Developing a Generic Genetic Algorithm

Melvin Neville
Northern ArizonaUniversity
College of Engineering & Technology
Flagstaff, AZ 86004
1-928-523-4613

Melvin.Neville@nau.edu

Anaika Sibley
Northern Arizona University
College of Engineering & Technology
Flagstaff, AZ 86004
1-928-523-2206

abs5@dana.ucc.nau.edu

ABSTRACT

Genetic and evolutionary algorithms, inspired by biological processes, provide a technique for programs to "automatically" improve their parameters. We discuss the basics of the algorithms and introduce our own hybrid. The development of this hybrid and its application to a simplified problem, evolving the coefficients for the sine function in a Taylor series, presents opportunities for computer science education with respect to model-building, data structures, and language features. Students must decide upon the representation of the chief mechanisms of genetic algorithms: mutation to alter the values of parameters directly and crossover to vary the groupings of co-evolved parameters in order to break away from local fitness maxima. They must examine the meaning of fitness itself as well as make many other modeling decisions. Ada itself provides both challenges and advantages: linked-lists must be well understood to be updated in an object-oriented context and hard-typing produces mixed reactions in students used to C++, but generics provide a powerful way to generalize the algorithm and incorporating different problem domains,.

Categories and Subject Descriptors

D.3.3. [**Programming Languages**] Language Constructs and Features – *data types and structures, inheritance, modules, packages;* I.2.2. [**Artificial Intelligence**]: Automatic Programming – *program modification.*

General Terms

Algorithms, Design, Languages.

Keywords

Genetic algorithm, evolutionary algorithm, generics, templates, teaching, Ada education, data structures, artificial intelligence, software tools.

1.   INTRODUCTION

This paper comes out of the serendipity of a research project conducted by the first author and a course in the Fall 2002 semester which he is teaching; the second author is a student researcher in the project and a student in the course. The research involves the modeling of biological neural networks in relation to the evolution of intelligence and especially learning ability [6]. In order to refine the values of the parameters used in the modeling, we decided to develop a genetic algorithm (GA) engine that could be used with variants of the developing simulation. In order to understand the GA better, we first developed it in connection with a simplified problem. During this development it became apparent that the development process itself offered excellent opportunities for teaching computer science principles in general and the usefulness of Ada in particular.

The Fall 2002 course (CSE 470, "Introduction to Intelligent Systems"), introduces advanced undergraduates and graduate students to non-symbolic artificial intelligence (AI) through the three course segments: neural nets, fuzzy logic, and genetic algorithms. These approaches are all inspired by biological phenomena. This inspiration extends to the desire to "automatically" refine the parameters of an executable program, i.e., have a process improve those parameters rather than depend upon specifying them correctly at the beginning. For example, a Hebbian or a Perceptron neural net can be used to recognize patterns after it has been trained [3]. This training is the "automatic" adjustment of the weights within the net to produce net-output match to a target output; the properties of the net are such that it can then generalize in its classifications. Similarly, an evolutionary or genetic algorithm can be used to alter the parameters used by a program so that it executes better. Again, the process is "automatic" in that the training algorithm causes the parameters to improve without explicit intervention by the programmer. The appropriate implementation of the algorithm, however, involves knowledge of the problem domain.

One of the higher-level purposes of the course is to expose the students to software tool construction: in this regard, the programming they do in the three segments of the course provides generalized experience in tool building that goes beyond what they would receive from using off-the-shelf software that provides the functionality of these particular methods [4]. The simplified GA model that was developed for the learning research is used as a framework for the GA portion of the course, with the students guided to and through the important concepts.

As the course is currently being taught and the GA portion of the course is the last, the presented paper will necessarily represent in part an update based on the course experiences.

2. GENETIC AND EVOLUTIONARY ALGORITHMS

2.1 The Basic Genetic Algorithm

GAs were introduced by John Holland in the early 1970s [8]. The biologically inspiring concepts are the *mutation* of Mendelian genetics and the phenomenon of *crossover*: these produce individual variation which is then acted upon by an analog to evolutionary theory's *natural selection*. Computer scientists have envied the ability of species to evolve towards more complex, higher-functioning forms without (from the standpoint of evolutionary theory) the necessity of invoking an outside supervisor. The fundamental ideas in evolutionary theory and their mapping to a GA (or Evolutionary Algorithm = EA) involve the following:

(1) A living organism is provided with *genes* that help determine its development. A particular gene has its own responsibilities; however, it generally interacts with other genes. The *organisms* (units of selection) in a GA can be represented by a vector that bears the parameters of interest to a solution.

