facilitates the instantiation of rules into individuals whose properties vary only slightly. This feature is attractive for implementing an evolutionary process in LEM.

Another useful characteristic of AQ learning is that given a set of training examples, it can generate different types of rules, depending on the program parameters. The rules may be very general, very specific, or of intermediate generality. Rules of high generality tend to be preferable at the beginning of the evolutionary process, when the search space has not been investigated much. At later stages, when the areas of the space that most likely contain the global optimum have been already identified, more specific rules tend to be preferable.

AQ18 takes as input the H-group and L-group, a specification of the types and domains of the variables, and, optionally, control parameters that define the type of rules to be learned. As output, it produces a set of attributional rules with annotations characterizing the rules (Kaufman and Michalski, 2000a). Each such rule is a conjunction of attributional conditions

that specify subsets of values (or ranges of values in the case of continuous attributes) an attribute can take to satisfy the condition.

Figures 2 and 3 illustrate an example of an input to and an annotated output from AQ18, respectively. The parameters listed in Figure 2 require that ambiguous individuals (common both to H-group and L-group) be ignored (ambig = ignore), that rules be maximally specific (genlevel = spec), and that training examples be treated as not containing noise (noise = no). In the description of variables, "lin" stands for linear-type attributes (discrete attributes whose domains are totally ordered sets); "size" specifies the total number of values in the attribute domain.

Since AQ18 assumes that all attributes are discrete, continuous attributes need to be discretized. In LEM1, the discretization was done prior to the learning process. In LEM2, a new method is used, called ANCHOR, which adaptively discretizes attributes during the learning process (Section 3.3).

Parameters

run	ambig	genlevel	noise
1	empty	spec	no

Variables

#	type	size	name
1	lin	15	x1.x1
2	lin	15	x2.x2
3	lin	15	x3.x3
4	lin	15	x4.x4

H-group events

#	x1	x2	x3	x4	Weight
1	7	7	7	8	12
2	7	6	6	7	9
3	7	6	7	8	7
4	6	6	7	8	5
5	6	7	6	8	5

L-group events

#	x1	x2	x3	x4	Weight
1	6	6	14	8	5
2	6	7	14	8	3
3	7	7	14	7	1

Figure 2. AQ18 input.

A rule generalizing the H-group:

[x1=6..7] & [x2=6..7] &
[x3=6..8] & [x4=7..8]

This learning process used:
System time: 0.0 seconds
User time: 0.0 seconds

Note: The above is a *characteristic* description of the H-group (stating characteristics common to all H-group examples). The attribute values in events and in the rule conditions are numerical symbols representing *ranges* of numerical values of these variables, not their original values in the raw data. These ranges have been determined in the process of adaptive anchoring discretization (Sec 3).

Figure 3. AQ18 output.

The learned hypotheses (attributional rules) are used to generate new individuals by randomizing variables within the ranges of values defined by the rule conditions. If a rule does not refer to some variable, it means that this variable was not needed for distinguishing between the H-group and the L-group. A problem then arises as to what values should be assigned to such variables when generating new individuals. There are different methods for handling this problem. In LEM2, we chose a very simple solution: variables not included in the rule are instantiated by assigning to them values occurring in randomly selected individuals from the current population. This is a conservative method that does not introduce values not already present in the population.

In *Darwinian Mode* new individuals are generated by selecting representative individuals, and mutating and/or recombining them. To make the paper self-contained, below is a brief description of these operators. The *Select parents* operator selects representative individuals

(parents) from the current population according to some selection method, such as fitness proportional selection, uniform selection, tournament selection, etc.

The *Generate New Individuals* module creates new individuals by mutation and/or recombination. The mutation operator takes one individual (the parent) and generates one offspring (the child). There are many different techniques for mutation, but they all share the fact that they are semi-blind. They use a random distribution to choose which variables to mutate, and how to mutate them. The recombination operator, also known as crossover, takes two individuals (the parents) as input, and generates one offspring (the child). The values of variables of the offspring are a mix of the values of variables of the two parents. As for mutation, there are several different methods for recombination; they typically use a random distribution to choose the crossover point. A more detailed explanation of how mutation and crossover work can be found in (Baeck, Fogel and Michalewicz, 1997).

As shown in Figure 1, certain modules are common to both modes of LEM2. These are:

- The *Evaluate Individuals* module, which determines the fitness of each individual. If fitness function is defined by a closed formula, it can be evaluated very fast. When its computation requires running a simulation model, such a process may be costly and/or time-consuming.

- B. The *Generate New Population* module, which creates a new population by combining individuals from the previous population with newly generated individuals generated. Again, many methods for executing this step have been explored by the evolutionary computation community.

- The *Adjust Parameters* module, which sets the settings for both machine learning and Darwinian mode. LEM keeps statistics regarding the number of successful births, the change in the highest-so-far fitness score, and so forth. Using these statistics, it can adjust its behavior in the evolutionary process. For example, depending on the stage of evolution, generating more general or more specific rules may be more desirable, the parameters controlling the selection of H-group and L-group may need to be changed, or the mutation rate in the Darwinian evolutionary mode may need to be adjusted.

- The *LEM Terminating Condition* module, which determines when to stop the execution of the program. The condition can be defined by the maximal number of births (fitness evaluations) to be executed, by the number of steps with no improvement in the fitness function, or by a limit on the execution time. Because descriptions generated by machine learning are only hypotheses about areas containing the solution, the global optimum may be missed in a given execution of the algorithm. When such a possibility is suspected, a *Start-Over* operation is executed that restarts LEM with a different initial population. This new initial population can be composed of new individuals generated at random, or by a new instantiation of hypotheses learned in the previous runs. Previous hypotheses can be used to avoid generating samples in areas of the landscape that are known to be low fitness.

3.3 Adaptive Discretization Method: ANCHOR

LEM2 uses a novel way of handling continuous attributes, namely the *adaptive discretization method*, called ANCHOR. The underlying idea is to start with a coarse discretization of each continuous variable, and in subsequent steps of evolution increase the discretization precision in only selected ranges of the variable values. Such ranges are determined by the hypotheses generated at each step. A detailed description of the ANCHOR method is in (Michalski and Cervone, 2001). To make this paper self-contained, a brief review of the method follows.

The name ANCHOR signifies the fact that the attributes are discretized into consecutively more precise discrete values that are rounded to the nearest whole numbers (anchors). This feature is reflective of the human preference for representing numbers/measurements with the minimum number of digits that are needed for a given problem. The adaptive discretization thus avoids excessive precision, and by that decreases the computational complexity of the learning process.

Let us assume that a continuous variable x_i ranges over the interval [*min*,…,*max*], where min is the smallest possible value, and *max* is the largest value of x_i to be considered. The first order discretization of x_i is to split the interval into fir*st order units* represented by single decimal digits m, m+1, m+2, … , M-1, M, where m is floor(*min*) and M is ceiling(*max*). The so-discretized variables are then used for determining hypotheses differentiating between the H- and L-groups.

The references of the variables in the attributional conditions are used as indicative of the first order units that may need further discretization. These units are discretized in a similar fashion as above to create next order units. LEM is iteratively applied until the range of the fitness profile function is smaller than *delta*, a parameter of the method.

To illustrate this process, let the domain of x_i be a range defined by *min* = -2.3 and *max* = 14.7. The first order discretization is {-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}. Suppose that the LEM-generated inductive hypotheses indicate that x_i needs to be approximated more precisely for values 3 to 5. The second order discretization would then be: {-3, -2, -1, 0, 1, 2, (3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9), (4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9), 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}.

This process will continue until the range of the fitness profile function becomes delta-small. As this example indicates, the growth of the domain size of a variable is adjusted to the needs of the task. Only those subranges are discretized that are found to need higher precision of representation.

To test the ANCHOR method, it was experimentally compared with a χ^2-based method (Kerber, 1992). Results below illustrate its performance in comparison to the χ^2-based method on the problem of minimizing the sphere function:

$$f(x) = \sum_{i=1}^{100} x_i^2$$

where each of 100 variables is bound between –5.123 and 5.123. The function has two maxima for each dimension, one at x_i = -5.123 and second at 5.123. Thus, there are in total 2^{100} possible solutions. Every solution requires three decimal places of precision.

Six experiments were performed, each consisting of five runs that differed in their initial population. In each run, the population consisted of 100 individuals, and each experiment lasted for 10,000 births. Out of six experiments, five used the χ^2 method (with 5, 10, 50, 100 and 1000 discretization units for each of 100 variables) and the sixth experiment used ANCHOR. The results are presented in Figure 4.

The vertical axis represents the total elapsed time to find the solution in seconds, while the horizontal axis indicates the average accuracy of the solutions from five runs. The accuracy of each solution was measured here by the ratio of the highest fitness solution found in the given run to the globally maximal solution, expressed in percentage. As shown in Figure 4, increasing the number of discretization units in the χ^2 method yields a more accurate solution at the expense of the execution time. The execution time grew very rapidly with the number of discretization units. For example, for 25 units the process took about 15 seconds, giving accuracy of 78%; and for 1000 units, it took about 240 seconds, giving the accuracy of 99% (these are averages of five runs). The best accuracy, 100%, was obtained by ANCHOR, in about 20 seconds. This means that ANCHOR produced 100% accurate solutions in each of the five runs.

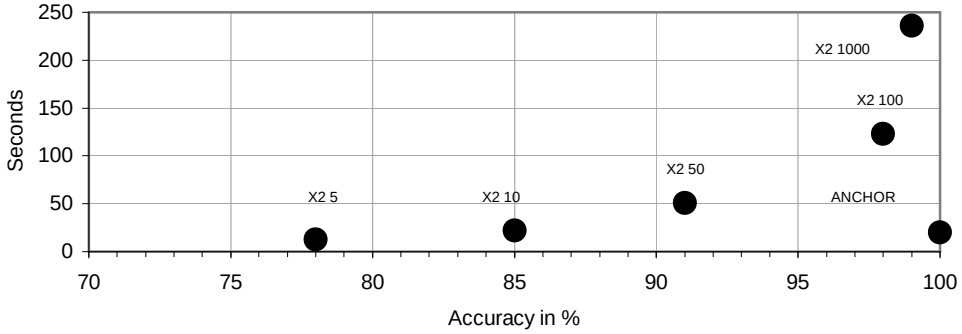


Figure 4. Accuracy and elapsed time for ANCHOR, and χ^2 method using 5, 10, 50, 100 and 1000 discretization units for each of 100 variables.

