A Lamarckian Evolution Strategy for Genetic Algorithms

Brian J. Ross
Brock University
Department of Computer Science
St Catharines, Ontario, Canada
email: bross@cosc.brocku.ca

Abstract

A Prolog implementation of a simple Lamarckian evolution module for genetic algorithms is discussed. Lamarckian evolution posits that characteristics acquired during a phenotype's lifetime may be inherited by offspring. Although largely dismissed as a viable evolutionary theory for natural systems, Lamarckian evolution has proven effective within computer applications. The strengths of the implementation discussed here are its speed and simplicity -- the latter promoting extensibility and specialization to particular applications.

1. Introduction

Prior to Charles Darwin's theory of evolution by natural selection, Jean Baptiste Lamarck (1744-1829) proposed a multifaceted theory of evolution [Dawkins 1996, Gould 1980, Cochrane 1997]. One aspect of his theory is the notion that characteristics acquired by an organism during its lifetime are inheritable by its offspring. Lamarck proposed this as the means by which organisms passed on specialized traits for surviving in the environment, and this has since become known as Lamarckian evolution or Lamarckism. For example, if a horse developed especially adept leg muscles for negotiating mountainous terrain, Lamarckian evolution suggests that its offspring would inherit similarly muscular legs. Lamarckian inheritance of acquired characteristics maintains that the acquired development of strong legs, through years of exercise in a mountainous environment, will influence the actual genetic makeup of the horse. This altered genetic information is inheritable by the horse's offspring. This contrasts with the Darwinian tenet that a horse that is genetically predisposed to having muscular legs will probably have offspring with a similar genetic tendency.

Lamarckism has been universally rejected as a viable theory of genetic evolution in nature. There are hundreds of millions of genetic variations in a typical DNA of an organism. Physical characteristics of a phenotype are the result of combined interactions between many separate components of the

DNA. In addition, the acquired characteristics of a phenotype are manipulations of the organism's tissues as permitted within their range of possible forms as dictated by the genotype. Darwinian evolution proposes that the complex structure of DNA results in sensible phenotypes due to the accumulation of millions of years of minute genetic mutations, and their subsequent success through natural selection. On the other hand, the mechanism required by Lamarckian evolution is that physical characteristics of the phenotype must in some way be communicated backwards to the DNA of the organism. The biological mechanism by which such communication would arise is unknown. Furthermore, the inversion of gross physical characteristics of the phenotype into the many genetic codes needed to reproduce them in subsequent generations seems to be an impossibly complex task. It is worth mentioning that Lamarckian effects have been conjectured in the context of some experiments with cellular life forms [Cochrane 1997]. These recent results are somewhat of a reprieve for the disrepute that Lamarck's evolutionary theory has suffered over the centuries.

Although discredited in biology, Lamarckian evolution has proven to be a powerful concept within artificial evolution applications on the computer. Unlike life in the natural world, computer programs use very simple transformations between genotypes and phenotypes, and the inversion of phenotypes to their corresponding genotypes is often tractable. In the case that genotypes are their own phenotypes, there is no transformation between them whatsoever. The overall implication of this with respect to Lamarckian evolution is that it is possible to optimize a phenotype in the context of a particular problem environment, and have this optimization reflected in the corresponding genotype for subsequent inheritance by offspring. Therefore, the problems encountered by Lamarckism in nature is solved on the computer: inverted communication from the phenotype to the genotype is typically a simple computation.

Lamarckian evolution and genetic search have been combined together in genetic algorithms (GA) [Hart and Belew 1996, Ackley and Littman 1996, Li *et al.* 1996, Grefenstette 1991]. Lamarckism typically takes the form localized search of a phenotype's structure space within the context of the problem being solved. [Hart and Belew 1996] found that the traditional genetic algorithm is most proficient for searching and reconciling widely separated portions of the search space caused by scattered populations, while Lamarckian localized search is more adept at exploring localized areas of the population that would be missed by the wide global swath of the genetic algorithm. Lamarckism may be especially practical when the population has converged to pockets of local minima that would not be thoroughly explored by a standard genetic algorithm. The contribution of Lamarckism is a noticeable acceleration in overall performance of the genetic algorithm.

2. Implementation

2.1 Basic implementation

The appendix contains Prolog code that introduces Lamarckian evolution to a genetic algorithm, and this code is discussed in detail in the remainder of this section. The language is Quintus Prolog 3.2, and the code was implemented in the Silicon Graphics Irix 5.3 environment.