(2) The genes are inheritable and hence genetic variation can be passed on from one generation to the next. The varying forms of the same gene are termed *alleles*. *Mutation* is a process whereby a gene may alter its *allelic* form Similarly, the values of the parameters on the vector of a GA organism can vary according to rules that the designer must stipulate, and the meaning of mutation, as the main introducer of allelic variance, must be decided.

(3) The genes are organized on *chromosomes* so that groupings of particular allelic conformations may remain intact from one generation to the next: reproduction involves passing copies of chromosomes form parent to child. A GA vector represents a chromosome.

(4) *Crossover* occurs during meiosis, the production of a gamete for sexually-reproducing species: portions of homologous pairs of chromosomes may interchange, thereby altering which constellations of alleles may be interacting with which on a single chromosome. (See Figure 1: homologous pairs of chromosomes, with the partial exception of the XY pair, bear the same gene sequences but in potentially different allelic forms.) The crossover operator in a GA has the function of enabling different groupings of parameters to come into contact with each other in possible offspring.

(5) Biological *evolution* is driven through the *genotype* of the individual, the individual's particular alleles and their interactions being displayed to the world through its *phenotype*. Individual phenotypes will have varying advantages in a given environment, and these advantages will, through *natural selection*, translate into better chances at survival and reproduction. In a GA the designer crafts a *fitness function* to measure relative

advantages to the individuals who have been subjected to mutation, crossover, and possibly other operations, and who are competing to constitute the next generation's population.
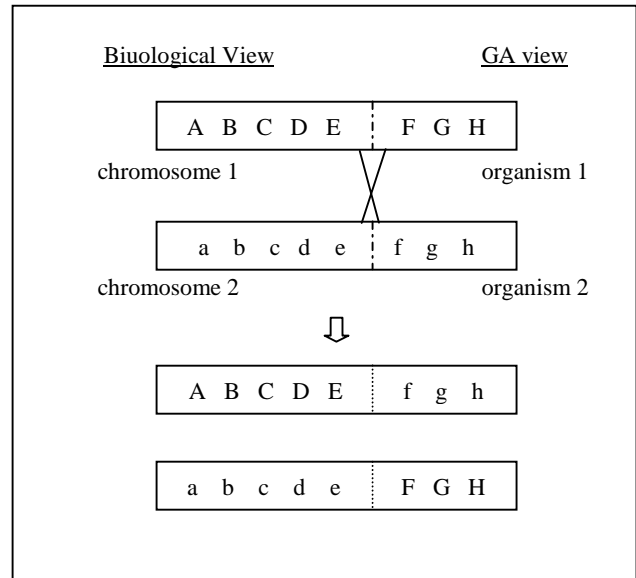

Figure 1. Biology inspires Computer Science: The reassortment of allele blocks on chromosomes after crossover (biology) and of parameter values after crossover (GA).

2.2 Genetic vs. Evolutionary Algorithms

Our algorithm is closer to an EA than a classic GA. Michalewicz [5] describes the essential differences between GAs and Evolutionary Strategies (ES, use here for EA) as being five-fold: (1) The classical GA uses a bit vector to represent the parameters of an individual, whereas an ES uses a floating-point vector. (2) A GA uses repetitive random selection with repetition to draw the next population out of the current population, with the chances being affected by a fitness evaluator (or by rank), while an ES draws the next population out of a larger number of offspring from the previous generation by using the fitness function without replacement. (3) The operations of mutation and crossover may be applied in a GA according to their separate probabilities after the next generation has been selected. In an ES the offspring already contain individuals subjected to mutation and crossover at the time that selection back to the base population size occurs. (4) In a GA the probabilities of mutation and crossover are constant, while in an ES they can alter from generation to generation. (5) A GA handles "illegal" offspring (those with organism parameters outside constraints) by penalizing them in the selection process, while an ES can discard them. In points (1) .. (3) we are closer to an ES, in (4) to a GA, and with respect to (5), in the simplified problem we only produce legitimate offspring. However, the term "Genetic Algorithm" also can imply the generalized evolutionary approach, and so we use it.