As shown in Figure 4, ANCHOR outperformed the χ^2 discretization method in accuracy regardless of the number of discrete units used in it. When the number of discretization units was 1000, which is sufficient to capture the exact value of the solution, the χ^2 method still did not produce 100% accuracy, while its execution time was about 12 times longer than that of ANCHOR achieving 100% accuracy (240 sec vs. 20 sec). ANCHOR is implemented in C++ (like the entire LEM2 system). Experiments were performed on a SUN Sparc20 with 64MB of RAM, running Solaris 5.7. LEM2 implements both ANCHOR and χ^2 methods to facilitate experimentation.

4 TESTING LEM2 ON FUNCTION OPTIMIZATION PROBLEMS

4.1 Problem Specification

LEM2 was systematically tested on a range of function optimization problems. The problems involved several functions widely used by the Evolutionary Computation community for benchmarking different evolutionary computation algorithms, and thus were particularly attractive for doing a comparative study of the LEM2 performance.

Due to space limitations, we present here just a sample of representative results concerning the optimization of three functions: the Rosenbrock function, the Rastrigin function, and the Gaussian Quartic function. These functions were selected because they have widely different properties, and the results from their optimization by conventional, Darwinian-type methods are readily available, thus enabling a comparison of results. To test the scalability of LEM2 to complex optimization problems, we extended the number of variables in each of these functions.

The comparison of LEM2's results with those obtained by Darwinian-type evolution algorithms was made both in terms of the *evolution length*, defined as the number of evaluations (or births) needed to determine the target solution, and the *evolution time*, defined as the execution time required to achieve this solution. The reason for measuring both characteristics is that LEM and Darwinian type algorithms represent tradeoffs between the complexity of the population generating operators and the evolution length. Operators of hypotheses generation and instantiation used in LEM2 are more computationally costly than operators of mutation and/or crossover, but LEM2's evolution length is typically much shorter than that of Darwinian evolutionary algorithms.

Using the concept of evolution length, we analyzed the *evolutionary speedup* of compared algorithms. To define this concept, assume that the function to be optimized (minimized or maximized) can be transformed (by inversion and/or adding a constant) into a form in which the function optimum is the maximum of the transformed function, and the function minimum is 0. Thus, the optimized function can be considered to be a positive-only fitness function, and the optimization problem becomes a problem of finding a solution with the maximum fitness.

Let us introduce a parameter δ as a measure of the relative distance between the highest fitness solution found by an algorithm and the globally highest fitness solution. Specifically, let δ be the ratio of the difference between the fitness of the globally maximal solution and the best solution found by the algorithm, divided by the fitness of the globally maximal solution. For example, if the global maximum of a function is 100 and the best solution found is 99, then δ is 0.01. By *δ-close solution* is meant a solution that differs from the global optimal by at most δ. For the purpose of evaluating performance of LEM2 and other algorithms in function optimization, it is assumed that δ is a controllable parameter, and the evolution process continues until a δ-close solution (also called the *target solution*) is found.

The relative performance of two algorithms is characterized by two measures: the *evolution speedup* and the *execution speedup*. The evolution speedup of algorithm A over B for a given δ is defined as the ratio, expressed in percentage, of the number of births (or fitness

evaluations) required by B to the number of births required by A to achieve the δ-close solution. The execution speedup of algorithm A over B for a given δ is defined as the ratio of the total computation time required by B to the total computation time required by A to achieve the target solution.

For each of the three functions to be optimized, experiments were performed assuming different numbers of function arguments (variables), specifically, 10 or 20, 50 and 100. For each function and each number of variables, we measured the evolution length and the execution time of the programs compared. Experiments were performed with different sizes of the population, and repeated 10 times (runs) with different random initial populations. The results represent the highest fitness solution from the 10 runs performed by LEM2 and, for comparison, by ES, a program implementing an evolutionary strategy method.

The ES method was chosen because it is widely used in the field of evolutionary computation. ES employs a real-valued vector representation of individuals and the deterministic selection (i.e., each parent is selected, and then mutated a fixed number of times, defined by the *brood* parameter). The mutation is done according to the Gaussian distribution, in which the mean is the value being mutated, and the standard deviation is a controllable parameter, called the *mutation rate*. Each variable has a 1/*L* probability of being mutated, where *L* is the total number of variables defining an individual. New individuals and their parents are sorted according to their fitness, and the *popsize* highest-fitness individuals are included in the next generation, where *popsize* is a numeric parameter denoting a fixed population size.

The ES program was implemented according to the description in (Baeck, Fogel and Michalewicz, 1997). It was implemented anew, rather than employing some existing program, as this has allowed us to closely integrate it with LEM2. Such integration facilitates running LEM2 and ES with identical parameters, that is, with identical initial populations, the same random seed for random number generator, and the same mutation and recombination operators when executing duoLEM and ES.

For the Rosenbrock and Rastrigin function optimization problems, we found on the Web their solutions, and for those solutions we determined comparable solutions from LEM2. These results are presented when discussing individual experiments. The website that reports and maintains these results can be found on the Internet at URL *http://www.maths.adelaide.edu.au/Applied/llazausk/alife/realfopt.html*. This website is a repository of the allegedly best results achieved by evolutionary algorithms on various benchmark functions found in the literature.

4.2 Minimization of the Rosenbrock Function

These experiments concern the minimization of the Rosenbrock function (Rosenbrock, 1960) for different numbers of variables, n = 10, 50 and 100, ranging between –5.12 and 5.12:

$$Ros(x_1, x_2, .. x_n) = \sum_{i=1}^{n-1} (100 \cdot (x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$

This is a non-trivial optimization problem (especially for n=100) because the function has a very narrow and sharp ridge, and runs around a parabola, so the variables are interrelated

(Figure 5). The global minimum of the function, which is 0, is achieved when all variables take value 1.

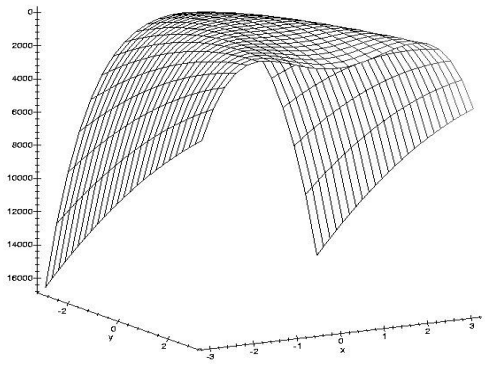


Figure 5. An inverted 2D projection of the Rosenbrock function.

For comparison, the ES program was also applied to the same problem. Table 1 presents best results from runs of LEM2 and ES for different values of δ (δ0, δ0.01, δ0.1 are abbreviations for δ=0, δ=0.01, δ=0.1, respectively). Entries in rows 3 and 4 present the best value of the *δ-close evolution length* for the corresponding program. By the *δ-close evolution length* is meant the number of births after which the obtained solution becomes δ-close. The third row shows the LEM2/ES evolution speedup for the corresponding δ.

Number of variables	**10**			**50**			**100**		
	δ0	**δ0.01**	**δ0.1**	**δ0**	**δ0.01**	**δ0.1**	**δ0**	**δ0.01**	**δ0.1**
LEM2 Evolution Length	850	850	850	7100	7100	7100	14400	12000	11000
ES Evolution Length	11000	10000	9000	110000	100000	88000	235000	220000	199000
LEM2/ES Evolution Speedup	13	12	11	15	14	12	16	16	18

Table 1. Peformance of LEM2 and ES on minimizing the Rosenbrock function of 10, 50 and 100 continuous variables.

The best solution from LEM2 for each value of δ and each number of variables occurred when the population size was 100, and the HPT and LPT were both set to 30%. The best run of ES occurred when the population size was 100, and the mutation rate was 0.7. As Table 1 shows, in every case, the evolution speedup LEM2/ES was greater than 1, ranging between 11 and 18, and the LEM2/ES Evolution Speedup grew with the function complexity (here, the number of variables).

LEM2's performance was also compared with the best published results for the Rosenbrock function using conventional evolutionary algorithms (Eschelman and Shaffer, 1993). These results concern cases with far fewer variables (2 and 4) than those in Figure 6 (100 variables). Table 2 shows the δ-close evolution length for δ=0 (that is, the number of births

needed to reach the global minimum), and the evolution speedup of LEM2 over the compared method. In the case of two variables, the best result had been achieved using the CHC+BLX method, and required 4893 births. In contrast, LEM2 found the global minimum in this case using only 101 births (fitness evaluations). Thus, the evolution speedup is nearly 50.

Case: Rosenbrock function of 2 variables	δ=0
LEM2 (uniLEM)	101
CHC+BLX	4893
Evolution Speedup LEM2/CHC+BLX	~49

Table 2. Results for the Rosenbrock function of 2 variables.

Case: Rosenbrock function of 4 variables	
LEM2 (uniLEM)	*δ=0:* 281
Breeder GA	*δ=0.1:* 77,000
Evolution Speedup LEM2/GA	≥ 750

Table3. Results from the minimization of the Rosenbrock function of 4 variables.

In the case of four variables, the best-published result was achieved by a breeder GA that required about 250,000 evaluations (births) for δ=0.1 (Schlierkamp-Voosen and Muhlenbein, 1994). LEM2 found the global minimum for a smaller δ (δ=0) with only 281 evaluations, that is, the speedup of LEM2 over GA was at least 750. Table 3 summarizes the results. These results indicate that LEM2 was able to locate the area of the landscape with the global optimum very rapidly. To see how the evolution speedup LEM2/ES changes with the complexity of the problem (the number of variables in the optimized function), we determined the evolution length for LEM2 and ES for 10, 50 and 100 variables.