The code presumes the following is done by the main genetic algorithm. Firstly, the individual genotypes are saved as separate Prolog clauses in the following form:

individual(ID, Fitness, Expression).

ID is an identification label such as an integer that uniquely identifies each individual in the population. *Fitness* is the evaluated fitness score for the individual, where the lower scores denote higher fitness. *Expression* is the individual's chromosome as use by the genetic algorithm for the particular problem at hand.

Secondly, two clauses are used to communicate control and population information from the genetic algorithm and user to the Lamarckian module. A clause

population_size(PopSize).

should contain the current population size as its single numeric argument. A clause

lamarckian(Percent, K).

is used to control the Lamarckian evolution itself. *Percent* is a fractional value between 0.0 and 1.0 denoting the percentage of the population upon which Lamarckian evolution should be applied. *K* is the number of iterations used by the search strategy, which is discussed below.

Finally, the user should have the following three predicates defined somewhere in the main genetic algorithm (change as appropriate):

(i) *select(ID) :* This returns a single ID from the population to the calling code. This predicate should use a selection technique of choice, such as tournament selection or fitness-proportional selection.

(ii) *mutation(Expr, Mutant)*: This applies an appropriate mutation on an *Expression*, resulting in its *Mutant.*

(iii) *eval_fitness(Individual, Fitness)*: This applies the problem-specific fitness evaluation function on an *Individual*, resulting in its Fitness value, such that lower values of Fitness denote fitter individuals.

The top-level predicate of the module is *lamarckian_evolution*/0. This predicate should be called within the main generational loop of the genetic algorithm, at a point after the main genetic reproduction has occurred for a generation. For example,

```
loop :- ...
        {perform reproduction and mutation for entire population}
        ...
        % (a new generation has now been formed...)
        lamarckian_evolution,
        loop.   % iterate to next generation
```

Although the code is written to be used in a generational style of genetic algorithm, as opposed to a steady-state genetic algorithm [Mitchell 1996], it is easily adapted to a steady-state approach if desired (see section 2.2).

The first clause of *lamarckian_evolution* takes care of the case when the user wants to perform Lamarckian evolution on the entire population (*Percent* is 1.0). The clause collects the entire set of individual identifiers from the population, and calls the Lamarckian algorithm on them. Otherwise, if a fraction of the population greater than 0.0 is to be processed, the second clause is used. Here, the rough number of individuals to process is computed from the percentage parameter and the population size, and an appropriately sized list of unique ID's is obtained and processed.

The second clause of *lamarckian_evolution* calls the predicate *get_unique_IDs*/3 to select a set of unique identifiers of a desired size between 1 and the size of the population as a whole. To do this, *get_unique_IDs* loops repeatedly until a list of the desired size has been obtained. The selection routine from the main genetic algorithm is used to select individuals. Note that if a high percentage of the population is to be selected for Lamarckian processing, this routine can become slow, since the selection of individuals unique to an already large list of selections can take time. It is recommended that the entire population be processed in such cases, as the first clause of *lamarckian_evolution* will process such selections efficiently. Finally, if the user does not wish to use Lamarckism, the final clause will activate.

The predicate *lamarck_loop*/2 controls the Lamarckian evolution of a list of selected individuals. For each individual as referenced by an ID in the list, its current expression and fitness are obtained from the database. Then a localized search is performed on it, for the number of iterations specified by the parameter *K*. The result of the localized search is a possibly new individual *NewExpr*, with fitness *NewFit*. If the new fitness is better than that of the original, then the original individual's clause in the database is updated with its newly optimized expression and fitness. Otherwise, the original individual is not altered. The calls to the database manipulation routines *retract* and *assert* within *lamarck_loop*/2 perform the actual Lamarckian evolution. The phenotype is explored using the search algorithm, and should a better result be obtained, the corresponding genotype (which is identical to the phenotype in this problem) is revised.

The localized search routine is *hill_climb*/3. It takes the iteration value, the individual and its fitness, and performs a hill-climbing search for a fitter individual. The search presumes that the genetic encoding is complex enough that cycling visits to the same individuals are unlikely. This also presumes that the mutation operator is adequately robust to permit exploration of the entire search space. The first clause of *hill_climb* activates when the iteration count has been depleted, and returns the best individual discovered during the search. The second clause takes the current fittest expression, and then mutates it with the genetic algorithm's mutation predicate. Its fitness is measured with the fitness evaluator, and the fitter of the old and new individuals is used for subsequent hill-climbing.

