Brock University

Department of Computer Science

An Examination of Lamarckian Genetic Algorithms

Cameron Wellock and Brian J Ross
Technical Report # CS-01-01
July 2001

Published in the Proceedings of Late Breaking Papers, Genetic and Evolutionary Computation Conference (GECCO-2001), pp.474-481.

Brock University
Department of Computer Science
St. Catharines, Ontario
Canada L2S 3A1
www.cosc.brocku.ca

An Examination of Lamarckian Genetic Algorithms

Cameron Wellock

Brock University, Dept. of Computer Science
St. Catharines, Ontario, Canada L2S 3A1
cw96af@cosc.brocku.ca

Brian J. Ross

Brock University, Dept. of Computer Science
St. Catharines, Ontario, Canada L2S 3A1
bross@cosc.brocku.ca

Abstract

In keeping with the spirit of Lamarckian evolution, variations on a simple genetic algorithm are compared, in which each individual is optimized prior to evaluation. Four different optimization techniques in all are tested: random hillclimbing, social (memetic) exchange, and two techniques using artificial neural nets (ANNs). These techniques are tested on a set of three sample problems: an instance of a minimum-spanning tree problem, an instance of a travelling salesman problem, and a problem where ANNs are evolved to generate a random sequence of bits. The results suggest that in general, social exchange provides the best performance, consistently outperforming the non-optimized genetic algorithm; results for other optimization techniques are less compelling.

1 INTRODUCTION

1.1 LAMARCK'S IDEAS OF EVOLUTION

By far the most common understanding of evolution today is based on the work of Charles Darwin, whose ideas have not only shaped the modern science of biology but have also influenced computer science, through the development of genetic algorithms and similar techniques. Darwin's model of evolution through natural selection however is not the only possible model, nor was it the first.

In 1801, a Frenchman by the name of Jean-Baptiste de Monet, Chevalier de Lamarck published his own theory of evolution (Burkhardt, 1977), almost sixty years before Darwin published his famous Origin of Species (Darwin, 1859). Natural selection did not play a part in Lamarck's theory; rather, plants and animals adapted to their environment over the course of their lifetime, and these adaptations were passed on directly to their offspring. To cite an example from Lamarck (1801) himself:

> The bird attracted by need to the water to find there the prey necessary for its existence, spreads the digits of its feet when it wishes to strike the water and move on the surface. The skin that unites these digits at their base thereby acquires the habit of stretching itself. Thus, with time, the large membranes uniting the digits of ducks, geese, etc. have been formed such as we see them today.
>
> But the bird whose way of life habituates it to perch in trees has necessarily the digits of its feet extended and shaped in another way. Its claws are elongated, sharpened, and curved in a hook to grasp the branches on which it often rests.
>
> In the same way one may perceive that the bird of the shore, which does not at all like to swim, and which however needs to draw near to the water to find its prey, will be continually exposed to sinking in the mid. Wishing to avoid immersing its body in the liquid, [it] acquires the habit of stretching and elongating its legs. The result of this for the generations of these birds that continue to live in this manner is that the individuals will find themselves elevated as on stilts, on long naked legs... (pp. 13-14)

While Lamarck's beliefs lie generally discredited in the field of biology (Burkhardt, 1977), one may yet wish to revisit his ideas: while Lamarckian evolution is not found in nature, it is entirely possible to implement systems based on his ideas in software.

1.2 AN OVERVIEW OF TRADITIONAL GENETIC ALGORITHMS

The Darwinian model of evolution has been used successfully to solve a number of search and optimization problems using computers (Mitchell, 1996), commonly using what is referred to as a "genetic algorithm." The basic operation of a genetic algorithm, or GA, is as follows:

A population of individuals is randomly generated. Each individual represents a potential solution to the problem at hand: every individual can be evaluated using a "fitness function" to determine how well it solves the given problem.

Individuals from the population are chosen for reproduction. The individuals are chosen randomly, with some sort of bias in favour of individuals who perform

well in the fitness evaluation. Using "genetic operators," new individuals are generated which are based in part upon the encoding of their "parent" individuals.

As the simulation runs, the problem "search space" is explored in an effort to find an optimal or near-optimal solution (Mitchell, 1996). The fact that individuals are selected for reproduction in a biased fashion means that on the whole, the simulation will spend most of its time examining good solutions rather than wasting its time on bad ones.