3. OUR ALGORITHM

3.1 A Comparison of Algorithms

Sipper presents a canonical form for the GA [7]:

1. Initialize the generation counter to 0;
2. Initialize the populations of entities (i.e., create them with an initial genetic makeup);
3. Evaluate the entities (apply a fitness function);
4. Iterate through generations until some criterion for stopping is met:
    4.1. Increment the generation counter;
    4.2. Choose the new generation from among the members of the last generation;
    4.3. Carry out crossover on the new generation;
    4.4. Apply mutation to the new generation;
    4.5. Evaluate the new generation;
5. End iteration;

The choice of the new generation is typically based on a fitness-influenced, but not fitness-determined, evaluation of the children; thus there is a possibility for a lower-ranking individual to contribute to the next generation. The crossover mechanism and the selection mechanism are both ways of avoiding being trapped short of a really good solution by local fitness maxima.

Our algorithm differs in a number of respects:

1. Initialize the generation counter to 0;
2. Initialize the population of entities;
3. Iterate through generations until some criterion for stopping is met;
    3.1. Increment the generation counter;
    3.2. Each individual produces a standard number of copies ("children");
    3.3. Each child is subjected to potential mutation;
    3.4. Each child has the possibility of crossover (multiple crossover disallowed);
    3.5. The children are evaluated;
    3.6. The most fit are selected for the next generation;
4. End iteration;

The principal differences between the two algorithms are that ours evaluates and selects the next generation after mutation and crossover have been applied, inverts the order of mutation and crossover, and selects the next generation strictly on the basis of the fitness scores. We feel that on the first two issues our approach is at least as effective and is more related to biology; furthermore, crossover serves to dislodge a run from local maxima and therefore non-fitness-based selection is not necessary.

3.2 The Architecture of Our Approach

There are a number of important constants and data structures involved in the program, which can be described in relation to the algorithm (Figure 2):

(1) **NumWin**: the number of Winners per cycle, i.e., the number of individuals that shall reproduce.
(2) **ChildPerWin**: the number of children per Winner.
(3) **Winners**: a list to hold the generation that is about to produce offspring. This list is NumWin long. This list is filled and initialized at the beginning of the run (step 2 of the algorithm) and is refilled at the establishment of the parents for the next generation (step 3.6 of the algorithm).
(4) **Children**: a list to hold the offspring that are generated from the Winners list: it will be NumWin *

ChildPerWin long. The individuals in the list are first subjected to the possibility of mutation and then (in random pairings) to the possibility of crossover. After these operations have occurred, all individuals are evaluated by **Evaluate**, the fitness function.
(5) **PriorityQueue**: a data structure that makes up a priority queue into which the Children are transferred and ordered strictly on the basis of their fitness. The size of the priority queue is the same as that of the Children list in our implementation; a variant has it the size of the Winners list.
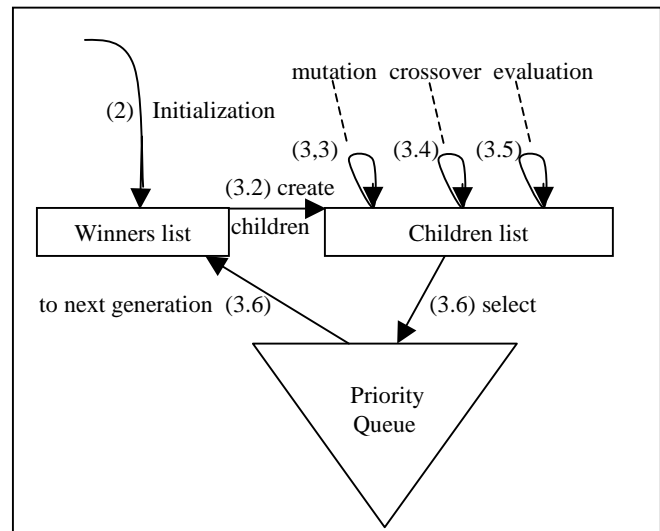

F**igure 2. The algorithmic cycle and the principal data structures (numbers refer to our algorithm above).**

The cycle completes with the fittest NumWin individuals being drawn from the PriorityQueue and placed in the Winners list.

4. SELECTING A GA PROBLEM

Our consideration here is with respect to the pedagogical advantages of a particular problem. The problem should be simple to describe and model, with parameters that give scope to the mutation and crossover processes, but also generating interesting modeling questions. In the following we consider a spectrum of these modeling questions, ranging from language issues to the process of modeling itself. The confronting of the questions is a key part of the pedagogy. We include statements of the decisions made for our research model, but the students are stimulated to consider alternatives.