Figure 6 shows the results. As seen in this figure, the evolution length of ES increases with the number of variables at a significantly faster rate than of LEM2 (the target solution was defined by δ = 0.1). In the range tasted, from 10 to 100 variables, LEM's evolution length grew very slowly and approximately linearly, while ES's evolution length grew very fast and seems to grow exponentially. For 10 variable, there was little difference between LEM2 and ES. For 100 variables, ES required about 200,000 births, while LEM2 required about 15,000.

This is a very important *result of the study*, because it suggests that the advantage of LEM2 over ES in terms of the evolution length grows rapidly with the complexity of the problem. This result, if confirmed by experiments using other problems, would indicate that LEM2,

and LEM methodology in general may be particularly useful for very complex function optimization problems.

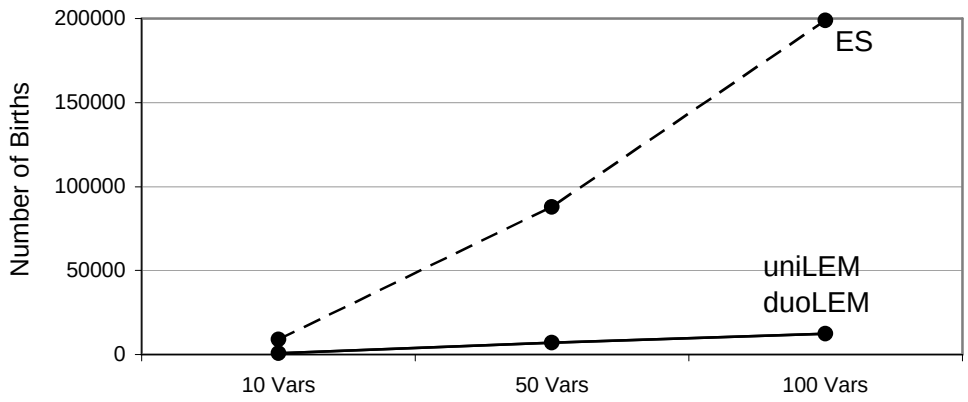


Figure 6. The increase of the evolution length with the number of variables in the Rosenbrock function optimized by ES and LEM2, using both uniLEM and duoLEM versions.

The next series of experiments concerned the evolution time consumed by the compared algorithms (the total computational time for obtaining the target solution). These experiments are important because the execution time of operators generating new individuals in LEM2 is higher than ES, because of the higher complexity of hypothesis formation and instantiation operators than that of mutation and recombination operators. The evolution execution time depends on the time of generating individuals and the time of evaluating individuals. Therefore, even when the evolution length is shorter (fewer evaluations), the execution time may be longer (if the generation time is long). In the experiments, we have controlled the evaluation time by introducing a variable, called *evaluation delay*. The total evaluation time of an individual is thus the sum of the actual evaluation time and the evaluation delay. A question that may arise is why a delay was introduced, rather than using a set of more complicated evaluation functions that require more time to evaluate. The answer is that adding an increasing delay, we can better observe the relationship between evaluation length and evaluation time for LEM2 and ES.

Figures 7, 8 and 9 present the dependence of the *execution speedup* LEM2/ES on the evaluation delay. The execution speedup LEM2/ES is defined by the ratio of the evolution time of ES to the evolution time of LEM2 to achieve the target solution. Thus, a speedup of 20 means that LEM2‘s evolution time was 20 times shorter than that of ES to achieve the same solution. A dashed horizontal line in all figures indicates the speedup of 1, i.e., when the execution times of ES and LEM2 are equal. The portion of the curves under this line indicates evaluation delays for which ES was faster than LEM2, and the portion above it indicates evaluation delays for which LEM2 was faster (by the number indicated on the vertical axis). In each graph, the top horizontal line corresponds to the evolution speedup. The figures show that with the growth of the evaluation delay, the execution speedup is converging to the evolution speedup. In these experiments, the target solution was specified by $\delta = 0.1$.

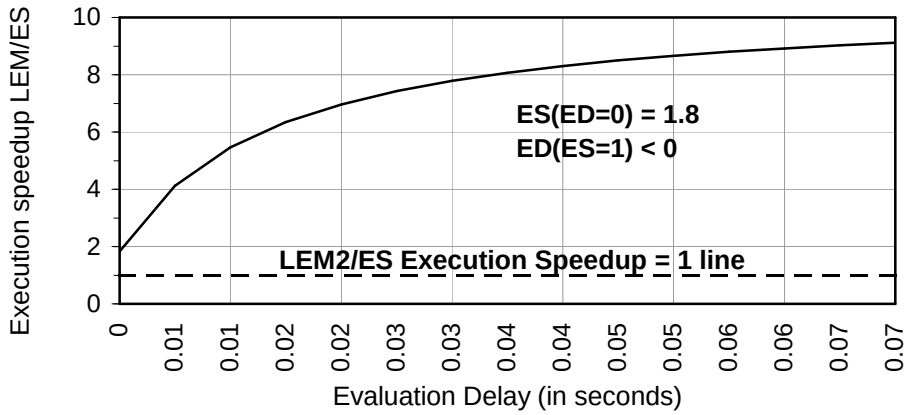


Figure 7. The dependence of the execution speedup of LEM2/ES for the Rosenbrock function of 10 variables on the evaluation delay parameter.

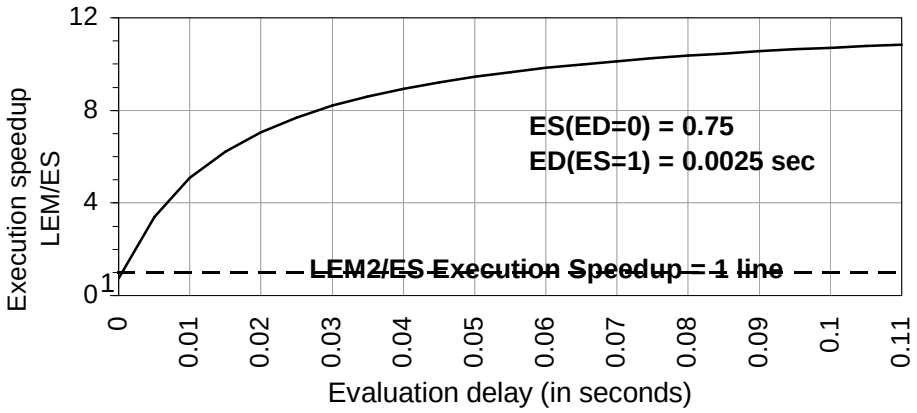


Figure 8. The dependence of the execution speedup of LEM2/ES for the Rosenbrock function of 50 variables on the evaluation delay parameter.

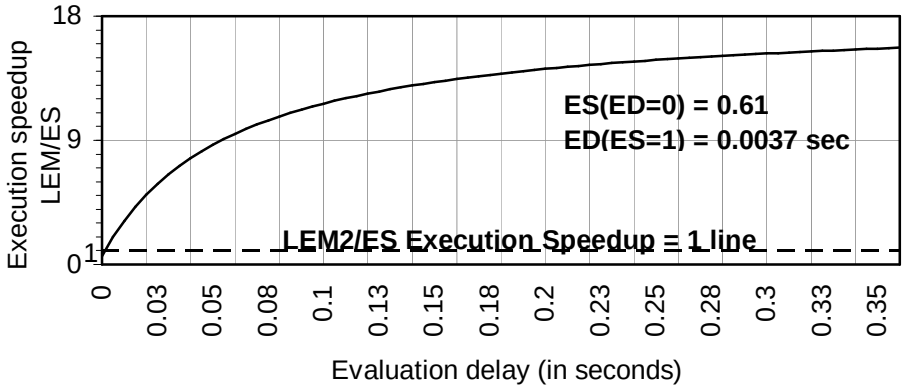


Figure 9. The dependence of the execution speedup of LEM2/ES for the Rosenbrock function of 100 variables on the evaluation delay parameter.

The value ES(ED=0) represents the evolution speedup, LEM2/ES, for the evaluation delay equal 0. The value ED(ES=1) represents the evolution delay (in seconds) needed for the evolution speedup to reach 1. If the execution speedup LEM2/ES is greater than 1, then this is denoted by ES(ED=0) < 0, meaning that the evaluation time has to be negative to produce the same execution time for LEM2 and ES.

As shown in the figures, in the worst case, it was sufficient to add about 4 milliseconds to the evaluation function to achieve the evolution speedup LEM2/ ES greater than 1.

4.3 Minimization of the Rastrigin Function

This experiment concerned the minimization of the Rastrigin function:

$$Ras(x_1, x_2, ..x_n) = n*10 + \sum_{i=1}^{n} (x_i^{\,2} - 10*\cos(2*\pi*x_i))$$

for the number of arguments, n, set to 20, 50 and 100, and x_i ranging between –5.12 and 5.12.

The Rastrigin function has many local optima, and it is easy to miss the global solution (Figure 10). In these experiments, both uniLEM and duoLEM versions were employed, and their results were compared with previously published results obtained by a parallel GA with 16 subpopulations and 20 individuals per subpopulation (Muhlenbein, Schomisch, and Born, 1991).

Table 4 presents the best results achieved by LEM2, by ES, and by the parallel GA in terms of the evolution length (rows 3, 4, and 5, respectively), and the evolution speedup (rows 6 and 7) for different values of δ. The best solution for LEM2 occurred when the population size was 100, and the HPT and LPT were set to 30%. The best solution for ES occurred when the population size was 100, and the mutation rate was 0.7. Cells marked by a "?" denote cases for which results were not published.

