The Simplest Evolution/Learning Hybrid: LEM with KNN

Guleng Sheri, David W. Corne

Abstract— **The Learnable Evolution Model (LEM) was introduced by Michalski in 2000, and involves interleaved bouts of evolution and learning. Here we investigate LEM in (we think) its simplest form, using *k*-nearest neighbour as the 'learning' mechanism. The essence of the hybridisation is that candidate children are filtered, before evaluation, based on predictions from the learning mechanism (which learns based on previous populations). We test the resulting 'KNNGA' on the same set of problems that were used in the original LEM paper. We find that KNNGA provides very significant advantages in both solution speed and quality over the unadorned GA. This is in keeping with the original LEM paper's results, in which the learning mechanism was AQ and the evolution/learning interface was more sophisticated. It is surprising and interesting to see such beneficial improvement in the GA after such a simple learning-based intervention. Since the only application-specific demand of KNN is a suitable distance measure (in that way it is more generally applicable than many other learning mechanisms), LEM methods using KNN are clearly recommended to explore for large-scale optimization tasks in which savings in evaluation time are necessary.**

I. Introduction

MICHALSKI introduced the Learnable Evolution Model (LEM) [10] in 2000. LEM is a highly generalised hybrid approach to optimisation, in which the overall idea is to run repeated stretches of evolution and learning in series, where the next 'evolution' stretch is informed by the previous 'learning' stretch, which in turn learned about the mapping between genotype and fitness from previous populations. For example, we may start by running an evolutionary algorithm for 10 generations; then we halt the evolutionary algorithm and run a learning method. This learning method might be a neural network, for example, which will try to learn to predict fitness from vectors of gene values. Or, it may be a decision tree learner which tries to classify chromosomes into five categories of 'goodness' and so forth. The result of the learning phase is then fed into the next stretch of evolution. The way the learning influences the evolution is not restricted by the LEM framework. For example (and as done in this paper), the learned model could be used to predict the fitness (or fitness category) of children before they are evaluated, and the evolution phase discards, without evaluation, children that are predicted to be particularly unfit. Alternatively, the learned model may be used to constrain the genetic operators in such a way that children are more likely to be fit. Or, the learned model may be used to 'repair' children that are generated in the normal way. And so on. Evolution then continues for another few generations, then more learning, and so it continues.

Guleng Sheri is with the Department of Computer Science, Heriot-Watt University, Edinburgh, Scotland, UK (phone: +44 (0)131-451-8428; fax: +44 (0)131-451-3327; email: gls3@macs.hw.ac.uk). David Corne is with the Department of Computer Science, Heriot-Watt University, Edinburgh, Scotland, UK (phone: +44 (0)131-451-3410; fax: +44 (0)131-451-3327; email: dwcorne@macs.hw.ac.uk).

In [10], the learning method employed was an AQ learner, specifically AQ15 [6], [13], and the results of this were highly compelling, with very significant improvement over the underlying GA on a suite of five test problems. In particular LEM led to dramatic speedup. LEM-based work subsequent to this includes a multiobjective form (using C4.5 as the learner), found to significantly speed up and improve solution quality for large-scale problems in water distribution networks [5], while the team that developed LEM have updated the framework [14] and continued to gain impressive results [15].

Meanwhile, of course, while LEM was initially published only in the machine learning community, at around the same time Estimation of Distribution Algorithms (EDAs) started to shoot to prominence in the evolutionary computation community [7]. EDAs can also be viewed as learning/evolution hybrids, with the emphasis on building and maintaining models of fit chromosomes. Both techniques (LEM and EDA) now have several published variants (particularly EDA variants), and it is interesting to consider what are the (if any) definitive differences. It seems correct to suggest that while EDAs focus on modelling as the key force behind search activity (i.e. search is guided closely by the modelling, with new sample points in the space generated directly from the model), in LEM the evolutionary component is most responsible for the search (i.e. new points are sampled mainly in the familiar way by using genetic operators on a population of chromosomes), with guidance from learning processes. Most interestingly, recent results from the LEM team compare EDAs and LEM3 directly [15]. They report using various EDA implementations from [1], with best results (of these) on the Rosenbrook and Griewank functions found by EMNA_$global$ [8]. Comparison of LEM3 (with AQ) and EMNA_$global$ on these functions showed LEM3 between 15 and 230 times faster in achieving its best value, which in turn was always better than that achieved by EMNA_$global$.

Finally it must be pointed out that *hybrids* of EDA and GAs, or of EDAs and other search methods, have started to appear since (at least) 2003 [17], [11], [18]. When contrasting the LEM framework with the EDA framework, it is perhaps clearest to say that LEM is similar in style to a hybrid EDA/GA, and this seems to be reflected in the relative success that has so far been shown for EDA/GA hybrids.