4.1 The Target Problem: Optimizing a Sine Function

To explore the GA process we decided upon using the Taylor series for the calculation of sine values. This is a simple mathematical model with a number of coefficients of varying importance to the accuracy of the model. The Taylor series for the sine is [2]:

$$\sin(x) = x^1/1! - x^3/3! + x^5/5! - x^7/7! \dots \; (x = \text{angle in radians})$$

We set the problem of evolving the coefficients α_i in of a power series in x:

$$\sin(x) = \alpha_0 x^0 /0! + \alpha_1 x^1 /1 + \alpha_2 x^2 /2! + \alpha_3 x^3 /3! + \alpha_4 x^4 /4! + \dots$$

Ideally, in the GA-evolved series every coefficient of an even power of x would be essentially 0.

Interesting questions arise about how big the series should be, what the initial coefficient values should be, and how the fitness of a particular set of coefficients should be measured. For example, in playing with the breadth of the range of powers students should discover the limitations of the precision of Float and even Long Float. We found ourselves in the research forced to use Long Float and to limit the exponent range to 0 .. 12. In setting the initial coefficient values, the expectation that coefficients will evolve approximately equally positively as well as negatively led us to select 0.0 for all values. The development of a good evaluation function is non-trivial: it should compute rapidly, test the whole range of relevant angle values, and force accuracy throughout the range. For example, our function ran from 0 degrees through 90 degrees by 5-degree intervals and summed the square of the error (difference between the tested organism's evaluation and that of a perfect sine series evaluation) for all intervals in the range. Its major fault is in not emphasizing more errors at the low end of the range of angles. We made the evaluation function one of the generic parameters to the genetic algorithm in order to be able to facilitate substituting for it.

4.2 The Interpretation of Mutation

Mutation obviously implies the altering of coefficient values. However, should mutational effects be the same for all coefficients? In the research we decided to set the mutational amount for a coefficient roughly proportional to the amount that we knew was the desired outcome. It is clear that this takes advantage of special knowledge and will not be generally possible.

The frequency of the mutation and the possibility for multiple mutations in the same organism are further questions. We made multiple mutations allowable and expressed the frequency of mutation as a parameter to the algorithm.

4.3 The Interpretation of Crossover

From the standpoint of a GA, the purpose of crossover is to allow the possibility that favorable constellations of organism-parameters may be brought together from different organisms. The analogy to biology is very stretched, so there is room for diverse interpretations. Our particular approach was to use a crossover probability algorithm parameter to randomly select which organisms among the Children could undergo crossover, randomly pair these (if an odd number, deselecting one individual randomly), and then use another parameter to randomly choose a break-point in the range of coefficient powers for swapping between the two individuals.

5. ADA AND EDUCATIONAL ISSUES

5.1 The Structure of Lists

The lists could have been implemented either through arrays or through linked-lists. We used linked-lists; furthermore, the nodes

of the lists each pointed to the objects of interest rather than containing the objects explicitly, so that in the interest of efficiency the lists and their nodes could exist statically once formed even though a given node might be "empty", while the objects could be easily transferred among nodes.

Arrays would have had the particular advantage of facilitating the accessing of organisms by index. Linked-lists are also more confusing to undergraduates (and hence need more practice), and we discovered in the neural nets segment of the CSE 470 course that putting them in a strictly object-oriented implementation with the fields of the nodes being only accessible through access methods produces particular updating problems (Figure 3a). The problem arose in the access for updating of the Node's Contents field: a *GetContents* function returns a *copy* of the Contents of the Node, so that direct updates to the Content's field only affect the copy. Three solutions were proposed as they were, perhaps surprisingly, not immediately obvious to the students: (1) have the Node's Contents field actually point to the object of interest rather than contain it – access through that pointer gives direct access to the Real_Contents fields (Figure 3b); (2) obtain a copy of the Contents object, update its fields, and write the copy over the original Node's Contents (Figure 3c); and (3) violate pure OO encapsulation principles by making the fields of the Node record public, so that Contents may be operated upon directly through the pointer that iterates along the list's nodes (Figure 3d).
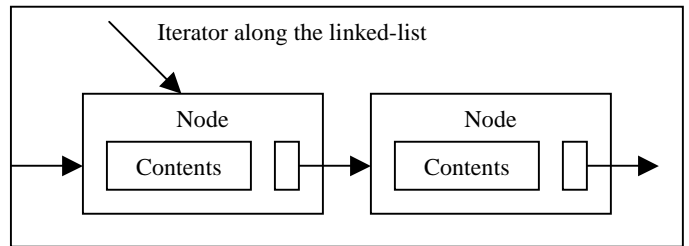