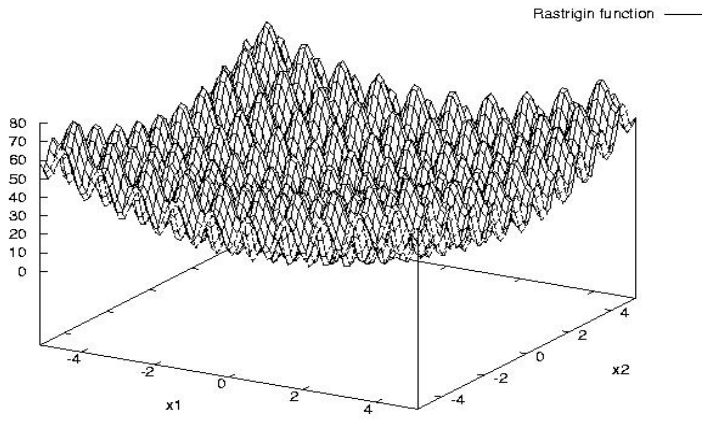


Figure 10. A 2D projection of the Rastrigin function.

Number of Variables	20			50			100		
	δ0	δ0.01	δ0.1	δ0	δ0.01	δ0.1	δ0	δ0.01	δ0.1
LEM2	1100	1100	1100	3000	2800	2800	8500	6900	6700
ES	12000	11200	11000	48000	43000	40000	173000	130000	118000
PGA	?	?	10000	?	?	39000	?	?	100000
Evolution Speedup LEM2/ES	11	10	10	16	15	13	20	19	18
Evolution Speedup LEM2/PGA	?	?	9	?	?	14	?	?	15

Table 4. Evolution speedups of LEM2/ES for the Rastrigin function of 20, 50 and 100 variables.

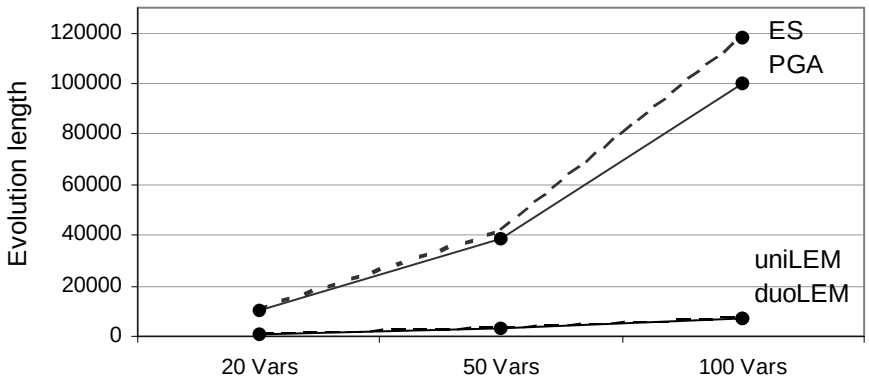


Figure 11. The increase of the evolution length (number of births) with the increase of the number of variables when optimizing the Rastrigin function using ES, parallel GA, and LEM2 (both uniLEM and duoLEM).

As in the previous problem, we determined the dependence of the evolution length on the number of function arguments (variables) for the different methods (Figure 11). As shown in the figure, the result in analogous to the one shown in Figure 6, namely, that the evolution length of ES and PGA increases with the number of variables at a significantly faster rate than that of LEM2 (the target solution was defined by $\delta = 0.1$). This result thus confirms the previous result that LEM2's advantage in terms of the evolution length grows with the complexity of the function, as measured by the number of variables.

We also conducted experiments to determine the dependence of the execution speedup of LEM2 over ES on the function evaluation time. To control the function evaluation time, we introduced a variable delay to the function evaluation time.

Figures 12, 13 and 14 present the results. As shown in these figures, the evaluation delay, ED, needed to achieve ES =1, was in every case very small (it varied between 0.0043 and .0025 seconds). For larger evaluation delays, the execution speedup quickly grew and converged to the evolution speedup (9 in the case of 20 variables, about 13 for 50 variables, and about 15 for 100 variables).

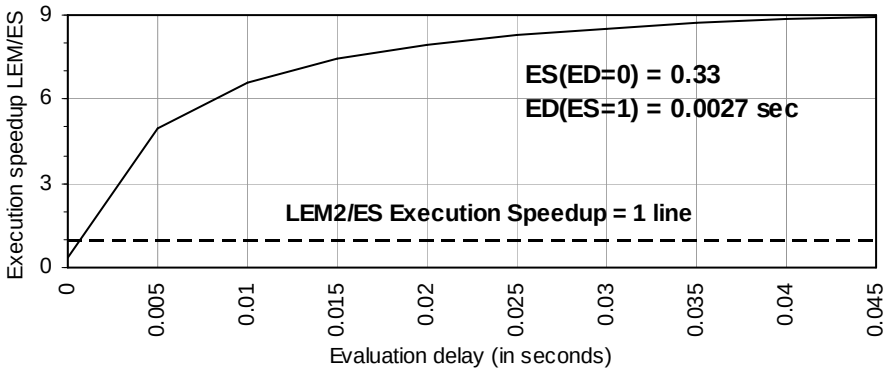


Figure 12. The dependence of the execution speedup of LEM2/ES for the Rastrigin function of 20 variables on the evaluation delay parameter.

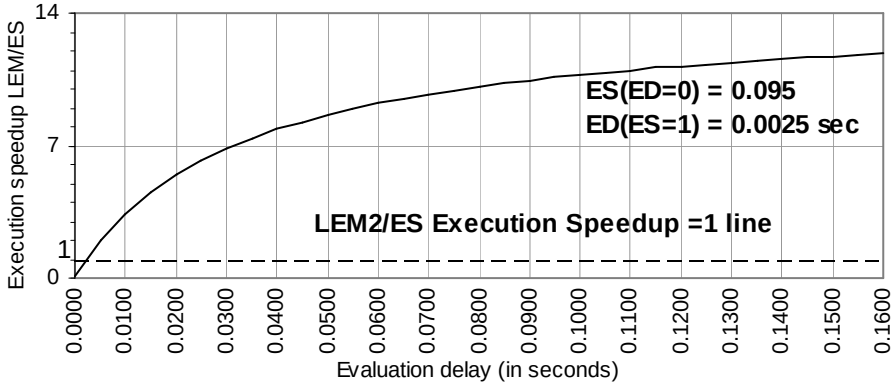
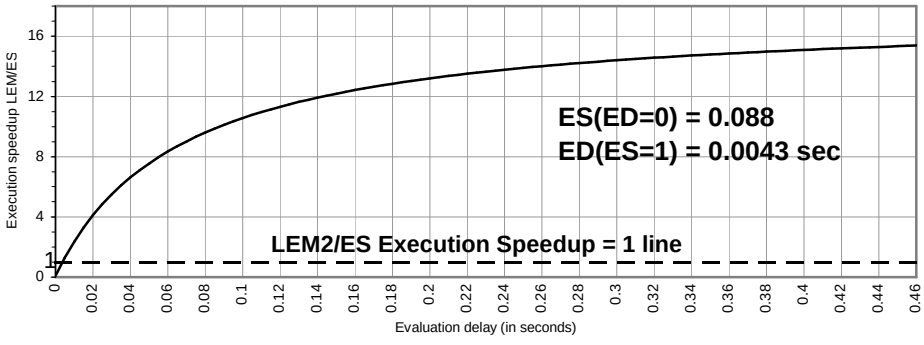


Figure 13. The dependence of the execution speedup of LEM2/ES for the Rastrigin function of 50 variables on the evaluation delay parameter.

Figure 14 shows that for the evaluation delay 0.0043 sec, the *execution speedup* of LEM over ES reached 1.



.

Figure 14. The dependence of the execution speedup of LEM2/ES on the evaluation delay for the Rastrigin function of 100 variables.

For delay = 0, the execution speed-up was about 0.1 (LEM2 was 10 times slower than ES), while for evaluation delay greater than about 0.1 sec, LEM2 was 10 times faster than ES

These above results confirm the hypothesis that the speedup of LEM2/ES increases with the evaluation delay, and converges to the evolutionary speedup.

4.4 Minimization of the Guassian Quartic Function

This experiment concerns the minimization of the Gaussian Quartic function:

$$Gauss(x_1, x_2, .., x_n) = \sum_{i=1}^{n} i x_i^4 + Gauss(0,1)$$

in which the number of variables, *n*, was set to 10, 50 and 100, and variables ranged between –5.12 and 5.12. This is a unimodal function padded with Gaussian noise (Figure 15). Due to the noise, repeating the same input will usually not produce the same value of the fitness function. Algorithms that do not do well on this test function do poorly on noisy data. A total of 500 runs were performed; each run used a different initial population of size 100.

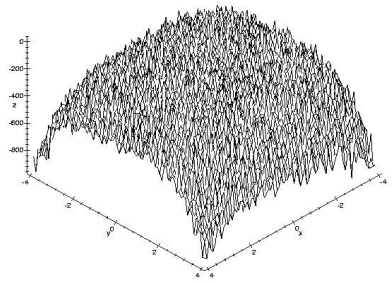


Figure 15. An inverted 2D projection of the Gaussian Quartic function.

Table 5 presents the best results achieved by LEM2 in uniLEM mode and by ES, in terms of the evolution length and the evolution speedup for different δ (0, 0.01, and 0.1). The best solution from LEM2 occurred when the population size was 100, and the HPT and LPT were set to 30%. The best solution for ES occurred when the population size was 100, and the mutation rate was 0.7

Number of Variables	**10**			**50**			**100**		
	δ0	**δ0.01**	**δ0.1**	**δ0**	**δ0.01**	**δ0.1**	**δ0**	**δ0.01**	**δ0.1**
LEM2	700	700	700	4000	4000	3900	15000	10100	10100
ES	3600	3200	2900	40100	40100	36700	220000	188000	175000
Evolution Speedup LEM2/ES	5	5	4	10	10	9	15	19	17

Table 5. Evolution speedups of LEM2 over ES for the Gaussian Quartic function.

Figure 16 shows the increase the evolution length with the number of function variables for ES and LEM2. As in previous problems, the evolution length of ES grows much faster with

the number of variables than that of LEM2 (the target solution was defined by $\delta = 0.1$). This experiment also shows that LEM2 is able to work with noisy functions.