An alternate hill-climbing predicate, *hill_climb_nocycle*/3, is also given. This version prevents cycling during the search, which can arise in some problem domains in which the number of mutations of a given individual can be less than the desired iteration value. The *hill_climb2* predicate keeps a list of the visited individuals, and does not permit repeated visits to members of this list. In addition, the list is constructed so that the best individual(s) seen are stored at its head. This permits a shunting procedure to be applied when a cycle has been detected, in which the front item, which must be one of the fittest, is moved to the right of its fellow least-valued siblings, if any. *insert_list*/2 places the mutant into the correct position of the list of saved individuals.

2.2 Options and enhancements

The code described in section 2.2 is simple and compact, and is easily specialized and enhanced if necessary. As is discussed later, a determining criterion in applying Lamarckian evolution is its relative computational cost in terms of the genetic algorithm as a whole. The need for alteration will critically depend upon the computational costs associated with the application being investigated.

The predicate *get_unique_IDs* uses the selection routine from the main genetic algorithm. This can be enhanced in a number of ways, for example, by permitting a subset of the most fit individuals to be selected, or even a random set of individuals. Empirical experimentation should be used to determine the best selection strategy for a given problem.

The best-first search procedure is not the only local search algorithm possible [Winston 1992], and others may be applicable depending on the shape of the search space and nature of the mutation operator. In addition, the hill-climber uses mutation to generate new individuals to explore. A possible unorthodox enhancement is to include crossover and other reproduction operators as methods for generating individuals. The frequency of use of these different operators might be specified by the user. Lamarckian evolution can be circumvented if an optimal individual replaces another weaker individual in the population, using an appropriate selection criterion.

The code can be easily adapted for use within a steady-state genetic algorithm. To do this, Lamarckian localized search should be treated as a reproduction operator along with crossover and the others. The simplest means to do this is to call the following on a selected individual ID:

lamarck_loop([ID], K)

This applies best-first search on the selected ID for *K* iterations, and replaces that individual in the database if a better variation is discovered.

3. Example Application

The application of the Lamarckian evolution system on a representative problem is now given. The Traveling Salesman Problem (TSP) is an NP-complete problem that has attracted much attention in the theoretical computer science and genetic algorithm communities. The approach to the TSP taken in this section is not particularly advanced, as it uses an inefficient denotation that has been proven shown to hinder genetic algorithm performance [Tamaki *et al.* 1994] . Rather, the experiment described here is designed to clearly illustrate the overall effect of Lamarckian evolution on a straight-forward problem.

The TSP is defined as follows. A map of cities and paths between them is given. There is a distance associated with each inter-city path. The TSP is to

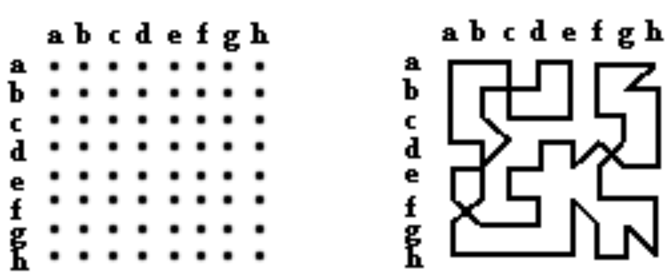

Figure 1: City grid, and one route with distance of 70.14

determine a tour order for the cities such that all the cities are visited, the tour finishes where it began, and the overall tour distance is minimal. In the experiment attempted, we consider 64 cities laying on a square 8 by 8 grid (see Fig 1). The vertical and horizontal distance between neighbour cities is taken to be 1. The cities are maximally connected with one another, making it possible to travel between any two cities on the grid. For convenience, we label each city with the row and column label from the grid, for example, *aa* and *cf*. Table lookup is used to determine the coordinates of a city, and path distances are computed directly from the city's grid coordinates. Routes must explicitly refer to the connected cities: a path between *aa* and *ac* that happens to pass through *ab* is not considered to connect to *ab*.