One of the most important parts of any genetic algorithm is the selection mechanism, by which individuals are chosen for reproduction. A popular selection mechanism is the tournament selection: an arbitrary subset of the total population is selected at random; the individual with the best score from the fitness function is then chosen for reproduction.

Another important aspect of a GA is the set of genetic operators used to produce new individuals. Some of the most common operators are:

- Crossover—in crossover, two (or possibly more) individuals are combined to create a new individual. Several variations on crossover exist, which vary primarily in terms of how the components from the parent individuals are combined. Crossover serves to quickly propagate beneficial pieces of an overall solution across the larger population, and is usually the most-commonly invoked genetic operator.
- Mutation—mutation involves randomly modifying some part of a parent individual. Mutation serves to introduce new potential solutions to the population, and also to help delay the onset of convergence—the tendency of any GA to breed populations of increasingly similar individuals, which makes crossover useless and effectively stops evolution. Convergence may be problematic if the simulation converges before finding an optimal solution.
- Reproduction (copying)—reproduction simply copies an individual, without change, and inserts the new copy into the population. This is useful in order to help prevent the loss of particularly good individuals in the population.

Another aspect of GAs are how they handle the updating of the population: GAs which replace the entire population in a turn-based fashion are called generational; this is in contrast with steady-state GAs, which continuously replace small parts of the population with new individuals. (Mitchell, 1996)

1.3 A LAMARCKIAN APPROACH TO GENETIC ALGORITHMS

The strictest interpretation of Lamarck's ideas—that evolution could take place strictly by means of individual adaptation—would not translate into anything resembling a genetic algorithm (GA) when applied to computing. This view would instead seem to represent a kind of hillclimbing search or beam search. As we wish to remain focussed on GAs in this paper, we will instead apply the key elements of Lamarckian evolution to a traditional genetic algorithm.

The central idea of Lamarck's vision (when contrasted with Darwin's) is that evolution may take place by means of individuals adapting to their environment, and passing these adaptations on to offspring. In the context of a genetic algorithm, this would represent some kind of optimization step before the individual is evaluated, in which the results of the optimization were permanently written back into the individual. An examination of such optimization techniques are the subject of the experiments described in this paper.

2 LITERATURE SURVEY

A survey of the existing literature reveals that many elements of this paper have been examined by others in some form. A number of publications have explored the use of hybrid genetic algorithms; among these are Hart and Belew, who examined the use of genetic algorithms with local optimizers, and explicitly noted the Lamarckian nature of such techniques (1996). Whitley, Gordon, and Mathias have observed that Lamarckian evolution may be faster than a simple genetic algorithm, but note that it may also encourage premature convergence to local optima (1994). Ackley and Littman likewise note that Lamarckian evolution drives convergence (1994). Dozier, Bowen, and Homaifar have reported good results when using a hybrid evolutionary search (1998). One paper by Kaytama, Sakamoto, and Narihisa explores a hybrid genetic algorithm using hillclimbing as an optimization on the Travelling Salesman Problem, and finds positive results when compared with a non-optimized GA (2000). Cheng and Gen have explored the use of memetic optimizations, again with positive results (1997).

Several papers exist describing more strictly Lamarckian systems, in which the optimizations are in fact based on an individual's response to its environment; clearly such an optimization is possible only in certain applications. Grefenstette (1991) and Li, Tan, and Gong (1996) have all written on such systems; both use the evolution of control systems as the basis of their applications.

Research has been conducted on using more sophisticated techniques for optimization, in a line of inquiry similar to the ANN-based optimization in this paper. Most of these have involved the use of problem-specific heuristics; papers by Cheng, Gen, and Tsujimura (1999), and by Gen, Ida, and Li (1998) explore such heuristic optimizations.

A number of papers have explored the possibility of using genetic algorithms with ANNs: Hinton and Nowlan have experimented with using GAs to evolve ANNs, which in turn were allowed to adapt by means of backpropagation-based learning (1996); this GA was not Lamarckian however as these adaptations were not subsequently passed on. Papers by Kim and Han (2000)

and by Han, Moraga, and Sinne (1996) have examined the use of GAs as tools to optimize the configuration of ANNs, and a paper by Sexton, Dorsey, and Johnson compares the performance of ANN learning by backpropagation to direct evolution of ANNs; the findings suggest that direct evolution of ANNs may be successful in some problem domains where backpropagation is less so.