The design and application of LEM is clearly worth considerably more research. Our own interest was sparked by the promise shown in [10] for considerable speedup, leading to our investigation of a LEM variant on large scale water distribution network problems [5]. In such, and many, many other problems in which fitness function evaluation takes considerable time, time savings are precious, and can easily make the difference between the problem being solvable at all or not. Following the success of that work, we decided to investigate the design of LEM more closely. The current paper is the result of starting this expedition, in which we evaluate the performance of what we argue is the simplest possible version of LEM; that is, the original LEM framework, using *k*-nearest-neighbour (the simplest possible learning scheme) as the learning mechanism, and employing learning in the evolution mode only by using the current KNN model to predict whether a new child should be evaluated.

In the remainder we continue as follows. Section II provides more detail on the original LEM and on our 'KNNGA', which we also sometimes denote as LEM(KNN). Section **??** provides a pictorial view of how LEM(KNN) works, which may be useful. Section IV covers experiments and results on the test problems that were employed in the original LEM paper. We conclude and discuss in section 6.

II. LEM(AQ) and LEM(KNN)

A. A brief exposition of the original LEM algorithm

We start by explaining how LEM(AQ) works, as described in [10]. First, the initial population is generated and evaluated. It is then divided into high-performance (H-group) and low-performance (L-group) groups according to the initial individuals' fitness values. These two groups are then used as the positive and negative training examples for the AQ learning algorithm. The outcome of the AQ learning algorithm is a set of rules expressing inductive hypotheses (in terms of intervals of gene values) for the positive and negative examples. LEM(AQ) then proceeds with an otherwise normal evolutionary algorithm, except that the operators are designed so that new individuals are generated only with gene values within the ranges of values sanctioned by the recently learned inductive hypotheses. LEM(AQ) then continues for a specified amount of generations, and then pauses for more learning based on the current population. This in turn feeds into the next stage of evolution, and so on. There are additional complications and sophistication in LEM(AQ) that mediate the transitions between learning and evolution, and we refer readers to [10] for fuller details.

B. LEM(KNN) – KNNGA

There is a big difference between LEM(AQ) and our LEM(KNN) in how the learning influences the evolution, which is quite simplified in LEM(KNN). In LEM(AQ), the generation of new individuals are instantiating of the description (set of rules) of the H-group or L-group. However, in LEM(KNN), new individuals are still generated by the common GA mutation and crossover operators, KNN is applied as a particular form of survival selection operator which judges an individual according to the fitness values of its neighbours. A detailed description of our LEM(KNN) algorithm is given below, in which we assume a maximization problem is being considered.

As with LEM(AQ), LEM(KNN) divides the population into high-performance (H-group) and low-performance (L-group) groups according to their fitness values and a given *threshold* (here, 30% – that is, the fittest 30% form the H-group and the worst 30% form the L-group). This is then saved as the *learning population*. Individuals of the H-group and L-group in the *learning population* form the training examples used by the KNN algorithm. Effectively, the 'learning' here corresponds entirely to the process of classification into these groups based entirely on fitness, and hence is one of the simplest learning schemes conceivable. However, this goes hand in hand with the use of the *learning population* in predicting the quality of newly generated individuals, which goes as follows.

The common mutation and crossover operators are used to generate new individuals in the normal way. Once a new individual is generated, KNN is used to predict if this individual is 'good' or 'bad'. First, we find the *k* nearest neighbors for this new individual; if the majority of these are in the H-group, then this individual is predicted as 'good', otherwise this individual is predicted as 'bad'. The 'good' individuals are retained to form the new population for the next generation. The 'bad' individuals are discarded. This continues until sufficient new individuals are generated in (or, predicted to be in) the H-group to form a new population. When a fixed number of generations, we indicate this as *learning gap* (LG), are generated, the *learning population* is updated by the current generation. Again, the *learning population* is classified into the H-group and L-group. This is repeated until a termination condition is reached.

Now we try to ensure a replicable explication with pseudo-code. 'Overview' pseudo-code for LEM(KNN) is as follows:

1) Parameters: Set values for *population size*, parameters for mutation (mutation value, mutation rate), parameters for crossover (parents number, children number) and set elite-preserve operator option. Set *k* (indicating the number of neighbours in KNN algorithm), *learning gap* (indicating the interval before one learning population is updated by another) and the *threshold*.
2) Generate initial Population: Choose a method to create the initial population with *population size* and evaluate this population.
3) Derive extrema: Copy the *current population* as the *learning population* from which create the high fitness group (H-group) and low fitness group (L-group), according to fitness values and *threshold*. These two groups could have a joint set, or their union could be a subset of the whole population set or even equals to the whole population set. These two groups are stored for KNN algorithm.
4) Generate new generations: After reproducing the *current population*, apply the mutation, crossover operators to generate new individuals. Once a new offspring

is generated, (it is not evaluated and is not placed in the mating pool immediately) KNN is applied to find its k nearest neighbors w.r.t H-group and L-group (not the whole *learning population*). For these k nearest neighbors of this offspring, KNNGA judges the majority according to their fitness values, there will be two cases:

i) if the majority is high (that is, most of this offspring's k neighbours are members of H-group), then it is evaluated and retained into the newly created population.

ii) if the majority is low (that is, most of this offspring's k neighbours are members of L-group), then it is aborted.

The generating procedure continues until this new population is filled with such newly generated individuals nearer to H-group. This finishes the generation of one generation.