Figure 3a. The Iterator pointer gives direct access to Node but not to Contents (all record fields private), a field inside Node.
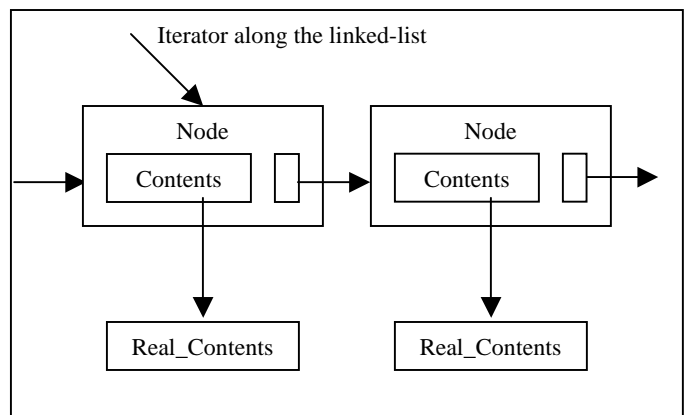

Figure 3b. Direct access to Real_Contents is obtained through the pointer returned by the *GetContents* primitive function of Node.

5.2 The Priority Queue

We implemented this through a binary heap, but a simple, ordered list could also have been used. The trade-off is interesting. In the

simplest case of the heap, the size of the queue must be NumWin * ChildPerWin long in order to sort the children; on the other hand, only NumWin organisms need be drawn from it. The students could be asked to formulate efficiencies such as a heap of size only NumWin, with insertions that occur after the heap has been filled either being logically excluded by the poor fitness of the candidate or by pushing a poorer-fitness candidate out. The list alternative is easier to program: its length need only be NumWin long, and it is formed through insertion-sort, with too-large values (if low values are considered better) simply disregarded or pushed off the end of the list.
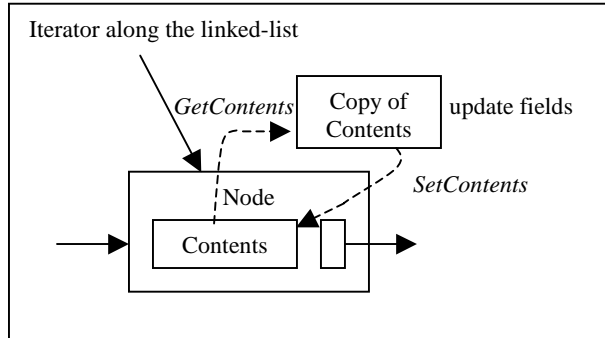
Figure 3c. *GetContents* **on the Node returns a copy of the Contents; this can be updated with respect to its fields and then the copy rewritten to the Node through** *SetContents***.**
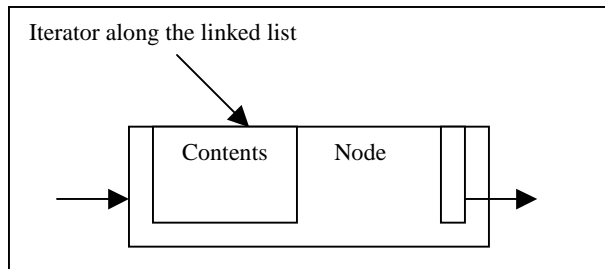
Figure 3d. Direct access to Contents for updating its fields is through the iterator pointer as the fields of Node are public.

5.3   Hard-Typing

The first author has reached the conclusion that reactions to Ada's hard-typing are partially a result of experience and partially reflect personality. He himself (usually) welcomes the assistance that hard-typing gives, but the advantages of the technique do not necessarily promptly win over new recruits who are more used to the freer compiler environment of C and even C++. However, it is hard to imagine the Ada's complex generics system functioning without a very careful typing control.

5.4   Use of Generics

Ada's generics is a very powerful mechanism for the generalizing of code. To indicate the scope of its involvement in the development of our tool (and to justify the paper's title), we first sketch the interface to the eight generic packages (italicizing the first appearance of package names for ease in reference). We then show the instantiation necessary for the driver of this program, as

specialized by the target problem calculating the coefficients of the Taylor series for the sine function.