3 THE EXPERIMENTS

For this paper, a simple genetic algorithm was modified to use a series of different optimization techniques; each of these techniques was tested on a set of three different problems.

Notwithstanding the differences in optimization technique and problem, all experiments used the same underlying GA, implemented in the Python programming language. This was a generational GA using tournament selection. Individuals in all problems were represented as binary vectors; the length and interpretation of these binary vectors was dependent on the problem at hand.

Each possible combination of optimization technique and problem was tested in a series of ten runs, except for the ANN-based optimizations, which were tested in a series of three runs. All runs were conducted in the same environment (a 350-Mhz Pentium II, with no other significant processes running); effectiveness was measured in terms of both absolute (wall-clock) and relative (per-generation) performance.

3.1 OPTIMIZATION TECHNIQUES

Four different optimization techniques were used in the experiments; runs with no optimization were also included for comparison.

3.1.1 Hillclimbing

In a hillclimbing optimization, the candidate individual in improved in a series of "steps": a new individual is created which is changed slightly from the original in a random fashion (essentially a mutation operation). This new individual is compared with the original; if the new individual has a better fitness level, then this new individual is kept; otherwise the original individual is retained. This modify-and-test sequence is repeated some number of times, so that the optimized result may be significantly better than the original.

From a performance perspective, hillclimbing has a number of desirable attributes: it is usually not expensive computationally, and it can significantly enhance the overall fitness of the population in a short time—with hillclimbing optimization, individuals in the population need only be near a local optimum in order to reach it.

The disadvantage to hillclimbing, like most optimization techniques, is that it promotes convergence (Ackley & Littman, 1994)—individuals who are somewhat different when created may be converted into identical or near-identical individuals by the hillclimbing optimization, reducing overall diversity in the population.

3.1.2 Social Exchange

While hillclimbing works by trial-and-error, testing out random variations in the hope that one will be advantageous, social exchange works by using components from a more reliable source: other, better-performing individuals in the same population.

Social exchange is also frequently referred to as memetic exchange, and is based upon information exchange in human societies: rather than having to evolve the ability to start fire for example, humans can simply demonstrate this skill to others. In a similar fashion, social exchange takes an individual of high fitness in the population and combines that with the candidate individual, in the hopes of creating a more-fit version of the original candidate. As hillclimbing is analogous to a mutation operation, social exchange is analogous to a crossover operation, combining elements from what already exists in the population.

The primary disadvantage of social exchange is that it drives convergence, to a degree even greater than that of hillclimbing. Because the source of optimization material is other individuals, social exchange may promote overall population fitness at the expense of population diversity.

3.1.3 Artificial Neural Networks

Another possibility is to use a more sophisticated type of optimization. While hillclimbing and social exchange may be able to improve a candidate individual, they do so essentially by chance—no analysis of the candidate is done before optimization.

The difficulty of course is in deciding what analysis to use, and how to identify the important features of a good candidate. Artificial Neural Networks here become a viable optimization technique: by nature, ANNs excel at identifying patterns and trends. (Watson, 1997)

Structure of the ANNs

All of the ANNs used in this paper are of a similar structure. The fundamental building block of ANNs is the artificial neuron; in this paper we limit ourselves to a simple kind of artificial neuron called a perceptron.

A perceptron is a simplified mathematical model of a neuron, consisting of a set of input weights w, and a firing threshold t. To make use of a perceptron, a set of input values v are multiplied against the corresponding input weights and summed; if the total is greater than the firing threshold then the perceptron "fires", emitting a "1" value; otherwise the perceptron emits a "0" value.

The ANNs used in this paper are all 3-layer feed-forward ANNs: the perceptrons composing each ANN are arrayed into a set of input perceptrons, a middle set of "hidden"

perceptrons, and an output layer. Each layer recieves input only from the preceding layer, and outputs only to the following layer; the input layer must receive its input from an external source, and the output layer must likewise output its results in some other fashion.

When used as an optimizer, the ANN receives as input the encoding for the candidate individual; the output of the ANN is then taken as the optimized result.

Coevolution of Optimizing ANNs

ANNs are used in two different ways in these experiments. The first is where a population of ANNs are co-evolved along with the main population. These ANNs are evaluated based on the success of their evaluations: every individual in the population is given the chance to optimize the candidate individual; the performance of the resulting individuals is used to decide on the fitness of the ANNs themselves.