5) Update H-group and L-group: When *learning gap* is reached, the *learning population* is replaced by the *current population*. The H-group and L-group are therefore recalculated according to the current *learning population* and the same *threshold*. The new H-group and L-group are stored for KNN.

6) Termination condition: The above steps 4),5) repeat until some termination conditions are satisfied:

i) the optimal (if known) is reached; or

ii) the maximum allowed generations is reached; or

iii) the best fitness value has not been improving for a certain number of generations.

The pseudo-code for our specific instantiation of LEM(KNN), which we call KNNGA, is set out here as Algorithm 1.

Algorithm 1 pseudo-code for KNNGA

```
1: $population\_size = 100; i = 0.$
2: $max\_generation\_number = 500.$
3: $k = 5, learning\_gap = 5, threshold = 0.3.$
4: $generation\_number = 0.$
5: Initialize a new population with $population\_size.$
6: Evaluate current population.
7: repeat
8:    reproduce current population.
9:    if ($generation\_number \% learning\_gap$ == 0) then
10:       copy current population into learning population.
11:       calculate the $H$-group and $L$-group according to threshold.
12:   end if
13:   while ($i < population\_size$) do
14:       mutate a parent individual to generate a new child.
15:       calculate the $k$ nearest neighbours for this child.
16:       if (the majority of this child's $k$ neighbours are nearer to $H$-Group) then
17:          evaluate and place it into the next generation.
18:          $j$++.
19:       else
20:          child is aborted.
21:       end if
22:       apply crossover on two parent individuals in the current population to generate two new children.
23:       for each of these two children, repeat steps 15-21.
24:   end while
25:   $generation\_number$++.
26: until (generation_number == max_generation_number)
```

C. KNNGA 'with verification'

In KNNGA, when a new individual is generated, the fitness of the neighbours of this individual from the *learning population* are checked, and this guides whether or not it enters the population in the next generation. The key aspect (presumably) of the difference between KNNGA and GA (or the difference between a 'learning-guided' search and a pure black box search) is that, in this way, a newly generated individual is discarded before evaluation if we predict beforehand that it will not be good enough.

The flip-side of this, of course, is that we may well admit new individuals into the population that pass this test, but ultimately they prove to be unfit. That is, it could be that the prediction provided by KNN is wrong.

To understand the degree to which this happens, we also test a modification of KNNGA which includes a step of verifying the correctness of the prediction. When an individual is generated by a mutation or crossover operation, as before, KNNGA calculates its *k* nearest neighbours. Again there are two possible cases, for the second case where the majority of its *k* nearest neighbours are in L-group (for maximization problem), this individual is aborted without evaluation; for the first case where the majority of its *k* nearest neighbours are in H-group, this individual is further tested instead of being immediately placed into next generation. That is, after being evaluated, it is compared with a pre-selected value (eg., the worst fitness value in the current population), if this individual's fitness value is higher than this value, then it survives into the next generation; otherwise, it is still aborted. We call this modified version of KNNGA as KNNGA(V) (KNNGA with verification). Compared with KNNGA, KNNGA(V) adds one more condition restricting the new individuals to be able to survive. Namely, in order to survive into the next generation, the new individual should be not only nearer to the H-group, but also better than the worst individual in the current generation. The predictions made by KNNGA are verified as correct or not in this sense. The corresponding KNNGA algorithm should also be modified. KNNGA(V) is the same as KNNGA except that Algorithm 1's 16-21 lines are modified as follows:

This 'with-verification' variant does not at first sight seem well-suited to the goal, in problems with time-expensive fitness functions, of reducing the number of evaluations as much as possible. However, we were interested in any trade-off there may be between the increase in computation time

Algorithm 2 part pseudo-code for KNNGA(V)

```
1: if (the majority of this child's $k$ neighbours are nearer to
   H-Group) then
2:     evaluate this child with the fitness function.
3:     find the worst_fitness value of the current population.
4:     if (fitness(child) > worst_fitness ) then
5:         place this child into the next generation.
6:         j++.
7:     else
8:         child is aborted.
9:     end if
10: else
11:     child is aborted.
12: end if
```

and the quality of solutions obtained.

D. KNNGA and KNNGA(V) Execution time

One of our own motivating factors for investigating LEM-based methods is their promise of speedup on large-scale optimisation problems. That is, achieving good results in relatively few evaluations, which is particularly important when a single evaluation is time-expensive. We therefore provide this simple analysis of execution time for completeness, to better understand how the number of evaluations depends on other aspects of the algorithms studied.

We assume for both KNNGA and GA that the population size is *p*, the maximum number of generations is *M*, the time for evaluating one single individual is t_{eval}, and the crossover operation has the setting m parents and n children (m generally equals to *p*, and $m \geq n$), and the mutation operation has m parents and m children. Meanwhile, t_{search} represents the time spent on searching for a satisfying individual. For the general GAs, the time spent on the whole evolutionary process T_{GA} is calculated by:

$$T_{GA} = (p + M(m + n))t_{eval}$$

There are p evaluations for the initial population, and $m+n$ evaluations for each of the following M generations. The time spent on the evolution/learning process T_{KNNGA} is calculated by:

$$T_{KNNGA} = (p + Mp)t_{eval} + Mpt_{search}$$

Again, p evaluations are needed for the initial population, and p evaluations in each of the following M generations. In addition to the evaluation time, KNNGA needs search time pt_{search} in each M generations.