(1) A package for the basic nodes in the linked lists, exporting GenListNode and GenListNodeClassPtr, each GenListNode containing a GenItemPtr as well as a pointer to the next node.

generic
    type GenItem is private;
    type GenItemPtr is access all GenItem;
package *GenListNode_Pack* is …

(2) A package for the linked-lists that we use for all program lists, exporting GenList and GenListPtr, and instantiated with a generic node package.

generic
    with package GLNP is new GenListNode_Pack(<>);
package *GenLinkedList_Pack* is …

(3) A package for the nodes which serve as the nodes directly stored both in the binary heap and by the pointers from the nodes in the linked-lists. These serve as intermediates which store the fitness values of the *organisms* (the term we will use for the entities that are to evolve) that are to be subjected to mutation, crossover, and evaluation; they also contain a pointer to the organisms themselves. The package exports GenHeapNode and pointers to GenHeapNode. (see Figure 4)
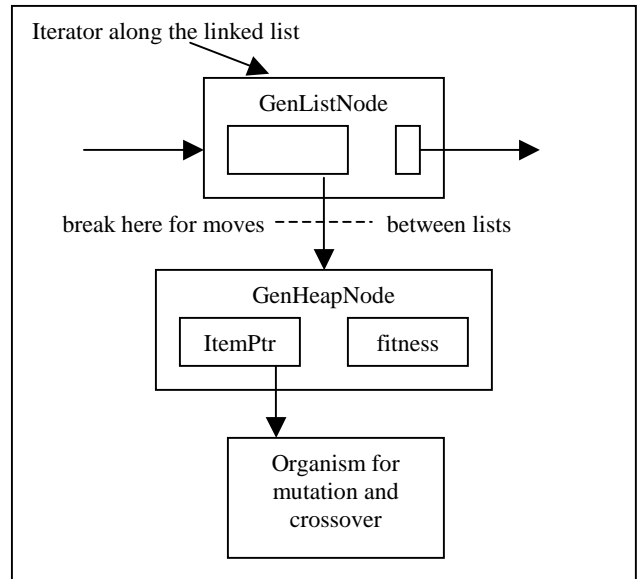
Figure 4. A GenHeapNode and its connection to the GenListNode in the list and to the organism node. Changing an organism from one list to another (or to the priority queue) involves moving the GenHeapNode-Organism grouping as a unit.

generic
    type GenItem is private;
    type GenItemPtr is access all GenItem;
package *GenHeapNode_Pack* is …

(4) A package for the priority queue, which exports GenHeap, GenHeapPtr, HeapArray (the implementation of the binary

heap), and HeapArrayPtr, and which was instantiated with GenHeapNode_Pack.

```
generic
     with package GHNP is new GenHeapNode_Pack(<>);
package GenBinaryHeap_Pack is …
```

(5) A bodyless package to bundle-together types by making them the parameters of this otherwise null package [1]. This bundling assures that the objects representing the organisms to be subjected to mutation are passed as parameters to the genetic algorithm together with the mutation mechanism appropriate for them.

```
generic
     type GenObject is private;
     -- instantiated with the organisms of interest
     type GenObjectPtr is access all GenObject;
     type GenMutator is private;  -- package for mutation
     type GenMutatorPtr is access all GenMutator;
package GenNode_Pack is  end GenNode_Pack;
```

(6) A package to bundle together types and subprograms dealing with crossover and re-export these under new names. This package has no body but eliminates a multitude of potential generic parameters that had formerly made coding the instantiation onerous.

```
generic
     with package GN_Pack is new GenNode_Pack (<>);
              -- bundles types together as in (5) above
     type ConcreteCrossOver is private;
              -- instantiate with the mechanism specialized
              -- for the organisms of interest
     type ConcreteCrossOverPtr is access all
              ConcreteCrossOver;
     with function CrossOverOccursForOrg … return
              Boolean;
              -- mainly debugging
     with procedure DoCrossOver …;
              -- to execute the crossover
package GenericCrossOver_Pack2 is …
```

(7) A package to handle crossover at the level of the list of organisms potentially undergoing the process. This package exports MasterCrossOver and MasterCrossOverPtr.

```
generic
     with package GCOP is new
              GenericCrossOver_Pack2(<>);
              -- instantiate with a concrete crossover
              -- package
     with package GHNP is new GenHeapNode_Pack (…);
     with package GLNP is new GenListNode_Pack (…);
     with package GLLP is new GenLinkedList_Pack (…);
package MasterCrossOver_Pack2 is …
```