Co-evolving the optimizers along with the population to be optimized may seem counterproductive, as the optimizers may find themselves trying to "keep up" with the main population. Indeed, no great expectations can be placed upon the co-evolving ANNs. Instead, we wish to use the resulting population of ANNs for another purpose: a set of previously-generated ANNs could be used to optimize the main population without the computational overhead of evolving the ANN population.

Optimization Using Pre-generated ANNs

Instead of coevolving ANNs, we may instead wish to use a set of pre-generated ANNs as optimizers. Such ANNs do not adapt as the run progresses; however the computational costs of using these ANNs as optimizers is far lower without the overhead of a second evolving population. Conveniently, the ANNs generated by the co-evolving runs should be ideal for use as a static set of optimizers.

The primary disadvantages to such a technique are computational complexity and unpredictability: co-evolving a set of neural nets can be several times more expensive in terms of both memory and speed; in addition allowing neural nets to essentially configure themselves means that we must take it on faith that the ANNs will arrive at sensible conclusions about what does and does not constitute a beneficial optimization.

Early tests using ANNs as optimizers revealed that given the opportunity, the ANNs would quickly converge to an "optimization-by-duplication" strategy, in which they would optimize by converting every single input instance into a reasonably good, but identical or near-identical, output instance; such an optimization would essentially eliminate all diversity in the population in a single generation. To avoid this, the ANNs were used more like intelligent mutators: a small subset of the ANN's optimized individual (approx. 5%) would be randomly selected and incorporated back into the candidate individual, allowing the ANN to make improvements but preventing it from utterly destroying the original.

3.2 THE TEST PROBLEMS

A set of three test problems were used to gauge the performance of the various optimization techniques. The tests chosen represent a range of difficulties, based on how hard the problem is to solve by traditional means.

3.2.1 Minimum-Spanning Tree Problem

The Minimum-Spanning Tree (MST) is a well-understood problem, for which a number of polynomial-time algorithms exist to find solutions (Grimaldi, 1994). Because the MST problem is well-understood, and is a provably "tractable" problem, MST was chosen as a problem for which a GA should be able to perform reasonably well.

The MST problem consists of a connected graph with weighted edges, on which one must remove edges in order to minimize the total of all edge weights in the graph, while ensuring that the graph remains connected.

All of the experiments based on the MST problem used the same 38-vertex graph, which was generated by hand.

The genetic encoding for the MST problem is very simple: one bit for each edge in the graph; if the bit is set, the edge is included in the individual.

Individuals are evaluated in the MST problem as follows: the sum of the weights of all edges in the individual is taken, and the resulting graph is examined to find out how many vertices can be reached from vertex 0. For each unreachable vertex, a "penalty weight" is added to the sum which is of higher value than the highest possible legitimate route.

3.2.2 The Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is another well-known problem which can be used to test GAs. While based on a graph like the MST problem, the TSP is a NP-complete problem and hence cannot be solved in polynomial time (Arora, 1996); this makes it a reasonable target for GAs, which may be able to find an approximate solution in far less total time.

The TSP involves finding a path through the graph such that each vertex is visited exactly once, with the trip finishing on the starting vertex. The objective is to find a route of minimum total distance (weight).

The experiments in this paper use a symmetric TSP, which essentially means that all edges are bidirectional, and if an edge e between vertices v_1 and v_2 exists, then the cost to travel from v_1 to v_2 is the same as the cost to travel from v_2 to v_1. All experiments used the same 16-vertex graph for their test of the TSP.

The genetic encoding for the TSP is somewhat more sophisticated than that for the MST. The individual is

divided into a series of eight-bit blocks, one block for every vertex in the graph. These eight-bit values are decoded into an integer in the range of -128 to +127; this value is used to find a vertex index by way of modulo arithmetic. Each vertex index points to the next vertex in the tour; for example a value of 5 in the third index location represents a move from vertex three to vertex five in the tour.

3.2.3 The Pseudo-Random Bit Generation Problem

The final problem uses ANNs to generate a series of bits; the ANNs are evaluated based on the "randomness" of the resulting bit sequence. This problem was inspired by the work of Koza (1992), who used cellular automata to generate pseudo-random numbers.

This problem is even less tractable than the TSP; while the TSP may not have efficient algorithms to find an optimal solution, the problem of creating random-bit generators has no algorithm to find an optimal solution whatsoever. It was for this reason that the PRBG problem was chosen.