III. Picturing the LEM(KNN) procedure

In this section, we use a simple example problem to illustrate how KNNGA operates. We assume the problem has a two-dimensional population space, therefore each individual consists of two genes (attributes). The problem is a linear function maximization problem. As KNNGA is running, the sequential populations will be occupied by the individuals nearer to the H-group in the current population.

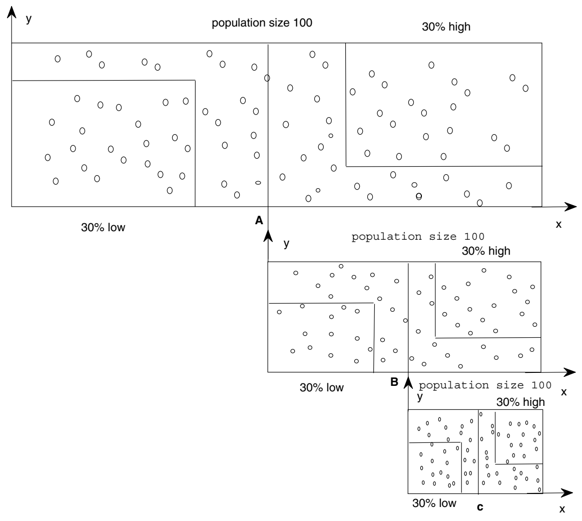

Fig. 1. Example illustration

Figure 1A shows the first generation, the initial individuals are evenly distributed within the whole population space, and for a given *threshold* (eg.,30%) the H-group and L-group are formed.

Figure 1B shows the second generation derived by KNNGA, this population space is crowded by individuals that are within the high fitness half of the first population. Since the degree to which the individuals are now spread out in genotype space is around half what it was previously, the density in genotype space is roughly doubled. This population now undergoes classification into H-group and L-group, resulting in Figure 1C.

Figure 1C shows the third generation of the population, and we see continued reduction in the 'spread' of the population. Clearly, the current whole population has focused on a region increasingly defined by the H-group individuals of the first and second generations.

An obvious and perhaps important aspect of LEM(KNN) (and LEM methods in general, is this strongly defined movement of the population between generations, which is clearly guided (by the results of learning) and less randomised and exploratory than a normal GA. Naturally this has potential drawbacks; we could expect the learning process to misguide the population on certain landscapes, and become stuck in poor regions. Whether or not this generally happens on problems of interest and importance, and (if so) whether the deceptive nature of the landscape is equally deceptive for normal GAs in such cases, are moot points. Empirical evidence to date is suggestive that this general strategy is certainly more often effective than not.

IV. Comparing LEM(KNN) with the corresponding GA

This section describes the experimental results derived from the comparison between KNNGA and the corresponding GA (i.e. the evolutionary algorithm identical to our KNNGA implementation in all respects other than the use of KNN). The test problems used are those that were used in [10] to evaluate the performance of LEM(AQ). In that work, Michalski et al report on two problems from the De Jong's suite [3], and variants are tested with different numbers of dimensions (as in [10]). (Michalski et al also report that similar findings were achieved with the other De Jong problems and are reported in [9]). An additional problem tested in [10] is also tested here; this is from the domain of parameter estimation in nonlinear digital filter design, simulated using equations gleaned from [16].

We were interested in the basic performance of KNNGA vs GA, so that we could sample the degree to which (if any) the LEM framework could be successful when using the simplest possible learning scheme. However we also took the opportunity to contrast with- and without-crossover versions for both the GA and KNNGA. Thus we use notation such as 'GA(m)' (the GA with mutation only) and 'KNNGA(c,m)' (KNNGA with both crossover and mutation).

In all cases, the encoding was a vector of real-valued genes each encoding numbers within a specified interval. We used binary tournament selection [4], elitism (the next generation's population is always initialised with the best of the previous generation), and uniform crossover [12]. Mutation is implemented by randomly adding or subtracting a small value to one gene. For different problems, the values for k, *learning gap* may be different and this is specified later. For each problem, KNNGA and GA use the same initialization method to generate the initial population. For all cases, the population size is 100.

All experiments are repeated 100 times independently to provide sufficient evidence for claims of statistical significance. For statistical analysis, we use randomisation testing [2], which is relatively free of assumptions about the true distributions of the samples involved. As it turns out, the differences in performance as suggested by the plots shown were all confirmed significant at a confidence level of 99.9%, except in those cases where the best two are clearly close (usually KNNGA(c,m) and KNNGA(c,m)(V)), in which case the difference in performance was inconclusive at this confidence level.

Finally, it is worth pointing out again that all algorithms began with the same initial population. It sometimes appears from the graphs (e.g. see figure 2) that the KNN variants began with an advantage, however they didn't. The KNN variants tended to achieve a very rapid improvement in fitness in the first few generations, which is horizontally compressed to almost nothing in the plots.