(8) The master package for the generic algorithm, exporting GeneticAlg and GeneticAlgPtr. This is the most complex genetic package, taking as it does the packages providing building blocks which must go together to make the master algorithm. The list of the generic parameters is uncomfortably long, but it allows the tailoring of the genetic algorithm to handle many different decisions, which are indicated in the following comments that accompany the parameters.

```
generic
     with package GLNP is new GenListNode_Pack (…);
              -- exports the nodes for lists
     with package GLLP is new GenLinkedList_Pack (…);
              -- exports the lists using the above nodes
     with package GHNP is new GenHeapNode_Pack (…);
              -- exports the fundamental entity for
              -- movement among lists
     with package GBHP is new GenBinaryHeap_Pack (…);
              -- exports the priority queue: for more
              -- generality, should be renamed simply as a
              -- priority queue!
     with package GCOP is new
              GenericCrossOver_Pack2(<>);
              -- has the rules for an organism's crossover
     with package MCOP is new MasterCrossOver_Pack2
              (…);
              -- has the rules for crossover among
              -- organisms
     with function Evaluate (…) return …;
              -- fitness evaluator – so must be
              -- specific to a concrete organism
     with procedure CopyOver (…);
              -- copies an organism – so must be
              -- specific to a concrete organism
     with function MakeMutate (…) return …;
              -- creates the desired mutation object –
              -- so will be working with a concrete
              -- organism
     with procedure DoMutate (…);
              -- carries out mutation – so must be
              -- specific to a concrete organism
package GeneticAlg_Pack is …
```

The process of instantiation of these generic packages follows logical lines but is still complex: we think it worthwhile to have a pattern to follow. The following pattern instantiates for the target problem of the sine function and hence demonstrates the use of a set of target-related packages:

```
     package SinSeries_Pack, exporting SinSeries and
              SinSeriesPtr types;
              -- "SinSeries" from the series of coefficients
              -- for calculating the sine value
     package SinSeriesEvaluate_Pack, exporting fitness
              function Evaluate (decoupling is to allow easy
              substitution of alternate fitness functions);
     package SinSeriesMutate_Pack, exporting Mutate
              and MutatePtr types;
     package SinSeriesCrossOver_Pack, exporting
              CrossOver and CrossOverPtr types
```

This grouping of four package reinforces the need for domain-specific modules and also the usefulness in decoupling these modules.

The sequence of instantiations (and some renamings) is as follows:

(1) Instantiation of the "signature" [1] generic package that bundles together the SinSeries object and the Mutate that goes with it:

```
package SSP renames SinSeries_Pack;
package SSGNP is new GenNode_Pack
  ( GenObjec        => SSP.SinSeries,
    GenObjectPtr    => SSP.SinSeriesPtr,
    GenMutator      => SinSeriesMutate_Pack.Mutate.
    GenMutatorPtr   =>
                SinSeriesMutate_Pack.MutatePtr );
```

(2) Two sequential instantiations to set up the priority queue: the first sets up the nodes of the priority queue (and also the values attached to the nodes of the linked-lists) and the second sets up the priority queue itself, here, a binary heap:

```
subsypte SSPtr is SSP.SinSeriesPtr;
package SSHNP is new GenHeapNode_Pack
  ( GenItem         => SSP.SinSeries;
    GenItemPtr      => SSPtr );
package SSHP is new GenBinaryHeap_Pack
  ( GHNP            => SSHNP );
```

(3) Two sequential instantiations to set up the linked-lists (the first makes use of the above instantiations, and the second sets up the linked-lists):

```
subtype SSHeapNodePtr is SSHNP.GenHeapNodePtr;
package SSLNP is new GenListNode_Pack
  ( GenItem         => SSHNP.GenHeapNode,
    GenItemPtr      => SSHeapNodePtr );
package SSLLP is new GenLinkedList_Pack
  ( GLNP            => SSLNP );
```