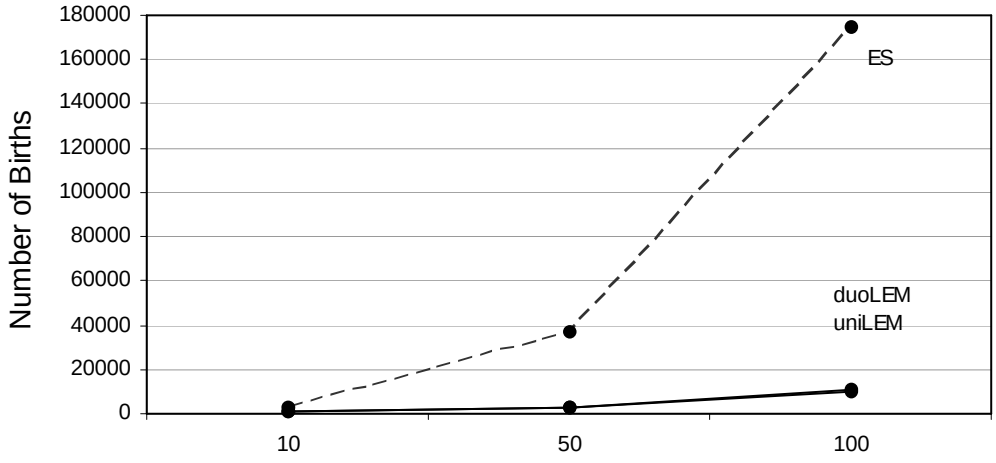


Figure 16. The increase of the evolution length with the number of variables in minimizing the Guassian Quartic Function using ES and LEM2.

Figures 17, 18, and 19 show the dependence of the execution speedup of LEM2/ES on the evaluation delay. In these graphs, when the evaluation delay is equal to 0, the LEM2/ES speedup is below 1 (that is, ES finds the optimum faster despite a longer evolution length). In Figure 19, when the evaluation delay is 2.7 msec, the speedup LEM2/ES reaches 1, and then slowly converges to the evolution speedup of 4.

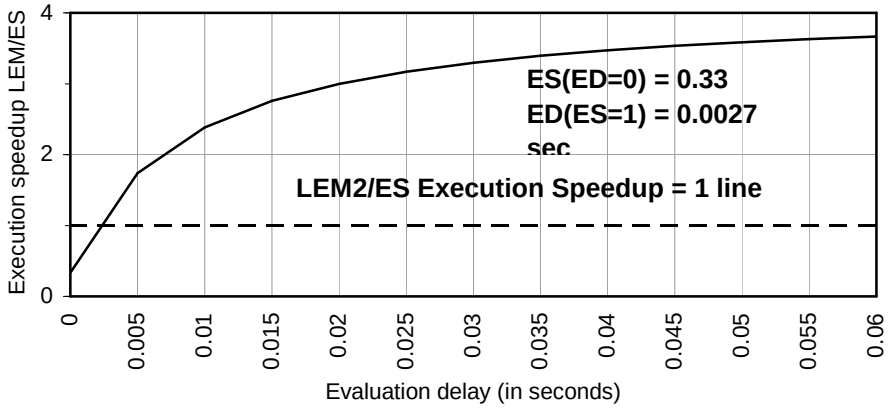


Figure 17. The dependence of the execution speedup of LEM2/ES on the evaluation delay parameter for the Guassian Quartic function of 10 variables.

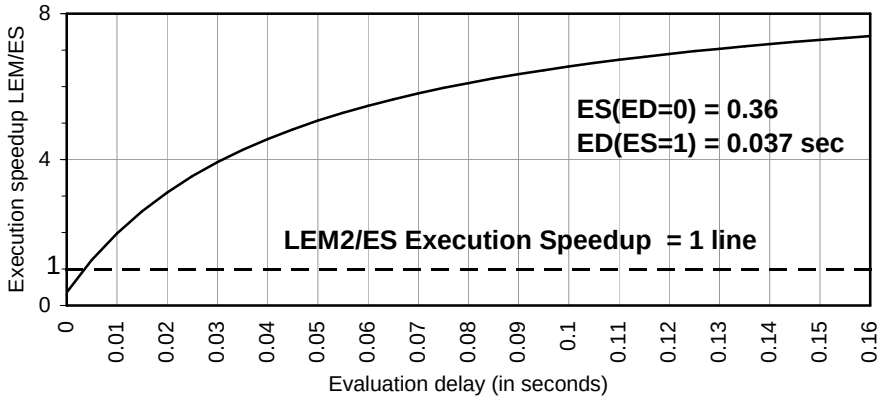


Figure 18. The dependence of the execution speedup of LEM2/ES on the evaluation delay for the Guassian Quartic function of 50 variables.

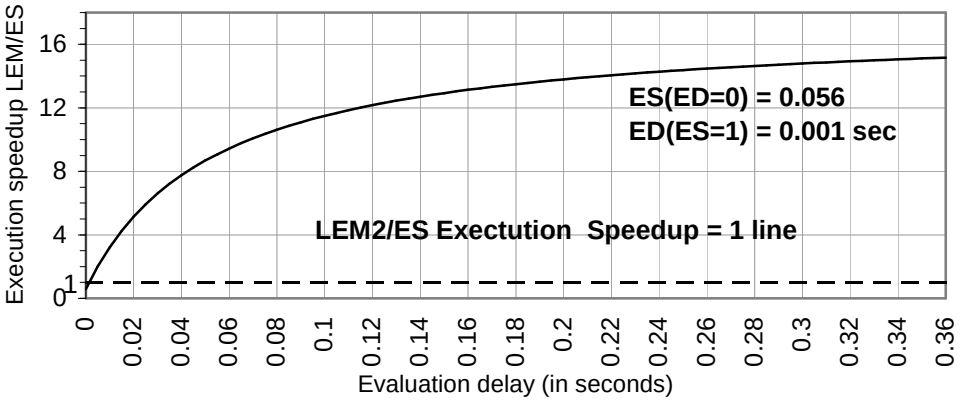


Figure 19. The dependence of the execution speedup of LEM2/ES on the evaluation delay for the Guassian Quartic function of 100 variables.

4.5 Discussion of Function Optimization Results

In all experiments conducted, LEM2 outperformed ES and other tested Darwinian-type algorithms in terms of the evolution length, sometimes by two or more orders of magnitude. In a very few experiments, this advantage did not result in a shorter evolution time, but when a small delay was added to the fitness evaluation computation (about 0.001sec or less), LEM2 outperformed the tested evolutionary computation algorithms also in terms of the execution time. One of the significant results of these experiments was that with an increase in the evaluation delay, the execution speedup converged to the evolution speedup.

The most significant finding from the experiments is that LEM2's evolution speedup over ES increased with the number of function variables, that is, with the complexity of the optimization problem. This would suggest that LEM2 may be particularly well suited for problems in which fitness evaluation is very time-consuming and/or costly. Such a situation

occurred, for example, in heat exchanger design (Kaufman and Michalski, 2000b) in which the fitness evaluation is done through a numerical simulation (see next section).

A reader can experiment with the LEM2 program by downloading it from the website www.mli.gmu.edu/lem.

5 DESIGN OPTIMIZATION: LEMd AND ITS ISHED IMPLEMENTATION

5.1 The LEMd Architecture

Complex design optimization problems appear to be an application domain for which the search capabilities of evolutionary computation algorithms are often very useful. They differ from the function optimization problems described above in one important respect. Specifically, in function optimization problems, the values of any of the variables may be usually set or modified in any fashion (within their defined ranges) without any harm to the integrity of the proposed solution. In design problems, however, variables will typically represent positions or configurations of components, and are subject to various constraints. Thus, arbitrary changes to them will often result in solutions that are either physically or practically infeasible. Even if we may assume that the fitness evaluator will recognize such infeasibility and return an appropriate score, the high density of such designs will likely hamper the evolution process, whether Darwinian or machine learning-based.

This was the motivation for the development of LEMd, a LEM methodology oriented toward design problems. In contrast to LEM, LEMd includes the following features:

- Domain-specific representation of variables. That is, the optimized variables represent real-world aspects of the design configuration, and are subject to domain-specific constraints.
- Instead of generic Darwinian operators, such as random mutations and recombinations, LEMd employs domain-specific *Design Modification (DM)* operators that make various changes in the candidate designs that are potentially useful according to the domain knowledge.
- The algorithms for generating individuals, and instantiation of learned rules employ domain knowledge-based formulas and constraints tailored toward generating feasible designs.

In order to implement LEMd and apply it to a specific design problem, one needs to define a general representation for designs under consideration, define and implement appropriate DM operators, and generate an initial design population. To apply AQ learning in Machine Learning mode, the designs need to be represented as vectors of attribute values, but attributes can be of different types, as specified in the learning program (nominal, structured, ordinal, cyclic, ratio, and/or absolute). The initial design population can be generated by utilizing existing designs, randomly, by an expert employing known design knowledge, or through a combination of these techniques.

LEMd assumes also that the quality of initial and subsequently generated designs can be evaluated in some way, for example, by a design simulator. LEMd employs duoLEM, but the Darwinian Evolutionary mode is done in a special way, because in practical design

applications an expert often has domain knowledge that can be used to define meaningful design variations. As mentioned above, to generate such variations, Design Modification (DM) operators need to be defined based on the expert advice or knowledge. These operators perform design modifications that are tried in the process of evolutionary computation, rather than the random mutations and/or recombinations used in conventional evolutionary computation. Since in real-world problems, various domain constraints are known, LEMd also allows the user to define various constraints and applies them in both Darwinian and Machine Learning modes of operations. Figure 20 illustrates the control flow within LEMd.

In LEMd, individuals represent different designs under consideration. Each design is defined by a vector of attributes that characterizes it. The Control Module takes the current design population and determines which of LEM's evolutionary modes to apply. A new population of candidates is generated: through DM operators in Darwinian Evolutionary mode, and through rule learning and instantiation in Machine Learning mode. The created population is then passed to the simulator for evaluation. The designs and their evaluations are passed to the Control Module for the next generation (iteration).