A. Experimental Study 1: Simple Function Optimization

1) Problem 1: Find the maximum of function f_1 with five variables.

$$f_1(x_1, x_2, x_3, x_4, x_5) = \sum_{i=1}^{5} integer(x_i)$$

$$-5.12 \leq x_i \leq 5.12 \quad Maximum : 25.$$

Figure 2 shows the results of running KNNGA(c,m), KNNGA(m), KNNGA(c,m)(V), GA(c,m) and GA(m) on problem 1. For both of KNNGA and GA, mutation standard deviation is 0.1, mutation rate is 0.2, and crossover is implemented with 100 parents and 10 children. For KNNGA, k is 5, *threshold* is 30%, and *learning gap* is 1.

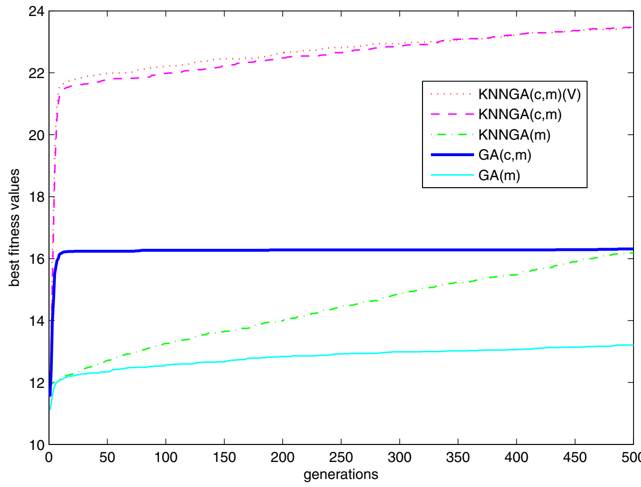

Fig. 2. Results on problem 1 comparing KNNGA(c,m), KNNGA(m), KNNGA(c,m)(V), GA(c,m) and GA(m). This is a maximisation problem. Mean results over 100 trial runs.

All KNNGA variants outperform the GA variants. Within 500 generations, GA(m) only reaches the best fitness value of 13.21, and GA (c,m) reaches 16.31. In contrast, within the same number of generations, KNNGA(m) and KNNGA(c,m) achieve the best fitness values 16.18 and 23.47, respectively. KNNGA(c,m)(V) achieves best fitness value 23.0. It is interesting that the extra evaluation step of KNNGA(c,m)(V) does not yield any advantage in solution quality.

2) Problem 2: Find the maximum of the function f_2 of 30 continuous variables with Gaussian noise:

$$f_2(x_1, x_2, x_3, \ldots, x_{30}) = \sum_{i=1}^{30} ix_i^4 + Gauss(0, 1)$$

$$-1.28 \leq x_i \leq 1.28$$

$$Maximum : approximately\ 1248.225.$$

For this problem, the number of optimum increases as the number of variables scales up. Figure 3 shows the results of running the five KNNGAs and GAs algorithms on problem 2. For both KNNGA and GA, the mutation value is 0.005 due to the smaller variables range (-1.28, 1.28) and the mutation rate is 1/30. Crossover is implemented with 100 parents and 10 children. For KNNGA, k is 5, *threshold* is 30%, and *learning gap* is 1.

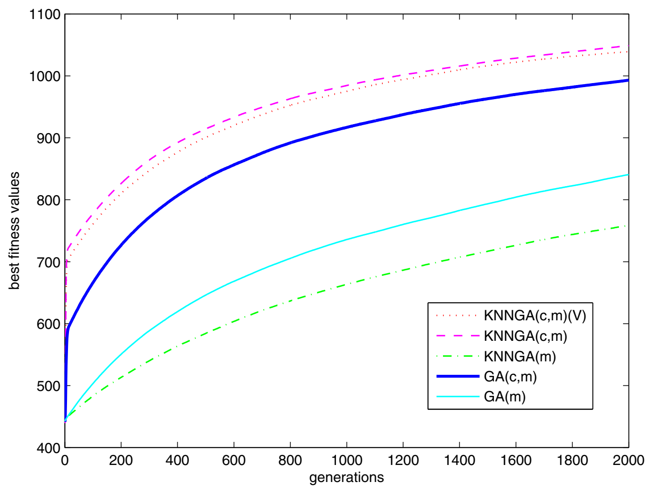

Fig. 3. Results on problem 2 (maximization); mean results over 100 runs.

Within 2000 generations, KNNGA(m) and KNNGA(c,m) reach the best fitness values 758.3 and 1048.7, GA(m) and GA(c,m) can reach 840.7 and 993.1, respectively. KNNGA(c,m)(V) achieves best fitness value 1039.2.

B. Experimental Study 2: Designing Digital Filters

In this study, we test KNNGA algorithm on the problem of parameter estimation for digital filter design. The fitness function was defined by equations specifying linear and nonlinear filters presented in [16].