(4) Two sequential instantiations to set up the crossover mechanism (the mechanism is more complicated than with mutation because of the interaction among organisms):

```
package SSCP renames SinSeriesCrossOver_Pack;
package SSGCOP is new GenericCrossOver_Pack2
  (      -- package:
    GN_Pack              => SSGNP,
        -- types:
    ConcreteCrossOver    => SSCP.CrossOver,
    ConcreteCrossOverPtr => SSCP.CrossOverPtr,
        -- methods:
    MakeCrossOver        =>
        SSCP.MakeCrossOver,
    CrossOverOccursForOrg =>
        SSCP.CrossOverOccursForOrg,
    DoCrossOver          => SSCP.DoCrossOver );
package SSMCOP is new MasterCrossOver_Pack2
  ( GCOP          => SSGCOP,
    GHNP          => SSHNP,
    GLNP          => SSLNP,
    GLLP          => SSLLP );
```

(5) Finally, the instantiation itself of GeneticAlg_Pack; this instantiation depends upon the previous instantiations:

```
package GAP is new GeneticAlg_Pack
  (      -- packages:
    GCOP          => SSGCOP,
    GHNP          => SSHNP,
    GBHP          => SSHP,
    GLNP          => SSLNP,
    GLLP          => SSLLP,
    MCOP          => SSMCOP,
        -- methods:
    Evaluate      =>
        SinSeriesEvaluate_Pack.Evaluate,
    CopyOver      => SSP.CopyOver,
    MakeMutate    =>
        SinSeriesMutate_Pack.MakeMutate,
    DoMutate      =>
        SinSeriesMutate_Pack.DoMutate );
```

6. CONCLUSION

The closely-related approaches of Genetic and Evolutionary Algorithms offer the promise of a semi-automatic adjustment of relevant programs under paradigms inspired by biological genetics and evolutionary theory. We present the mechanics of a generic genetic-algorithm machine written in Ada in which the use of extensive generic parameters provide flexibility. The object-oriented paradigm itself can raise problems with respect to the implementation of updatable linked-lists; these problems are readily solvable and offer a useful didactic point in themselves. Another issue is Ada's hard-typing: it is rarely popular with students but it is basic to Ada's success.

In order to model how one could apply the machine to a specific problem, we give the generic parameters of the eight generic packages in the program and then trace the instantiation of this machine for the test problem of "evolving" the coefficients of the Taylor series for the sine function: the sequences of instantiations must be carefully choreographed.
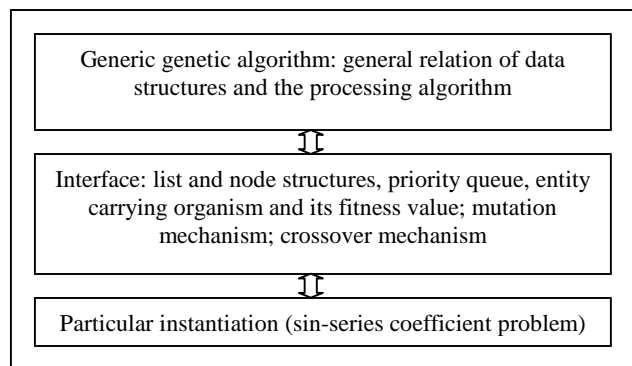
Figure 5. Layers of the Genetic Algorithm: The fundamental data structures and algorithm, the interface, and the actual instantiation.

Figure 5 illustrates the conceptual layering of the solution that results from consideration of the basic questions: what is fundamental to the generic genetic algorithm itself, what should stand as a generic interface between the algorithm and a particular instantiation, and how can likely instantiations be so generalized

that the interface and the genetic algorithm can be implemented in a way that the approach is extensible to many different problems? The consideration of these issues are a related set of significant modeling problems; our particular solution in this case is only one way in which the decisions could have been made.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] Barnes, J. Programming in Ada95. Addison-Wesley, 1998, 2nd ed.

[2] Dwight, H.B. Tables of Integrals and Other Mathematical Data. Macmillan, New York, 1957 (3rd ed).

[3] Fausett, L. Fundamentals of Neural Networks. Prentice-Hall, 1994.

[4] Hines, J.W. Matlab Supplement to Fuzzy and Neural Approaches in Engineering. Wiley, 1997.

[5] Michalewicz, Z. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, Berlin, Heidelberg, New York, 1996 (3rd ed)

[6] Neville, M., and Gray, L. Modeling the evolution of behavioral learning circuitry: *Aplysia* stage, CONIELECOMP 2001 (International Conference on Electronics and Computers), (Cholula, Mexico 2001).

[7] Sipper, M. On the origin of environments by means of natural selection. AI Magazine, 22(2001): 133-140.

[8] Tsoukalas, L. , and Uhrig, R. Fuzzy and Neural Approaches in Engineering. Wiley, 1997.