The TSP denotation chosen for the genetic algorithm is a simple one in which a list of 64 city names representing a particular order of cities in a route:

aa bf fh gh ah ... cc

where *cc* at the end is presumed to connect to *aa*. Each city should reside on the list once and only once. This denotation has ramifications on the crossover operation. During crossover, two random crossover points are determined for two selected paths. The intervening strings between the crossover points are then swapped between the routes. This will often result in illegal routes, as cities may be duplicated and missing after the transfer. Corrective post-processing is therefore performed on the offspring to remove these anomalies, for example, with the OX strategy in [Goldberg 89]. Mutation does not require correction. It simply takes a city and moves it to a random alternate location in the tour, which is correctness-preserving.

Population size:	200
# generations:	150
# runs:	35
Probability of crossover	1.0
Tournament size, reproduction:	2
Tournament size, replacement:	3
Lamarckian evolution	
% population processed:	0.50
# iterations in search (K):	(i) 5, (ii) 20
Selection criteria:	tournament reproduction
Unique population:	yes

Figure 2: Lamarckian experiment parameters

Other parameter settings are in Fig. 2. The genetic algorithm uses a hybrid steady-state approach, in which pseudo-generations are defined for the purposes of applying Lamarckism and collecting statistics. Population changes occur to a single population. After a set number of population changes (taken to be the population size), a generation is said to have passed. At this point, the *lamarckian_evolution*/0 routine is invoked. Tournament selection is performed for both reproduction and replacement of individuals. Since the use of tournament selection tends to result in populations saturated around the fittest individual, the algorithm disallows duplicates in the population.

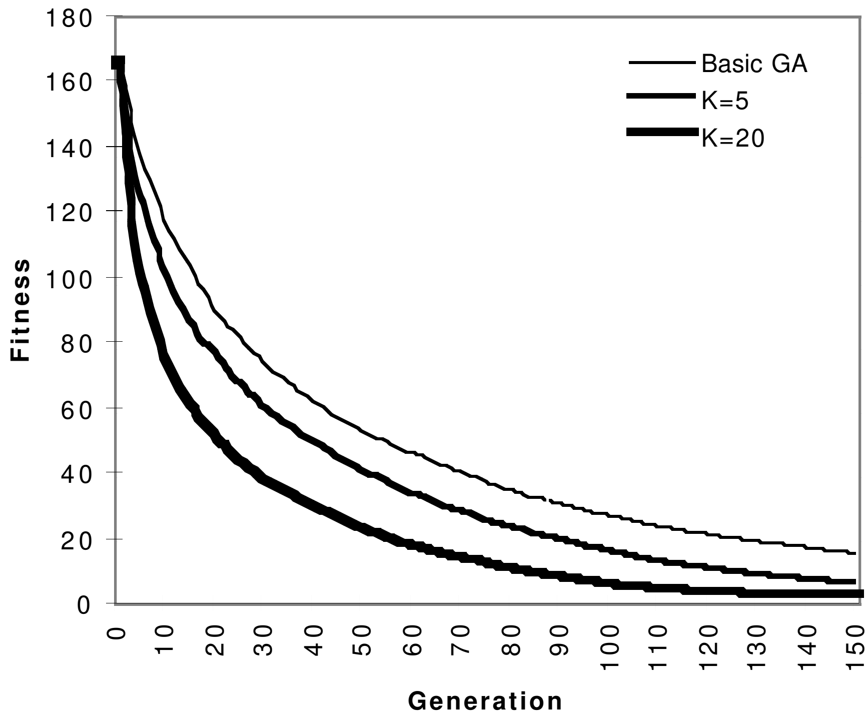

Figure 3: Performance graph (avg. 35 runs)

Fig. 3 is a performance graph of the average best fitness per generation, for 35 runs with Lamarckian evolution (iteration values of 5 and 20), and 35 runs without any Lamarckism. The use of Lamarckism clearly accelerates performance with respect to generational time. The asymptotic nature of the curves shows that the gains become most significant during later generations, since the Lamarckian-influenced populations are concentrated on fitter areas of the search space. As is expected, the run using 20 iterations yields the best results. Even with the seemingly negligible iteration value of 5, however, by generation 100 the Lamarckian runs have a lead of approximately 50 generations over the basic genetic algorithm.