Operation of the PRBGs

To create a random bit sequence, the ANNs must be provided with an appropriate set of input values. In the PRBGs generated in this paper, four bits of input data were provided from which to generate a single bit of output data. At this point, the leftmost bit of input data was discarded, and the output bit was appended to the input data, to create the next element in the input sequence. This is in keeping with the general operation of pseudo-random number generators, which depend upon previous values for the generation of additional values. With four bits of input data, each PRBG has a total of sixteen possible states; this is a fairly low number but in the interests of keeping the problem computable in a reasonable time frame this was a necessity. The PRBGs were given four random seed bits and asked to produce 128 random bits, giving ample time to exhibit any patterns which might be in their output.

Each PRBG was encoded as a large sequence of eight-bit blocks; these blocks were decoded into integer values as in the TSP problem and used as the input weights and threshold values in the construction of an ANN.

Testing the PRBGs

The greatest difficulty in evolving PRBGs lies in testing the "randomness" of the sequences produced; GAs in general are very good at exploiting weaknesses in fitness functions and may end up creating random-number-test-breakers rather than random bit sequences as hoped. A set of three tests were used on the random bit sequences, which seemed to do a reasonable job of identifying random-looking bit sequences. These tests were based on tests used for more general random-number generators described in Knuth (1969).

The first test was a simple frequency distribution test of the bits; on average a good random bit generator should produce as many 1 bits as 0 bits. The second test was a frequency distribution test for pairs of bits; on average a good random bit generator should produce equal distributions of all possible two-bit sequences.

The final test used was a run-length distribution test. Based on the Run Test given in Knuth (1969), it tries to ensure that the distribution of continuous-sequence length approximates that of a fair random-bit generator.

By an empirical analysis using what one may assume to be a fair random-bit generator (the Python whrandom.randint() function), it was determined that the proper distribution of continuous-sequence lengths was such that there should be twice as many sequences of length k as sequences of length $k+1$. The PRBGs were therefore tested according to how close their continuous-sequences matched this distribution.

3.3 EXPERIMENTAL PARAMETERS

The following table describes the parameters used in this experiment.

Table 1: Experimental Parameters

PARAMETER	VALUE
General Parameters	
Selection Method	Tournament (size 3)
1-pt Crossover Probability	75%
Mutation Probability	15%
Mutation Impact	~5% of target randomized
Copy Probability	10%
Runs per Experiment	10 (3 for ANN optimizers)
Parameters for MST, TSP	
Population Size	100
Generations	50
Parameters for PRBG	
Population Size	50
Generations	25
Hillclimbing Parameters (All Problems)	
Max. Steps	3
Per-Step Impact	~5% of target randomized
Social Exchange Parameters (All Problems)	
1-pt Exchange Probability	80%

2-pt Exchange Probability	20%

ANN Optimization Parameters (MST, TSP Only)

ANN Population Size	10
ANN Selection Method	Tournament (size 3)

ANN Optimization Parameters (PRBG Only)

ANN Population Size	5
ANN Selection Method	Tournament (size 2)

4 EXPERIMENT RESULTS

The following sections provide the results of the experiments. Figures 1 – 6 illustrate these results. Two types of graphs are provided; one based on per-generation performance, and one based on run-time performance, which is probably a more realistic measure of efficiency. All graphs represent the mean population performance, averaged across all runs. All of the graphs share a common legend: "NO" indicates no optimization, "HC" indicates hillclimbing, "SE" indicates social exchange, and "NN" indicates one of the neural net optimizations.

4.1 MINIMUM-SPANNING TREE PROBLEM

In terms of relative (per-generation) performance, both social exchange and hillclimbing optimizations outperformed the no-optimization default early on, with all three methods converging at similar values later in their runs. Overall, social exchange provided the best performance. What is of interest is that both ANN optimizations performed considerably worse, converging at local minima early on. The ANN optimization based on pretrained ANNs performed marginally better than the ANNs being co-evolved along with the main population.