1) problem 3: Determine optimal parameters of nonlinear filters defined by the equation:

$$\begin{aligned} y(k) = &\left[\frac{3 - 0.3y(k-1)u(k-2)}{5 + 0.4y(k-2)u^2(k-1)}\right]^2 \\ &+ (1.25u^2(k-1) - 2.5u^2(k)) \\ &\times \ln(|1.25u^2(k-2) - 2.5u^2(k)|) + n(k) \end{aligned}$$

where: k is the sample index or time, $n()$ is a noise component ranging from -0.25 to 0.25, and $u()$ is an inserted function (sin, step, random).

The coefficients -0.3, 0.4, 1.25, and -2.5 are assumed as variables which will be optimized and can be seen as the genes of individuals. The problem is to find their correct values using samples $\{\langle vector_i, y(vector_i)\rangle\}$, where $vector_i$ is a specific assignment of values to variables and $y(vector_i)$ is the value of the equation for this assignment. When substituted in the equation, individuals generate a value of y that is compared with the value computed when correct coefficients are used in the equation. The fitness of an individual is defined as in [16] as the reciprocal of the mean-square error over 200 sample window (equation (1)):

$$\begin{aligned} Fitness(Vector) &= \frac{1}{MeanSquareError} \\ &= \frac{200}{\sum_{200}(Vector - KnownValue)^2} \end{aligned} \tag{1}$$

For this minimization problem, the fitness landscape is not clear even for the low variables cases. Figure 4, 5 shows the results of running the five KNNGAs and GAs on problem 3. For both KNNGA and GA, the mutation value is 0.1 and the mutation rate is 1/4. Crossover is implemented with 100 parents and 10 children. For KNNGA, k is 5, *threshold* is 30%, and *learning gap* is 1.

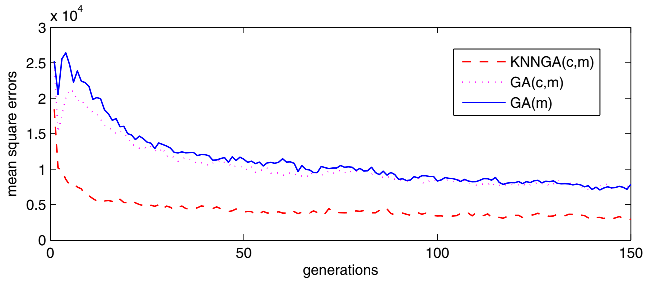

Fig. 4. Results on parameter estimation for nonlinear filter design, problem3; each curve is average of 100 runs, and the problem is minimization.

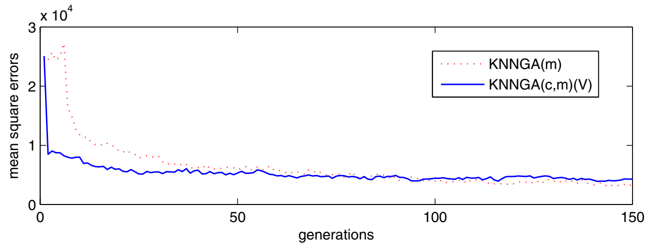

Fig. 5. KNNGA(m) vs KNNGA(c,m)(E) on the nonlinear filter design problem (we divided the results over two plots in this case for ease of visualisation).

The reduction in mean square errors achieved by KNNGA over GA is evident. Within 1500 generations. KNNGA(m) and KNNGA(c,m) reduce the mean square errors to 3426.3 and 2896.7, GA(m) and GA(c,m) can reduce the mean square error values to 7886.5 and 7588.2, respectively. KNNGA(c,m)(V) reaches mean square error to 4301.5.

C. Experimental Study 3: More Complex Function Optimization

1) problem 4: Find the maximum of function f_4 with 100 variables.)

$$f_4(x_i) = \sum_{i=1}^{100} integer(x_i) \qquad -5.12 \le x_i \le 5.12$$

$$Maximum: 500.$$

This is the same problem with problem 1, but with more variables. Figure 6 shows the results of running the five KNNGAs and GAs algorithms on problem 4. The mutation value is 0.1, the mutation rate is (10/100 = 0.1). Crossover is implemented with 100 parents and 10 children. For KNNGA, k is 5, *threshold* is 30%, and *learning gap* is 5.

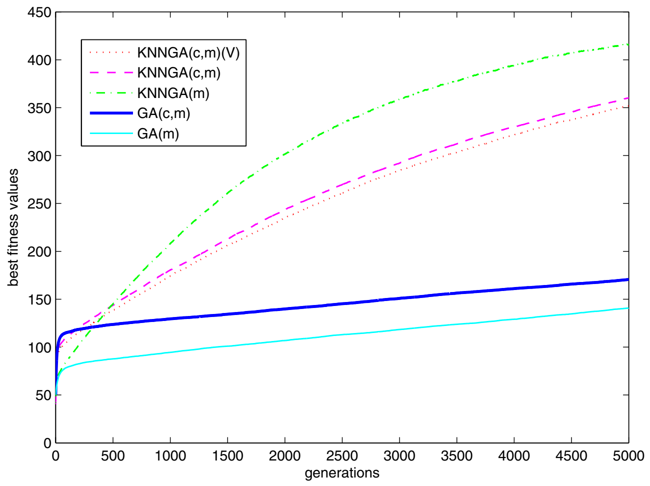

Fig. 6. Results on problem 4; the 100-variable version of problem 1. Each line is the average of 100 runs.