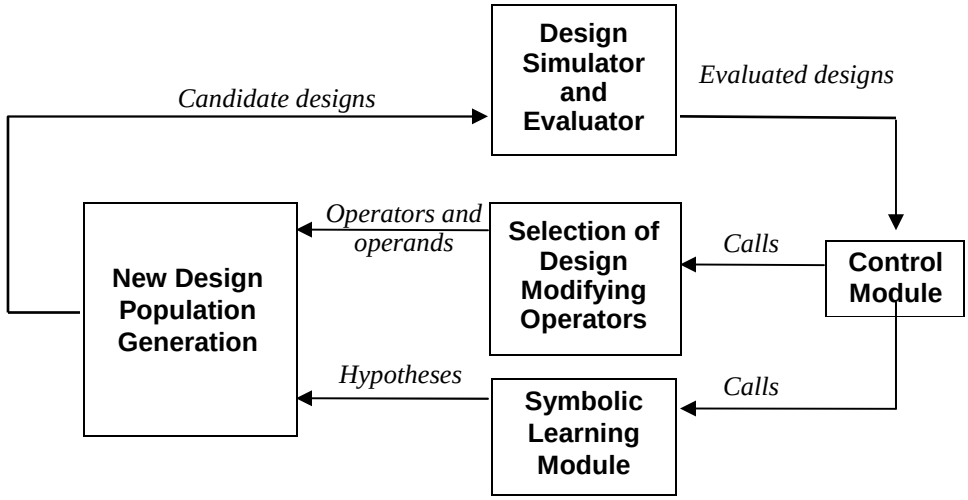


Figure 20. Control flow structure of the LEMd methodology.

Two related parameters guide the Control Module in determining which evolutionary mode to apply. LEMd starts in a default mode, which we assumed to be Darwinian Evolution mode, and continues until the Mode Termination Condition is satisfied, e.g., when the best design in the population has plateaued. It then switches to Machine Learning mode, and continues in it, again until the Mode Termination Condition is satisfied. The two modes alternate until the LEMd termination condition is satisfied (a sufficiently good design has been generated, or allocated computational resources have been exhausted).

5.2 ISHED for Evaporator Design Optimization

To test the LEMd methodology in a real-world application domain, we implemented it in the LEMd-ISHED, or, briefly, the ISHED (Intelligent System for Heat Exchanger Design) program tailored to the problem of optimizing evaporators in air conditioning units under

given environmental and technical constraints. To give the reader a better understanding of the complexity of this problem, a brief explanation follows.

In an air conditioning unit, the refrigerant flows through a loop. It is superheated and placed in contact with cooler outside air (in the condenser unit), where it transfers thermal energy (heat) out and liquifies. Coming back to the evaporator, it comes into contact with the warmer interior air that is being pushed through the evaporator, as a result cooling the air and heating and evaporating the refrigerant. An evaporator consists of arrays of parallel tubes through which the refrigerant flows back and forth. For the purposes of identification and illustration, these tubes are typically numbered from left to right in each row, starting with the first row (as viewed from the direction from which the air flowing over the tubes arrives).

The path the refrigerant takes through these tubes will affect both the temperature of the refrigerant when it reaches a given tube, and the temperature of the air after it passes over the tube. The amount of heat transfer (cooling) the air conditioner will provide is the aggregate of the heat transfer provided by each of its evaporator's tubes. These terms will be a function of the temperature and volume per unit time of both the air and the refrigerant coming into contact at that tube. Different orderings of the tubes will change the characteristics of the refrigerant passing through each tube, and the results of prior air/refrigerant interactions will affect both substances' temperatures at later interactions. Other factors also affect later interactions. For instance, the refrigerant will lose pressure (and velocity) while passing through the bends between tubes; it thus helps in maximizing heat transfer if adjoining tubes are physically close to each other. By changing the path of the refrigerant flow, one can also therefore change the amount of heat transferred between air and refrigerant. The more that can be transferred overall, the more efficiently the interior air will be cooled to the desired temperature.

The optimization problem involves determining an arrangement of tubes that produces the highest evaporator capacity under given technical and environmental constraints. Because of the nature of the problem and the feasible ways of internally representing evaporator structures, both evolutionary modules utilize problem-specific customization. Traditional genetic operators, random mutations and crossovers, would be unworkable in this domain; therefore, we implemented eight domain-specific design modifying (DM) operators based on discussions with a domain expert. The DM operators change the characteristics of candidate evaporators in ways likely to lead to *admissible* new structures, that is, structures satisfying the given constraints. A selected operator is tried repeatedly with different operands in order to generate a feasible design, until it either succeeds or "times out" (based on control parameters specifying the allowed number of iterations), in which case another operator, hopefully more applicable, will be tried.

For example, one operator may create a split in a refrigerant path by moving the source of a tube's refrigerant closer to the inlet tube (Figure 21), a second operator may swap the tubes in the structure (Figure 22), another operator may graft a path of tubes into another path (Figure 23), etc. In these figures, solid arrowed lines represent the initial connections, with dashed ones representing new ones created by the DM operator. Each tube has only one source, so the dashed links replace links from their destination tubes' prior sources. An arrow that is open-ended on one side represents an inlet or outlet flow (flow into or out of the evaporator

from/to other parts of the refrigerant circuitry). The application of these operators is domain knowledge driven, that is, operators are applied according to known technical constraints.

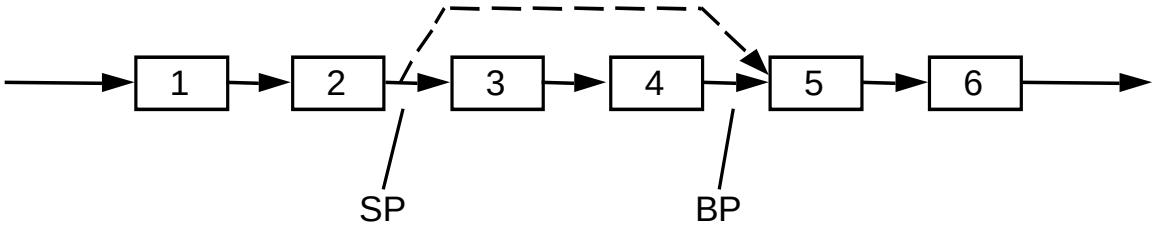


Figure 21. Application of the SPLIT Operator, SPLIT(2,5).

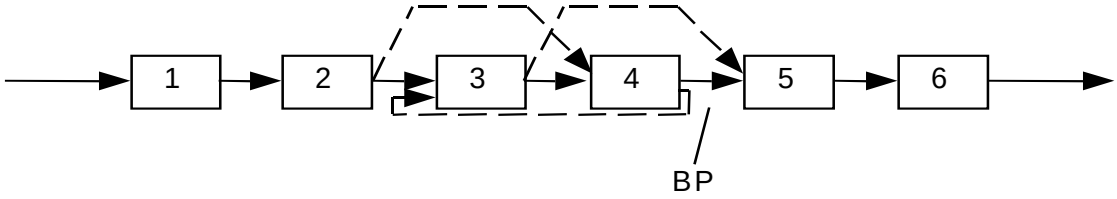


Figure 22. Application of the SWAP Operator, SWAP(5).

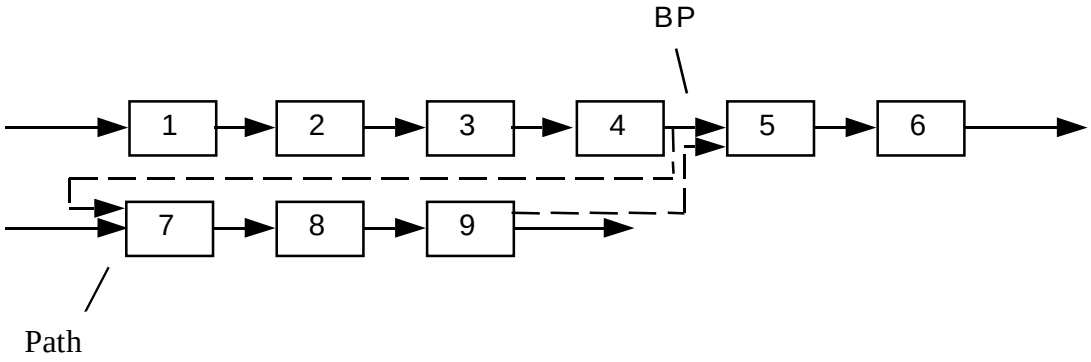


Figure 23. Application of the INSERT Operator, INSERT(7,5).

The second strategy, based on symbolic learning, examines the characteristics of both well- and poorly-performing designs, and automatically creates hypotheses (in the form of attributional rules) that characterize the well-performing architectures. These hypotheses are then applied to generate a new population of designs.

Machine learning mode in ISHED is also tailored to the evaporator design task. The hypotheses generated describe abstractions of the individual structures. Specifically, they specify only the location of inlet, outlet and split tubes. Beyond that, the instantiation module may choose among the different designs that fit the AQ-generated template, and generate the most plausible one according to the real-world background knowledge. These programs use high and low fitness thresholds of 25% to select the H-group and L-group. Once rules are generated, an elitist strategy is used to form the next generation of proposed

architectures. The best architecture found so far, as well as all members of the H-group are passed to the next generation, along with various instantiations of the learned rules.

An ISHED run proceeds as follows. Given instructions characterizing the environment for the target evaporator, an initial population of designs (specified by the user and/or randomly generated), and parameters for the evolutionary process, ISHED evolves populations of designs using a combination of Darwinian and machine learning operators for a specified number of generations. The programs return a report that includes the best designs found and their estimated quality (capacity). Throughout the execution, design capacities are determined by a heat exchanger simulator (Domanski, 1989).

6 EXPERIMENTS ON EVAPORATOR OPTIMIZATION

Experiments with ISHED were performed under different sets of conditions, such as different refrigerants, evaporator sizes and shapes, and airflow patterns. Industrially available air conditioning systems typically perform very efficiently as long as the airflow is fairly uniform. However, their efficiency drops off sharply if that is not the case; the side of the unit over which more air flows has a heavier cooling burden, so for best performance it needs to carry more and colder refrigerant. Manufacturers generally have not been building models for such a situation.