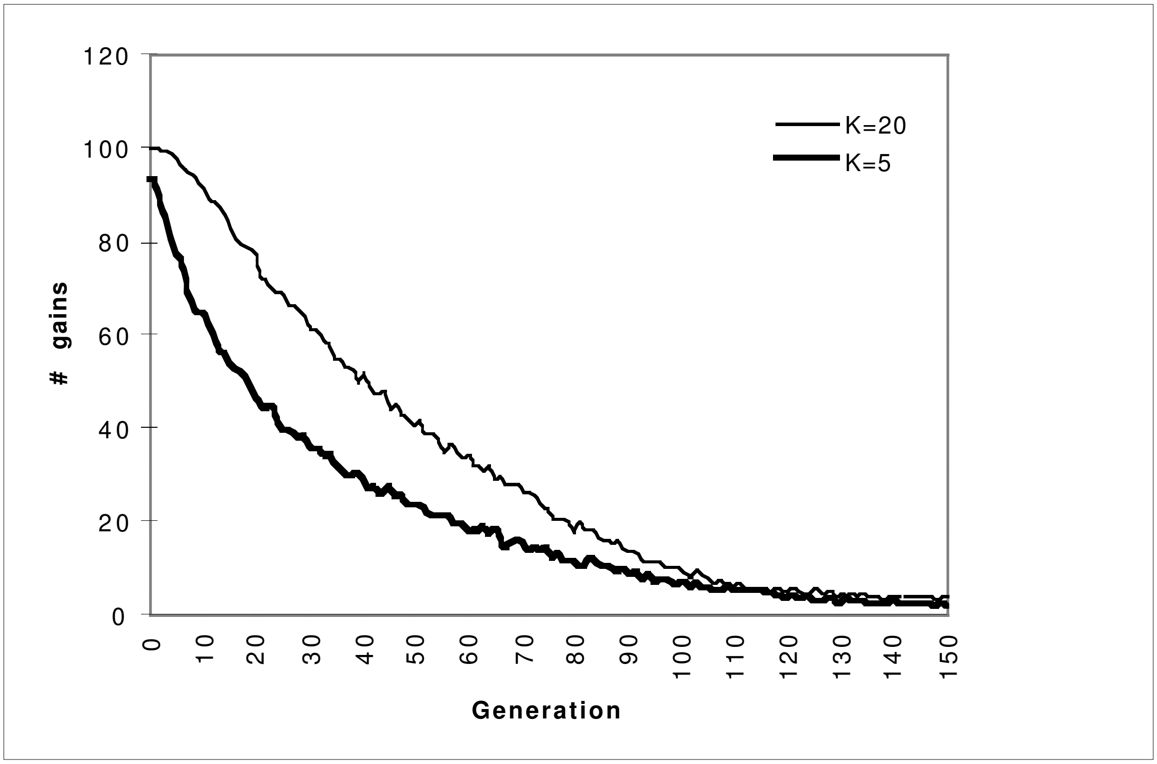

Fig. 4: Optimized individuals

Fig. 4 shows the average number of optimized individuals found with localized search over the run. The 20-iteration search does a better job of finding individuals to optimize for the first 100 or so generations. After 100 generations, it performs only marginally better than the 5-iteration search. This implies that the number of iterations must be increased dramatically if mutation is to make significant progress during later stages of a run.