The improvement achieved by KNNGA over GA is evident. Within 5000 generations. KNNGA(m) and KNNGA(c,m) reach the best fitness values 415.7 and 360.1, GA(m) and GA(c,m) reach the best fitness values 140.8 and 170.6, respectively. KNNGA(c,m)(V) achieves 351.87.

2) problem 5: Find the maximum of the function f_5 of 100 continuous variables with a Gaussian noise:

$$f_5(x_1, x_2, x_3, \ldots, x_{100}) = \sum_{i=1}^{100} ix_i^4 + Gauss(0, 1)$$

$$-1.28 \leq x_i \leq 1.28 \quad Maximum : 13556.$$

This is the same problem with problem 2, but with 100 variables. The optima are approached more difficultly than in problem 2. Figure 7 shows the results of running the five KNNGAs and GAs on problem 5. The mutation value is 0.1, the mutation rate is 0.1. Crossover is implemented with 100 parents and 10 children. For KNNGA, k is 5, *threshold* is 30%, and *learning gap* is 5.

The improvement achieved by KNNGA over GA is evident within 60000 generations. KNNGA(m) and KNNGA(c,m) reach the best fitness values 13042.7 and 12787.3, GA(m) and GA(c, m) can only reach 11805.4 and 12269.7, respectively. KNNGA(c,m)(V) reaches 12971.5.

D. Summary of Results

The results on this suite of problems show clear and very significant superiority for the KNN variants. Neglecting the 'with verification' case for the moment, either KNNGA(c,m) or KNNGA(m) were in top place on each problem, and

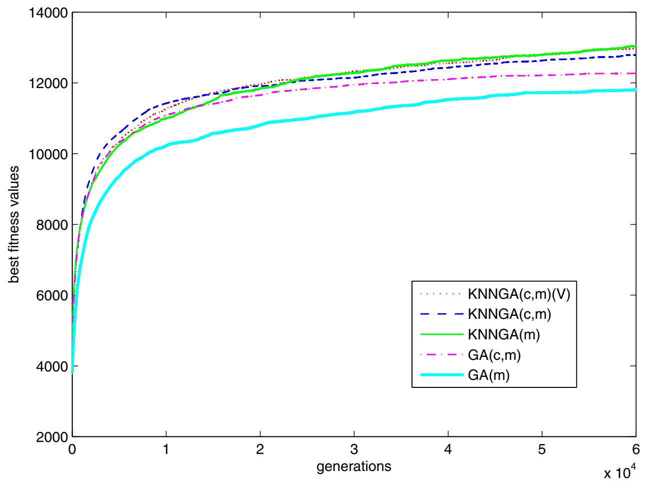

Fig. 7. Results on problem 5, the 100-variable version of problem 2.

always better than the non-KNN versions. The typical result is that the KNN variants show a significant acceleration in fitness in the very early generations, followed by steady further improvement, leaving the ordinary' versions far back in their wake.

These findings reflect those of [9], [10] and other recent LEM works that used more sophisticated learning mechanisms and interaction between the learner and the GA. Since the only application-specific demand of KNN is a suitable distance measure (in that way it is more generally applicable than many other learning mechanisms), it seems fair to say that LEM methods using KNN are clearly recommended for trial in the case of large-scale optimization tasks in which savings in evaluation time are necessary. Far more work needs to be done to establish this properly, however when LEM methods have been tried on large-scale real-world problems so far their promise has indeed been realised [5].

Meanwhile, the performance of the 'with-verification' version of KNNGA was generally not significantly different from that of KNNGA(c,m), which suggests that the 'without' verification version is preferable, simply because it is faster. More interestingly, the lack of a major difference in performance between these two suggests that KNN's predictions, at least in the cases of the problems tested here, are generally not misleading. Finally, it is clear that the differences between GA(m) and GA(c,m) were generally reflected in the differences between KNNGA(m) and KNNGA(c,m).

V. Concluding Discussion

We investigated a simple version of Michalski's LEM [10] which used k-nearest-neighbour as the learning component, and had a straightforward interaction between the learning and the GA, in which new individuals only entered the population if the majority of their k nearest neighbours in the current *learning population* were in the top 30% group. One contribution of this work is the KNNGA algorithm, a simple instantiation of LEM with KNN, which very clearly trounces the corresponding GA in both speed and solution quality. The speed advantage is particularly impressive in general. Another contribution of this work is the fact that the LEM framework has been shown to work well in the context of using perhaps the simplest possible learning method. This is in contrast to published approaches which have either used AQ learners or C4.5. KNN is both simpler and more generic, suggesting that LEM(KNN) may be applied to large-scale optimisation problems independently of the chromosome encoding required, needing only a suitable distance metric to be defined.