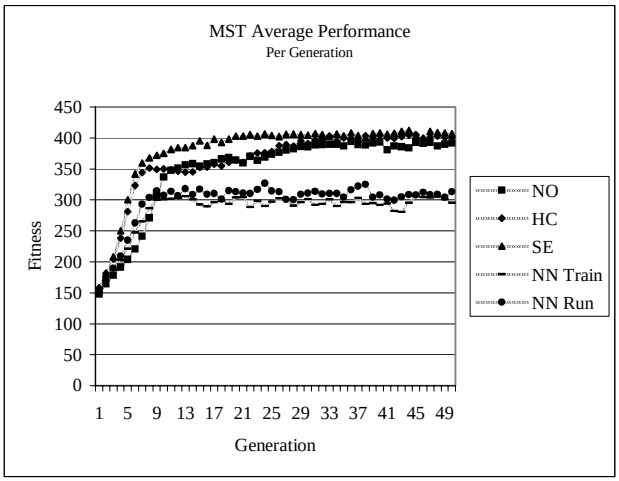

Figure 1: MST Average Performance, per Generation

In terms of absolute performance, social exchange proved superior, working even more quickly than the no-optimization default; although its run took longer in total, it also found better solutions to the problem. Hillclimbing performed somewhat worse, and not unexpectedly the ANN optimization runs took considerably longer.

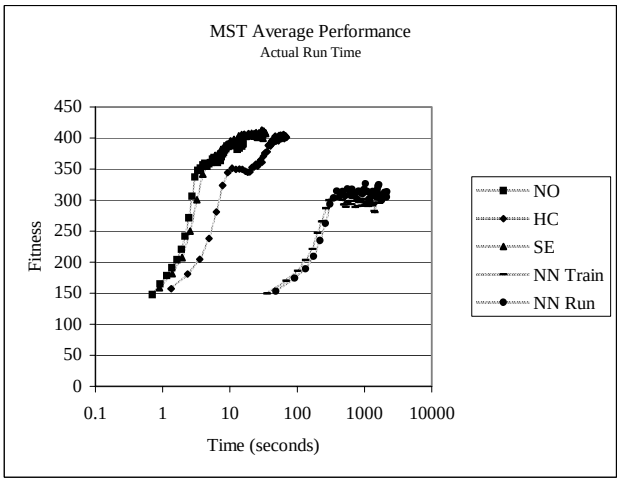

Figure 2: MST Average Performance, by Run Time

4.2 TRAVELLING SALESMAN PROBLEM

For relative performance, social exchange and hillclimbing again outperformed the no-optimization default, although with hillclimbing ultimately achieving better results. Once again, the ANN optimizations fared poorly, indeed even more so than in the MST problem.

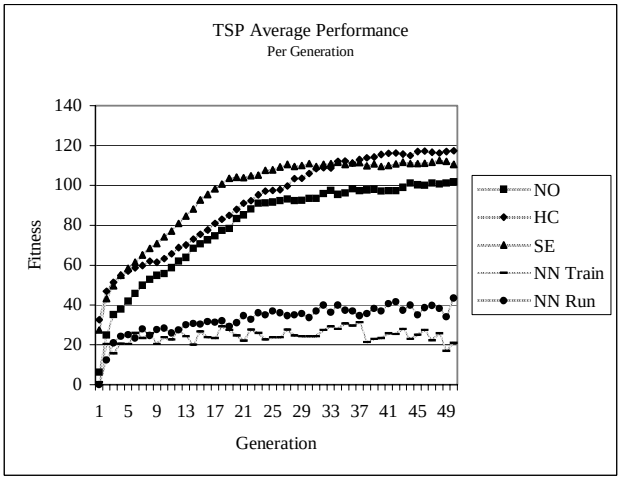

Figure 3: TSP Average Performance, per Generation

For absolute performance, social exchange found better solutions faster than any other method; of course as mentioned above, hillclimbing ultimately outperformed it.

The non-optimizing default did nearly as well as social exchange, however.

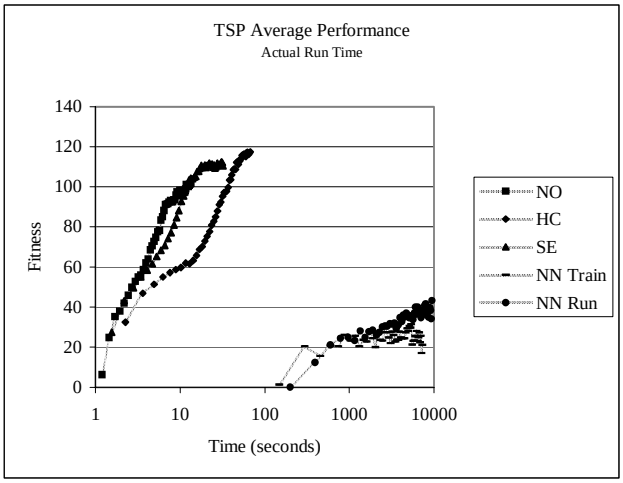

Figure 4: TSP Average Performance, by Run Time

4.3 PSEUDO-RANDOM BIT GENERATION PROBLEM

The PRBG problem in many respects was the most difficult of all of the problems, and so perhaps its results are most telling. In terms of relative performance, social exchange was clearly the superior optimization, finding better results more quickly than any other method. Hillclimbing outperformed the no-optimization default in the best-case analysis, but not in the average-case analysis. The ANN optimizations continued to underperform relative to the other optimization techniques.