An example of the output from an ISHED run is shown in Figure 24. This run was done in a verbose mode, and as such, a full log details every design tested, every operator applied, and every rule learned. The figure only shows a very small sample of the full output in order to give the reader a flavor of ISHED's optimization process. Numbers in parentheses on the right hand side are pointers to the explanations in the text below. In addition to these explanations, we have added some comments to the log itself (in italics).

The structure of an evaporator is represented as a vector of integer values. Each value indicates the number of the tube that is the source of each tube's refrigerant (tubes are numbered left-to-right, starting with the first row), or an 'I' is displayed to indicate that a tube is an inlet tube. The simulator evaluates the structure represented by this vector, and returns a 'capacity'. This value is an indication of the cooling capacity of the given structure according to the simulator model.

The first part of the log, marked by **(1)**, provides the user with a summary of the parameters under which the program ran. Here we see that ISHED was creating designs of heat exchangers consisting of 3 rows of 16 tubes. It evolved a population of 15 individuals over 40 generations. The Operator Persistence parameter instructs the program to attempt a Darwinian design modifying operator 5 times before giving up on it and instead choosing another operator. The Mode Persistence parameters instruct it to shift from Darwinian to Symbolic (Machine Learning) mode after two consecutive generations fail to provide improvement in the population, and to shift back to Darwinian mode after one symbolic generation does not provide improvement.

The log then shows the initial population (generated randomly in this case). At **(2),** two individuals are shown, specifically, the third and eighth member of the population, whose capacities are 5.5376 and 5.2009, respectively. From this population, the seeds for the next generation, with the exception of the first one, are selected probabilistically, based upon the

capacities of the individuals of the current population. The first individual is selected elitistically – the best individual observed so far. The log shows **(3)** that the fifteen members of the new (Generation 1) population will be built from individuals 0.3, 0.2, 0.3, 0.7, 0.9, 0.3, 0.9, respectively.

Exchanger Size: 16 x 3

Population Size: 15 Generations: 40

Operator Persistence: 5

Mode Persistence: GA-probe=2 SL-probe=1 **(1)**

Initial population:

Structure #0.3: 17 1 2 3 4 5 6 7 8 9 12 13 29 15 31 I 18 33 20 36 22 38 24 40 26 42 11 2 7 45 14 47 16 34 35 19 37 21 39 23 41 25 43 44 28 46 30 48 32: Capacity = 5.5376

Structure #0.8: 17 1 20 3 4 22 6 24 8 26 10 28 27 15 16 32 33 2 18 19 5 38 7 40 9 42 11 44 13 46 30 48 34 35 36 I 21 37 23 39 25 41 27 43 29 45 31 47: Capacity = 5.2099 **(2)**

and 13 others

Selected Members: 3, 2, 3, 7, 9, 3, 9, ... **(3)**

Operations: NS(23, 39), SWAP(8), SWAP(28), ..., SWAP(29), SWAP(25), SWAP(1) **(4)**

Below is one of the structures created by the application of a SM operator in Darwinian mode (by swapping the two tubes following tube 29 in Structure #0.8)

Generation 1:

Structure #1.13: 17 1 20 3 4 22 6 24 8 26 10 28 27 15 16 32 33 2 18 19 5 38 7 40 9 42 11 4 13 45 30 48 34 35 36 I 21 37 23 39 25 41 27 43 46 29 31 47: Capacity=5.2093 **(5)**

and 14 others.

Selected Members: 6, 15, 11, 3, 13, 1, ...

. . . . . .

The program soon shifts into Machine Learning Mode:

Generation 5: Learning mode

Learned rule:
[x1.x2.x3.x4.x5.x6.x7.x8.x9.x11.x12.x13.x14.x15.x17.x18.x19.x20.x21.x22.x23.x24.x25.x26.x27.x28.x29.x30.x31.x32.x33.x34.x35.
x36.x37.x38.x39.x40.x41.x42.x43.x44.x45.x46.x47.x48=regular] & [x10=outlet]&[x16=inlet] (t:7,u:7,q:1) **(6)**

An example of a generated structure:

Structure #5.1: 17 1 2 3 4 5 6 7 8 9 12 29 45 30 31 I 18 33 20 36 22 38 24 40 26 42 11 27 13 15 47 48 34 35 19 37 21 39 23 41 25 43 44 28 46 14 32 16: Capacity=5.5377 **(7)**

. . . . . .

Below is a structure from the 21st generation:

Generation 21: Learning mode

Structure #21.15 2 18 4 1 6 3 5 7 8 9 12 13 45 15 31 I 33 17 35 36 22 39 24 40 42 25 11 44 30 46 32 47 34 19 20 37 21 23 38 41 26 43 28 27 29 14 48 16: 5.5387 **(8)**

and 14 others

Selected Members: 11, 4, 4, 13, 15, 10, 12, 13, 15, 15, 12, 2, 3, 5, 10.

. . . . . .

ISHED1 continues to evolve structures, and finally achieves:

Generation 40:

Structure #40.15: 33 17 2 41 4 5 6 9 7 8 12 29 46 45 47 I 1 34 20 36 22 38 24 3 42 43 44 27 13 15 32 16 18 11 19 37 21 32 23 25 40 26 28 35 30 14 48 31: Capacity=6.3686 **(9)**

Figure 24. An excerpt from the log of an ISHED1 run.

Each seed for the new population then has a structure modifying operator applied to it **(4)** as follows: Individual 1.1 is created by applying operator NS(23,39) to individual 0.3 (change the source of refrigerant for tube 23 from whatever it was to tube 39); individual 1.2 is created by applying operator SWAP(8) to individual 0.2 (swap the positions of the two tubes

preceding tube 8); etc. It is shown **(5)** that individual 1.13, generated by applying operator SWAP(29) to design 0.8, has a capacity of 5.2093.

After progress has stalled, ISHED1 switches to Machine Learning mode, and discovers a rule **(6)** that indicates a pattern in which high-performing designs consist of an outlet at position 10, an inlet at position 16, and interior tubes at all other positions. The learned rules are instantiated to become members of the new population, such as Structure 5.1 **(7)**.

The experiment continues in this manner, and at its halfway point, true progress in evolving better designs has not emerged **(8)**. But by the run's end **(9)**, there has been a significant leap in design quality. While the initial population was producing designs with capacities no better than 5.5376, ISHED1 has in this short run produced a design with a capacity of 6.3686.

During the course of ISHED development, many experiments with the system were conducted. The initial experiments concentrated on a well-known problem, using a common heat exchanger size and a fairly uniform airflow pattern. ISHED designs provided results comparable to the industry standard. One concern in some of the ISHED-generated designs was that after many generations of Darwinian evolution, designs would become chaotic in terms of their inter-tube connections (and the simulator wasn't fully reflecting the detrimental effect of this). Nonetheless, using available tools, an engineer could easily smooth some of the connections, hopefully at little or no cost to the estimated capacity of the exchanger.

In later experiments, the refrigerant was changed, and the airflow pattern was defined as highly non-uniform. Under such conditions, industry-standard heat exchangers do not perform well. The best ISHED-produced architectures conformed intuitively to expectations of what a successful architecture in a non-uniform airflow should look like, and indeed performed far better than the currently-used expert-designed structures.

Subsequent experiments varied the size and shape of the heat exchanger -- between 2 and 4 depth rows, with between 40 and 90 total tubes. Similar results were observed. Later, we began experimenting with pre-specified members of initial populations. The results were to some degree mixed. When a very large portion of the initial population was pre-specified with known high-quality designs, further improvement could often be found. To some degree, the pre-specification is analogous to an initial symbolic learning step using the prior background knowledge; as a result, ISHED begins with a solid population.

But when fewer individuals were used to seed the initial population, improvement was hard to come by. While further experimentation is needed to determine if this is a regular occurrence, and if so, its cause, it is possible that a level of imbalance is reached in the population that hinders both the establishment of large numbers of seeded examples and their kin for improvement, and the blossoming of promising, but relatively weak, randomly generated individuals. It is also possible that system parameters need to be adjusted for such a scenario.

The experiments during all stages of this work served to confirm the ability of ISHED to generate improved designs. There appears to be promise that the LEMd methodology would exhibit similar success in other design tasks.

7 RELATED WORK

The LEM methodology represents a new approach to evolutionary computation. Its relation to other evolutionary computation methods was briefly described in (Michalski, 2000). Unlike most methods of evolutionary computation that draw inspiration from Darwinian evolution, LEM attempts to model what we call an *intellectual evolution*. In intellectual evolution, which governs the development of human artifacts, the generation of new populations is based on the results of the analysis of the advantages and disadvantages of past populations.

The closest methods to LEM appear to be cultural evolution algorithms, as they utilize top performing individuals and use generalized beliefs in the evolutionary process (e.g., Reynolds, 1994; Rychtyckyj and Reynolds, 1999; Saleem and Reynolds, 2000). The approach taken in cultural algorithms differs, however, significantly from LEM. Unlike LEM, cultural evolution is a process of dual inheritance consisting of a "micro-evolutionary level," which involves individuals described by traits and modified by conventional evolutionary operators, and a "macro-evolutionary" level, in which individuals generate "mappa" representing generalized beliefs that are used to modify the performance of individuals in the population. LEM is different from cultural evolution algorithms in both, the way learning process is implemented and in the way its results are used in the process of evolution.

Sebag and Schoenauer (1994) applied AQ-type learning to adaptively control the crossover operation in genetic algorithms. In their system, the rules are used for the selection of the crossover operator. Sebag, Schoneauer and Ravise (1997a) used inductive learning for determining mutation step-size in evolutionary parameter optimization. Ravise and Sabag (1996) described a method for using rules to prevent new generations from repeating past errors. In a follow-up work, Sebag, Schoenauer and Ravise (1997) proposed keeping track of past evolution failures by using templates of unfit individuals, called “virtual losers.” An evolution operator, which they call “flee-mutation,” aims at creating individuals different from the virtual losers.