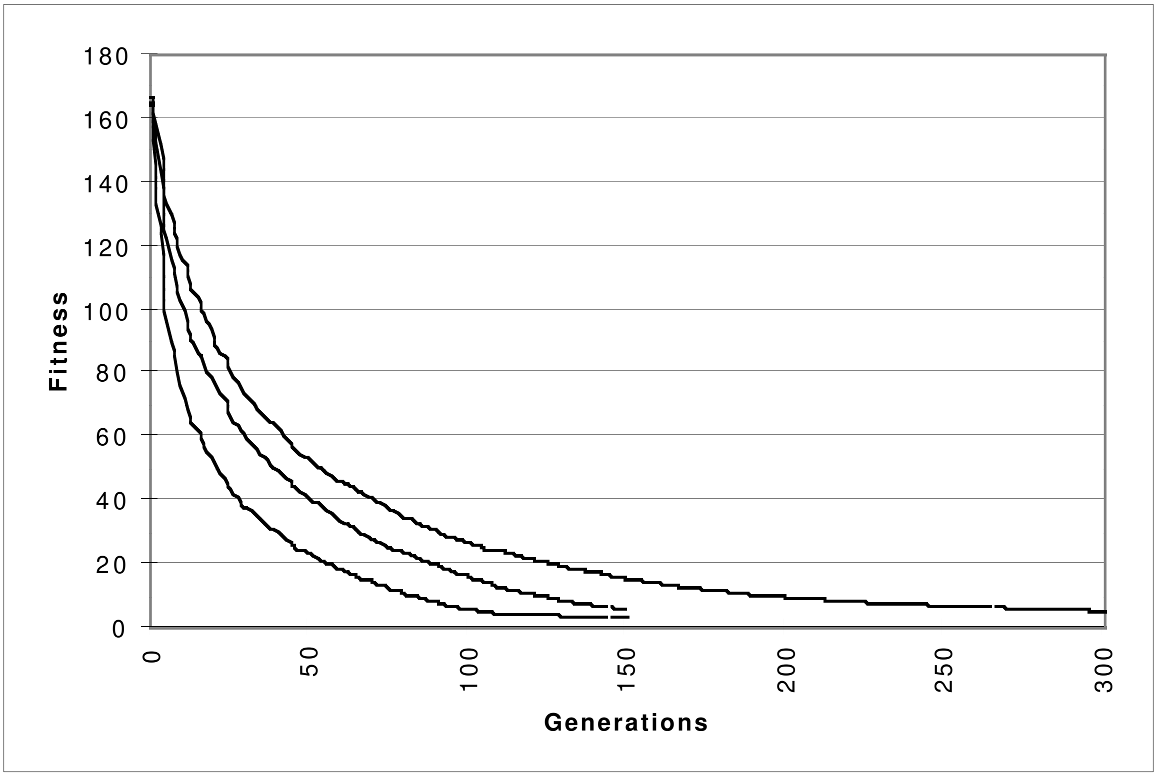

Figure 5: Extended performance graph (avg. 35 runs)

The above performance results must be examined in light of the overall computational overhead used by Lamarckian evolution. In order to further compare the performance of the basic genetic algorithm with the Lamarckian ones, extended runs of the non-Lamarckian case were done to 300 generations (see Fig. 5). In general, the non-Lamarckian runs would need to run to generation 150 before obtaining the K=5 performance at generation 100, and to almost generation 300 to reach the K=20 results at generation 100.

Consider the table in Fig. 6. The Lamarckian runs are much more expensive than the non-Lamarckian in terms of the total number of individuals processed. This measurement would initially indicate that Lamarckism is not an economical enhancement to the overall processing. On the other hand, the total processing time (Silicon Graphics O2 workstation, R5000 180 MHz CPU) is less than proportional to the number of individuals processed. One reason for this is that the basic route encoding used for the TSP is such that crossover is more computationally expensive to perform than mutation. Crossover requires the detection and correction of illegally repeated cities, whereas mutation's swapping of two cities in the list can be done quickly without correction. (See [Tamaki *et al.* 1994] for more advanced encodings of the TSP that circumvent the need for correction after crossover.) In addition, the cost of fitness evaluation in this experiment is negligible. Therefore, the time overhead of using Lamarckism was not proportional to

Experiment	Generations	Individuals processed	Avg. CPU time
K=5	150	105,000	00:07:52
K=20	150	330,000	00:12:38
Basic GA	150	30,000	00:06:01
Basic GA	300	60,000	00:12:18

Figure 6: Individuals processed and CPU processing time

the number of individuals processed, and the processing benefits from it, especially after later generations have lapsed.

4. Discussion

The practicality of Lamarckian evolution within a genetic algorithm depends upon the problem being investigated. Lamarckian localized optimization will typically boost the relative fitness of the population, and hence accelerate search performance of the parent genetic algorithm. This acceleration, however, can be costly if the fitness evaluation of phenotypes during localized searches is computationally expensive. Thus the benefits of Lamarckism must be weighed with the costs inherent in the problem being studied.

The performance of Lamarckian evolution can vary during a run. Further research is needed to determine strategies for dynamically adapting Lamarckism during execution. For example, the iteration values and percentage of the population processed by Lamarckism could automatically adapt to the performance of the run. Perhaps the most promising approach is to adaptively evolve various control parameters of Lamarckian learning, in the style of [Grefenstette 1986, Back 1992]. The code described in this chapter may be a stepping-stone for more sophisticated enhancements in this direction.

One other issue is the nature of mutation operator used by a Lamarckian strategy. The minimal requirement is that the mutation used is robust enough to permit exploration of the entire search space. The experience with the TSP experiment is that the simple mutation operator had difficulty making optimizations during later generations. The population was already relatively fit, and random mutation of very fit phenotypes resulted in a blind search. Processing may benefit, therefore, with the use of a library of varied mutation operators, as well as more intelligent next-state operators for generating new individuals.