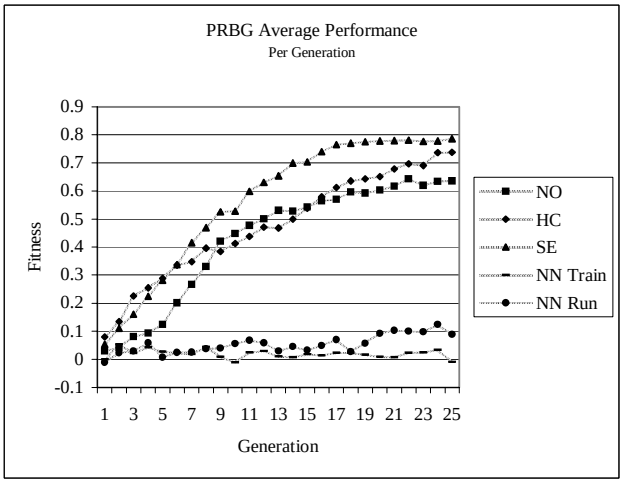

Figure 5: PRBG Average Performance, per Generation

For absolute performance, social exchange once again proved superior, although at least until no-optimization becomes trapped on a local maximum, it remains competitive with social exchange. The ANN methods continue to underperform, especially in light of the extraordinary processing time they require.

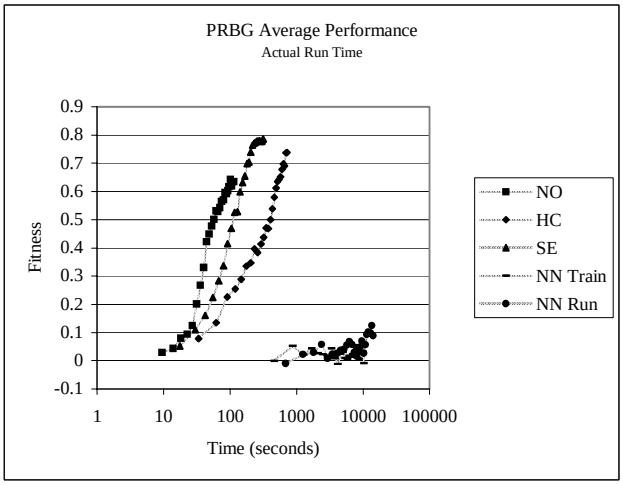

Figure 6: PRBG Average Performance, by Run Time

5 CONCLUSION

5.1 PROBLEMS WITH THE EXPERIMENTS

A number of potential problems within the experiments exist. The primary problem involves the implementation of the optimization modules: while reasonable care was taken coding the modules, the particular implementations may have fallen far short of the best possible outcomes, most notably in terms of run time, although also in terms of results achieved.

The clock used to time runs (the computer's system clock) is of course an imprecise measuring tool, and other processes running at the same time may have influenced the time for any particular run.

5.2 MEANING OF THE RESULTS

Overall, the results suggest that social exchange is a worthwhile optimization for most problems—in terms of both absolute and relative performance, social exchange routinely outperformed the non-optimized default. Hillclimbing seems to find better results than the default, although in absolute performance terms it does so at a slower rate. Different implementations of a hillclimbing optimization may perform more quickly.

The worst results overall, and perhaps the most interesting, were those for the ANN-based optimizers. These optimizers routinely fared worse than the default GA. If in some sense the TSP is more "difficult" than the MST problem, and the PRBG problem is again more

"difficult" than the TSP, then the general trend would be that the ANN-based optimizers perform progressively worse on more difficult problems. While a detailed analysis of why exactly the ANN-based optimizers fared so poorly is beyond the scope of this paper, the results would seem to indicate that the ANN-based optimizers strongly encourage convergence of the GA to a non-optimal solution.