We note that it has been difficult to compare our KNNGA with the specific LEM method used in [10], since not all parameters are provided in the LEM paper. However, while the improvements in performance over the GA are similarly vast, it does seem that the LEM(AQ) implementation reported there provides superior results to KNNGA. Two clear explanations for this are available: the simplicity of KNN compared with the relative sophistication of AQ, and the differences in the way that the learning influences the evolution in the two cases. We have deliberately opted for the simplest possible approaches in both cases here, and therefore can show that the bulk of the improvement afforded by the LEM framework is still present in these circumstances, suggesting that the specific choice of learning method and the design of the learning/evolution interaction provide opportunities for further improvement and refinement, rather than being crucial to being able to show superior performance at all.

Continued research on instantiations and variations of the LEM framework are clearly warranted. Lines of work that we expect to explore are: the relationship between the problem landscape and the choice of learning method; the interaction between the learning method and the learning gap, and the use of more than one learning method (with perhaps adaptive techniques to choose between them at different points). Further hybridisation and comparisons with EDA style approaches, and EDA/search hybrids are also warranted. Importantly, however, LEM-based approaches would seem to have much to offer for speedup of large scale optimisation, and we recommend its application to real-world problems of that nature. A specific issue with some possible LEM variants, including the KNN case in many dimensions, is that the learning method itself may take up nontrivial time. This is why we recommend LEM-based research in particular for problems where this 'learning time' remains trivial in comparison to the other aspects of the search, either because a single fitness evaluation takes significant time, or because, very many fitness evaluations are needed, or both.

REFERENCES

[1] E. Bengoextea, T. Miquelez, P. Larranga, J.A. Lozano (2002) Experimental results in function optimization with EDAs in Continuous Domain, in [7].

[2] E.S. Edgington (1995) Randomization tests, 3rd ed. New York: Marcel-Dekker.

[3] De Jong, K. A.,"An Analysis of the Behavior of a Class of Genetic Adaptive Systems", Ph.D. thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, 1975.

[4] D.E. Goldberg (1989) Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley.

[5] L. Jourdan, D. Corne, D. Savic, G. Walters (2005) Hybridising rule induction and multiobjective evolutionary search for optimizing water distribution systems, in Proc of the 4th Hybrid Intelligent Systems conference, published in 2005 by IEEE Computer Society Press. Pp. 434-439, ISBN 0-7695-1916-4.

[6] K. Kaufmann, R.S. Michalski (1999) Learning from inconsistent and noisy data, the AQ18 approach, 11th International Symposium on Foundations of Intelligent Systems.

[7] P. Larranaga, J.A. Lozano (eds) (2002) stimulation of Distribution Algorithms: A New Tool for Evolutionary Computation, Kluwer Academic Publishers.

[8] P. Larranaga, J.A. Lozano, E. Bengoextea (2002) Estimation of Distribution Algorithms based on Multivariate Normal and Gaussian Networks, Technical Report KZZA-1K-1-01, Dept Computer Science and Artificial Intelligence, University of the Basque Country, Spain.

[9] R.S. Michalski and Q. Zhang (1999) Initial experiments with the LEM1 learnable evolution model: an application to function optimization and evolvable hardware. Reports of the machine learning and inference laboratory, George Mason University.

[10] Michalski, R.S., "LEARNABLE EVOLUTION MODEL Evolutionary Processes Guided by Machine Learning," Machine Learning, Vol. 38, 2000, pp. 9–40.

[11] J.M. Pena, V. Robles, P. Larranaga,V. Herves, F. Rosales, M.S. Perez (2004) GA-EDA: Hybrid Evolutionary Algorithm using Genetic and Estimation of Distribution Algorithms, in Orchard, Yang, Ali (eds.) Innovations in Applied Intelligence: 17th Intel Conf. on AI and Expert Systems, Springer LNAI 3029.

[12] G. Syswerda (1989) Uniform Crossover in Genetic Algorithms, Proc. of 3rd International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc.

[13] J. Wnek, K. Kaufmann, E. Bloedorn, R.S. Michalski (1995) Inductive Learning System AQ15c: the method and user's guide. Reports of the Machine Learning and Inference Laboratory, MLI95-4, George Mason University,Fairfax, VA, USA.

[14] J. Wojtusiak, R.S. Michalski (2005) The LEM3 System for Non-Darwinian Evolutionary Computation and Its Application to Complex Function Optimization, Reports of the Machine Learning and Inference Laboratory, MLI 05-2, George Mason University, Fairfax, VA, USA.

[15] J. Wojtusiak, R.S. Michalski (2006) The LEM3 implementation of learnable evolution model and its testing on complex function optimization problems, in Proc. GECCO 2006.

[16] L. Yao and W. Sethares (1994) Nonlinear parameter estimation via the genetic algorithm. IEEE Trans. on Signal Processing, 42(4):927–935.

[17] Q. Zhang, J. Sun, E. Tsang, J. Ford (2003) Hybrid estimation of distribution algorithm for global optimisation, Engineering Computations 21(1): 91–107.

[18] Q. Zhang, J. Sun, E. Tsang, J. Ford (2006) Estimation of distribution algorithm with 2-opt Local Search for the Quadratic Assignment Problem, in Lozana, Larranaga, Inza and Bengoetxea (eds) Towards a new evolutionary computation: advances in estimation of distribution algorithms.