Bibliography

Ackley, D.H. and Littman, M.L. 1994. "A Case for Lamarckian Evolution". *Artificial Life III*, Ed. C.G. Langton, Addison-Wesley.

Back, T. 1992. “Self-Adaptation in Genetic Algorithms”. Proc. 1st ECAI, Dec 1991, MIT Press.

Cochrane, E. 1997. "Viva Lamarck: A Brief History of the Inheritance of Acquired Characteristics". [http://www.ames.net/aeon/].

Dawkins, R. 1996. *The Blind Watchmaker*. Norton.

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

Gould, S.J. 1980. *The Panda's Thumb*. New York: Norton.

Grefenstette, J. J. 1986. “Optimization of control parameters for genetic algorithms”. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1):122-128.

Grefenstette, J.J. 1991. "Lamarckian Learning in Multi-agent Environments". *Proc. 4th Intl. Conference on Genetic Algorithms*, Morgan Kaufman.

Hart, W.E. and Belew, R.K. 1996. "Optimization with Genetic Algorithm Hybrids that Use Local Search". In *Adaptive Individuals in Evolving Populations*. Ed. R.K. Belew and M. Mitchell, Addison-Wesley.

Li, Y., Tan, K.C., and Gong, M. 1996. "Model Reduction in Control Systems by Means of Global Structure Evolution and Local Parameter Learning". *Evolutionary Algorithms in Engineering Applications*, ed. D. Dasgupta and Z. Michalewicz, Springer Verlag.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press.

Tamaki, H., Kita, H., Shimizu, N., Maekawa, K., and Nishikawa, Y. 1994. "A Comparison Study of Genetic Codings for the Traveling Salesman Problem". *1st IEEE Conference on Evolutionary Computation*, June 1994.

Winston, P.H. 1992. *Artificial Intelligence (3e)*. Addison-Wesley.