Grefenstette (1991) developed a genetic learning system, SAMUEL, that implements a form of Lamarckian evolution. The system was designed for sequential decision making in a multi-agent environment. A strategy, in the form of *if-then* control rules, is applied to a given world state and certain actions are performed. This strategy is then modified either directly, based on the interaction with the environment, or indirectly by changing the rules’ strength within the strategy. The changes in a strategy are passed to its offspring. This is a Lamarckian-type process that takes into consideration the performance of a single individual when evolving new individuals.

Another approach that extends the traditional Darwinian approach can be found in the GADO algorithm (Rasheed, 1998). GADO is an evolutionary algorithm developed for complex engineering problem optimization. It differs from traditional genetic algorithms primarily in the way new individuals are generated. It uses five different crossover operators, three of which are introduced in this algorithm: Line crossover, double line crossover, and guided crossover. However, unlike LEM, the algorithm does not create any generalizations of the current population, and therefore is significantly different.

8 CONCLUSION

The results presented in this paper have confirmed previous preliminary findings that LEM offers a powerful new methodology for conducting evolutionary computation in a non-Darwinian fashion. Two implementations of LEM has been tested on selected and evaluated, LEM2 – on function optimization problems and ISHED – on an engineering design problem concerning optimization of tube arrangements in evaporators in air conditioners. In all function optimization experiments, LEM2 outperformed selected Darwinian-type evolutionary algorithms (mostly ES) in terms of the evolution length, sometimes achieving speedups of two or more orders of magnitude. In the evaporator design domain, ISHED was able to find solutions that were better or comparable to the best designs used in the industry.

Since operators of hypothesis generation and instantiation employed in LEM2 are significantly more computationally costly than operators of mutation and recombination used in ES, the evolution speedup does not always result in the execution speedup. It was shown experimentally that if the fitness evaluation time is above a small threshold (tens of milliseconds), LEM2 also outperforms ES in terms of the execution time. The execution time speedup grows with fitness evaluation time, asymptotically converging to the evolution speedup. The most remarkable result of experiments is that the evolutionary speedup advantage of LEM2 over ES (evolutionary strategy) grew rapidly with the complexity of the optimized function. It is likely that a similar advantage would be obtained with regard to other Darwinian-type evolutionary computation algorithms.

Experiments have revealed also a weakness in the LEM2 implementation that sometimes appears in the last steps of the evolutionary processes. When almost all the variables reach their global optimum value, the program may take a long time to find the absolute optimal value for the few remaining variables. This problem is currently handled by employing the *Start-Over* operator. Further research is needed to determine a better method for handling this problem.

The LEM methodology is at a very early stage of development, and poses many interesting new research problems. They include a theoretical and experimental investigation of the trade-offs inherent in LEM, an implementation of more advanced versions of LEM, experimentation with different combinations of conventional evolutionary algorithms and machine learning algorithms, and testing the methodology in different application domains. Among important research topics are also the development of methods for applying LEM to dynamic landscapes, and to optimization problems with complex constraints.

Concluding, the experiments described here have strongly confirmed the earlier results that LEM can very significantly reduce the length of evolutionary computation over Darwinian-type algorithms, and thus can be particularly useful in domains where the fitness evaluation is time-consuming or costly. They also indicated a pattern, potentially highly significant for practical applications, in which the LEM advantage over Darwinian-type evolutionary computation appears to increase with the complexity of the optimization problem.

REFERENCES

Baeck, T., Fogel, D.B. and Michalewicz, Z. (eds.), *Handbook of Evolutionary Computation*, Oxford: Oxford University Press, 1997.

Cervone, G., "An Experimental Application of the Learnable Evolution Model to Selected Optimization Problems," *Reports of the Machine Learning and Inference Laboratory*, MLI 99-12, George Mason University, Fairfax, VA, 1999.

Cervone, G, Kaufman, K.A., Michalski, R.S., "Experimental Validations of the Learnable Evolution Model," *Proceedings of the 2000 Congress on Evolutionary Computation*, La Jolla, CA, 2000.

Cohen, P. R., and Feigenbaum, E.A., *The Handbook of Artificial Intelligence*, Los Angeles: William Kaufmann Inc., 1982.

Coletti, M., Lash, T., Mandsager, C., Michalski, R.S., and Moustafa, R., "Comparing Performance of the Learnable Evolution Model and Genetic Algorithms on Problems in Digital Signal Filter Design," *Proceedings of the 1999 Genetic and Evolutionary Computation Conference*, 1999.

Domanski, P.A., "EVSIM-An Evaporator Simulation Model Accounting for Refrigerant and One Dimensional Air Distribution," NISTIR 89-4133, 1989.

Grefenstette, J., "Lamarckian Learning in Multi-agent Environmen," in Belew, R. and Booker, L. (eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA: Morgan Kaufmann, pp. 303-310, 1991.

Kaufman, K.A. and Michalski, R.S., "The AQ18 System for Machine Learning and Data Mining: User's Guide," *Reports of the Machine Learning Laboratory*, MLI 00-3, George Mason University, Fairfax, VA, 2000a.

Kaufman, K.A. and Michalski, R.S., "Applying Learnable Evolution Model to Heat Exchanger Design," *Proceedings of the Twelfth International Conference on Innovative Applications of Artificial Intelligence*, Austin, TX, 2000b.

Kerber, R., "ChiMerge: Discretization of Numeric Attribute," *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, CA, pp. 123-127, 1992.

Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer Verlag, Third edition, 1996.

Michalski, R.S., "AQVAL/1--Computer Implementation of a Variable-Valued Logic System VL1 and Examples of its Application to Pattern Recognition," *Proceedings of the First International Joint Conference on Pattern Recognition*, Washington, DC, pp. 3-17, 1973.

Michalski, R.S., "A Theory and Methodology of Inductive Learning," *MACHINE LEARNING: An Artificial Intelligence Approach*, Morgan Kaufmann, 1983.

Michalski, R.S., "Learnable Evolution: Combining Symbolic and Evolutionary Learning," *Proceedings of the Fourth International Workshop on Multistrategy Learning*, Desenzano del Garda, Italy, pp.14-20, 1998.

Michalski, R.S., "LEARNABLE EVOLUTION MODEL: Evolutionary Processes Guided by Machine Learning," *Machine Learning* 38, pp. 9-40, 2000.

Michalski, R.S., "ATTRIBUTIONAL CALCULUS: A Logic and Representation Language for Natural Induction," *Reports of the Machine Learning and Inference Laboratory*, George Mason University, Fairfax, VA, 2003 (to appear).

Michalski. R.S. and Cervone, G., "Adaptive Anchoring Discretization of Continuous Variables for the Learnable Evolution Model," *Reports of the Machine Learning and Inference Laboratory*, MLI 01-5, George Mason University, Fairfax, VA, 2001.

Michalski. R.S. and Zhang, Q., "Initial Experiments with the LEM1 Learnable Evolution Model: An Application to Function Optimization and Evolvable Hardware," *Reports of the Machine Learning and Inference Laboratory*, MLI 99-4, George Mason University, Fairfax, VA, 1999.

Muhlenbein, H., Schomisch, M. and Born, J., "The Parallel Genetic Algorithm as Function Optimizer," *Proceedings of the Fourth Int'l Conference on Genetic Algorithms and their Applications*, 1991.

Rasheed, K., "GADO: A Genetic Algorithm for Continuous Design Optimization," *Technical Report DCS-TR-352*, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1998. Ph.D. Thesis, 1998.

Ravise, C. and Sebag, M, "An Advanced Evolution Should Not Repeat Its Past Errors," *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 400-408, 1996.

Reynolds, R.G., "An Introduction to Cultural Algorithms," *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, 1994.

Rosenbrock, H.H., "An automatic method for finding the greatest or least value of a function," *Computer Journal* 3:175, 1960.

Rychtyckyj, N. and Reynolds, R.G., "Using Cultural Algorithms to Improve Performance in Semantic Networks," in Michalewicz, Z., Schoenauer, M.. Yao, X., and Zazala, A. (eds.), *Proceedings of the Congress on Evolutionary Computation*, Washington, DC, pp. 1651-1663, 1999.

Saleem S. and Reynolds, R.G., "Cultural Algorithms in Dynamic Environments," *Congress on Evolutionary Computation 2000 (CSC00), vol. 2*, La Jolla, CA, pp.1513-1520, 2000.

Sebag, M. and Schoenauer, M., "Controlling Crossover Through Inductive Learning," *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, Springer-Verlag, LNVS 866, pp. 209-218, 1994.

Sebag, M., Schoenauer M., and Ravise C., “Inductive Learning of Multation Step-size in Evolutionary Paramter Optimization,” *Proceedings of the 6th Annual Conference on Evolutionary Programming*, LNCS 1213, Indianapolis, pp. 247-261, 1997.

Sebag, M., Shoenauer, M., and Ravise, C., “Toward Civilized Evolution: Developing Inhibitions,” *Proceedings of the 7th International Conference on Genetic Algorithms*, pp.291-298, 1997.

Shekel, J., “Test Functions for Multimodal Search Techniques,” in Schlierkamp-Voosen, D. and Muhlenbein, H. (eds.), *Strategy Adaptation by Competing Subpopulations, Parallel Problem Solving from Nature, Proceedings of the Third Workshop, PPSN III*, 1994.

Wnek, J., Kaufman, K., Bloedorn, E. and Michalski, R. S., “Inductive Learning System AQ15c: The Method and User’s Guide,” *Reports of the Machine Learning and Inference Laboratory*, MLI 95-4, George Mason University, Fairfax, VA, 1995.

A publication of the *Machine Learning and Inference Laboratory*
School of Computational Sciences
George Mason University
Fairfax, VA 22030-4444 U.S.A.
http://www.mli.gmu.edu

Editor: R. S. Michalski
Assistant Editor: K. A. Kaufman

The *Machine Learning and Inference (MLI) Laboratory Reports* are an official publication of the Machine Learning and Inference Laboratory, which has been published continuously since 1971 by R.S. Michalski's research group (until 1987, while the group was at the University of Illinois, they were called ISG (Intelligent Systems Group) Reports, or were part of the Department of Computer Science Reports).