Acknowledgements

The authors would like to acknowledge the contributions of the following, whose assistance (direct or indirect) helped to make this paper possible.

- G. Strangman, whose Chi-square functions in his stats.py module were essential in the implementation of the pseudo-random bit generator tests;
- Stephen Pinker, whose discussion of Lamarckian evolution (1997) led ultimately to this paper.
- This research was partially funded by NSERC operating grant 138467-1998.

References

Ackley, D. H. & Littman, M. L. A Case for Lamarckian Evolution. (1994). In C. G. Langton (Ed.), *Artificial Life III*. (pp. 3-11). Reading, MA: Addison-Wesley.

Arora, Sanjeev. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. (1996). In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*. (pp. 2-12).

Burkhardt, Richard W., Jr. (1977). *The Sprit of System*. Cambridge, MA: Harvard University Press.

Cheng, R. & Gen, M. (1997). Parallel Machine Scheduling Problems Using Memetic Algorithms. *Computers & Industrial Engineering*, *33*(3-4), 761-764.

Cheng, R., Gen, M., & Tsujimura, Y. (1999). A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: hybrid genetic search strategies. *Computers & Industrial Engineering*, *36*, 343-364.

Darwin, Charles. (1859) . *On the Origin of Species by Means of Natural Selection*. London: J. Murray.

Dozier, G., Bowen, J., & Homaifar, A. (1998). Solving Constraint Satisfaction Problems Using Hybrid Evolutionary Search. *IEEE Transactions on Evolutionary Computation*, *2*(1), 23-32.

Gen, M., Ida, K. & Li, Y. (1998). Bicriteria Transportation Problem by Hybrid Genetic Algorithm. *Computers & Industrial Engineering*, *35*(1-2), 363-366.

Grefenstette, J. Lamarckian Learning in Multi-agent Environments. (1991). In *Proceedings of the Fourth International Conference on Genetic Algorithms*. (pp. 303-310). San Mateo, CA: Morgan Kaufmann.

Grimaldi, R. (1994). *Discrete and Combinatorial Mathematics*. Reading, MA: Addison-Wesley.

Han, J., Moraga, C., & Sinne, S. (1996). Optimization of Feedforward Neural Networks. *Engineering Applications of Artificial Intelligence*, *9*(2), 109-119.

Hart, W. E., & Belew, R. K.. Optimization with Genetic Algorithm Hybrids that Use Local Search. (1996). In R.K. Belew & M. Mitchell (Eds.), *Adaptive Individuals in Evolving Populations*. (pp. 483-496). Reading, MA: Addison-Wesley.

Hinton, G. E. & Nowlan, S. J. How Learning Can Guide Evolution. (1996). In R. K. Belew & M. Mitchell (Eds.), *Adaptive Individuals in Evolving Populations*. (pp. 447-457). Reading, MA: Addison-Wesley.

Katayama, K., Sakamoto, H. & Narihisa, H. (2000). The Efficiency of Hybrid Mutation Genetic Algorithm for the Travelling Salesman Problem. *Mathematical and Computer Modelling*, *31*, 197-203.

Kim, K. & Han, I. (2000). Genetic algorithms approach to feature discretization in artificial neural networks for the prediction of stock price index. *Expert Systems with Applications*, *19*, 125-132.

Knuth, Donald. (1969). *The Art of Computer Programming*, vol. 2. Reading, MA: Addison-Wesley.

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

Lamarck, Jean-Baptiste. (1801). *Système des animaux sans vertèbres*. Paris.

Li, Y., Tan, K. C., & Gong, M. Model reduction in control systems by means of global structure evolution and local parameter learning. (1996). In D. Dasgupta & Z. Michaelwicz (Eds.), *Evolutionary Algorithms in Engineering Applications*. New York: Springer-Verlag.

Mitchell, Melanie. (1996). *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press.

Pinker, S. (1997). *How the Mind W*orks. New York: W. W. Norton & Co.

Watson, M. (1997). *Intelligent Java Applications for the Internet and Intranets*. San Francisco: Morgan Kaufmann Publishers.

Whitley, D., Gordon, V. S., & Mathias, K. Lamarckian Evolution, the Baldwin Effect and Function Optimization. (1994). In Y. Davidor et al. (Eds.), *Parallel Problem Solving From Nature*, vol. 3. (pp. 6-15). New York: Springer-Verlag.