Appendix: Code listing

```
% Author: Brian J Ross
% Dept. of Computer Science, Brock University, St Catharines, ON, Canada
% July 1997

%  lamarckian_evolution
%
% Performs Lamarckian evolution on P% of population, iterating
% each K times using hill-climbing.
% lamarckian(P, K) is set by user. Population exists as facts in
% database of form: individual(ID, Fitness, Expression).
% Fitness is standardized (0 is best, higher values are less fit).
% An improved individual, if found, is replaced in the program database.
% First clause efficiently processes entire population.
% Second case is if less than entire population to be used, in which case
% selection must be performed.

lamarckian_evolution :-
        lamarckian(Percent, K),
        Percent >= 1.0,
        setof(ID, F^E^individual(ID, F, E), IDs),
        write('Lamarckian evolution...'), nl,
        lamarck_loop(IDs, K),
        !.
lamarckian_evolution :-
        lamarckian(Percent, K),
        Percent > 0.0,
        Percent < 1.0,
        population_size(PopSize),
        N is integer(Percent * PopSize),
        write('Lamarckian evolution...'), nl,
        get_unique_IDs(N, [], IDs),
        lamarck_loop(IDs, K),
        !.
lamarckian_evolution.

% get_unique_IDs(N, SoFar, IDs)
%       N - Number of unique individual ID's to collect
%       SoFar - ID's collected so far
%       IDs - final set of unique IDs
%
% Retrieves a list of N unique individual ID's,
% selecting each one via a tournament selection routine defined elsewhere.

get_unique_IDs(0, IDs, IDs) :- !.
```

```
get_unique_IDs(N, SoFar, IDs) :-
        repeat,
        select(ID),
        \+ member(ID, SoFar),
        M is N - 1,
        get_unique_IDs(M, [ID|SoFar], IDs),
        !.

% lamark_loop(IDs, K)
%       IDs - individuals to perform Lamarckian evolution upon
%       K - number of iterations for best-first search
%
% The individuals in IDs have hill-climbing performed on them.
% If the result of this search is a fitter individual, then that new
% individual replaces the original in the database. A side effect is that
% individual/3 clauses are replaced with improved individuals when found.
% The + and - characters written to screen to give overview of
% Lamarckian performance.

lamarck_loop([ ], _) :- !.
lamarck_loop([ID|Rest], K) :-
        individual(ID, Fit, Expr),
        hill_climb(K, (Fit, Expr), (NewFit, NewExpr)),
        % replace previous line with following if cycling is a problem
        % hill_climb_nocycle(K, (Fit, Expr), (NewFit, NewExpr)),
        (NewFit >= Fit ->
                write('-')
                ;
                write('+'),
                retract(individual(ID, _, _)),
                assert(individual(ID, NewFit, NewExpr))),
        lamarck_loop(Rest, K),
        !.

% hill_climb(K, Item, Soln)
%       K - iterations to perform on individual for search
%       Item - individual to perform localized search upon
%       Soln - The most improved solution found. Worst case is an expression
%               having same fitness as original.
%
% Does hill-climbing search for K iterations on an individual.
% Expression is mutated, and if mutant is fitter than individual, it is the new
% expression to mutate. Otherwise, the original will be used again.

hill_climb(0, Item, Item) :- !.
hill_climb(K, (TopFit, TopExpr), Soln) :-
```

```
        mutation(TopExpr, NewExpr),
        eval_fitness(NewExpr, NewFit),
        select_best((NewFit, NewExpr), (TopFit, TopExpr),  BestSoFar),
        K2 is K - 1,
        hill_climb(K2, BestSoFar, Soln),
        !.

% select_best(Item1, Item2, BestItem)
%
% BestItem is the one of the Items with the better (lower) fitness value.

select_best((F1, E1), (F2, _), (F1, E1)) :- F1 =< F2, !.
select_best(_, X, X).

% hill_climb_nocycle(K, Item, Soln)
% hill_climb2(K, Items, Soln)
%       K - iterations to perform on individual for search
%       Item - individual to perform localized search upon
%       Items -list of individuals searched thus far; first items are the fittest
%       Soln - The most improved solution found. Worst case is an expression
%               having same fitness as original.
%
% An alternate hill-climbing predicate. This version is useful if the problem
% domain is such that cycling (repeated visits) during the search is likely.
% It disallows cycling, and permits fairer  search from the best individuals.

hill_climb_nocycle(K, Item, Soln) :-
        hill_climb2(K, [Item], Soln),
        !.

hill_climb2(0,[Item|_], Item) :- !.
hill_climb2(K, [(TopFit, TopExpr)|Rest], Soln) :-
        mutation(TopExpr, NewExpr),
        \+ member((_, NewExpr), Rest),
        eval_fitness(NewExpr, NewFit),
        insert_list((NewFit, NewExpr), [(TopFit, TopExpr)|Rest],  List2),
        K2 is K - 1,
        hill_climb2(K2, List2, Soln),
        !.
hill_climb2(K, List, Soln) :-
        K2 is K - 1,
        shunt(List, List2),
        hill_climb2(K2, List2, Soln).

% insert_list(Item, List, NewList)
%       Item - Expression/Fitness to add to the list
```

```
%       List - list of Expr/Fit pairs found so far in search
%       NewList - List with Item appropriately inserted
%
% Item is placed into List, such that the List retains all the fittest members at
% the front. If Item is a fittest individual, it will go at the front. Otherwise it
% is placed behind them, and before an item in the list that is less-fit than
% itself (convenient to code - could be optimized if desired).

insert_list(Item, [ ], [Item]).
insert_list((Fit, Expr), [(Fit2, Expr2)|Rest], [(Fit, Expr), (Fit2, Expr2)|Rest]) :-
        Fit =< Fit2.
insert_list((Fit, Expr), [(Fit2, Expr2)|Rest], [(Fit2, Expr2)|Rest2]) :-
        Fit > Fit2,
        insert_list((Fit, Expr), Rest, Rest2).

% shunt(Items, NewItems)
%       Items - ordered list
%       NewItems - Items, but with first item shunted to the right-most
%               end of its sibling expressions of same fitness
%
% Optional utility. To decrease chance of thrashing on a poor expression
% in list, shunt the best individual at the start of the list to the far-right of
% equally best-valued siblings (if any).

shunt([(Fit, Expr1), (Fit, Expr2)|Rest], [(Fit, Expr2)|Rest2]) :-
        !,
        shunt([(Fit, Expr1)|Rest], Rest2).
shunt([A, B|Rest], [A, B|Rest]):- !.
shunt(List, List).

% etc...
member(A, [A|_]).
member(A, [_|B]) :- member(A, B).
```