A Genetic Programming Ecosystem

Judith Devaney, John Hagedorn, Olivier Nicolas, Gagan Garg, Aurelien Samson, Martial Michel
National Institute of Standards and Technology
Gaithersburg MD 20899-8951, USA
{judith.devaney,john.hagedorn,olivier.nicolas,aurelien.samson,martial.michel}@nist.gov

Abstract

Algorithms are needed in every aspect of parallel computing. Genetic Programming is an evolutionary technique for automating the design of algorithms through iterative steps of mutation and crossover operations on an initial population of randomly generated computer programs. This paper describes a novel parallel genetic programming (GP) system inspired by the symbiogenesis model of evolution, wherein new organisms are generated through the absorption of different life-forms in addition to the usual mutation and crossover operations. Different organisms are expressed in this GP system through multiple program representations. Two program representations considered in this paper are the procedural representation (PR) and the tree representation (TR). Populations of these representations evolve separately. Individuals in each population migrate to the other and participate in evolution via representation change algorithms. Parallelism is achieved through use of the AutoMap/AutoLink MPI library. The differences in the locality properties of the representations serve as a source of new ideas for creating the final algorithm.

1. Introduction

The need for algorithms is ubiquitous in computer and computational science. A well designed algorithm can make a difficult problem tractable. However, algorithm design is a person intensive activity. This dependence upon people limits the number of projects that can be attempted in a given time period. Additionally, for specialized disciplines such as parallel algorithms, there is a further dependence on domain knowledge. Thus there is another limitation due to the number of people who have the specialized knowledge to do the design work in the required domain.

It is desirable to automate the design of algorithms, both to increase their number and to search for better ones. Genetic Programming (GP) [8] [9] [10] is a methodology, inspired by Darwin's Theory of Evolution, to evolve algorithms in the form of computer programs from a high level statement of the problem. Because the representation is in the form of a computer program, the output algorithms can be used from one situation to another. At the end of a run, one has a method, not just a point solution. Thus GP can be used to increase the number of algorithms that can be designed by a fixed group of people, as well as to shorten the time to design the algorithms.

However, representation in Machine Learning methods, of which GP is one, is critical. In fact Winston [18] states the following:

Representation Principle:
Once a problem is described using an appropriate representation, the problem is almost solved.

Of course, finding the *appropriate* representation is the critical issue. The whole Machine Learning subfield of constructive induction [3] [19] strives to modify representation spaces to find one that enables a problem to be solved.

When using genetic programming, one can address the representation issue by evolving the representation as the evolution proceeds. Yet this does not remove difficulties such as insufficient diversity and getting stuck in local minima because of high scoring blind alleys.

Views of scientists on the process of biological evolution itself provide a source of ideas to address representation issues. One view of evolution is that it proceeds throught symbiogenesis. According to its chief proponent, Lynn Margulis, "[Symbiogenesis] is a kind ... of Lamarckianism (inheritance of acquired traits)." However, she says "...through symbiogenesis organisms acquire not traits but entire other organisms, and of course, their entire sets of genes... Symbiosis generates novelty... Symbiosis is not a marginal or rare phenomenon. It is natural and common."[12]

We borrow from this view by building a parallel genetic programming environment that attempts to leverage alternative representation spaces to increase the likelihood of finding solutions and to increase performance. Extending the biological analogy of genetic programming, we refer to this approach as a parallel genetic programming *ecosystem*.

This approach creates a set of separately evolving populations each with a distinctly different problem representation along with sub-populations with the same representations. These populations evolve within their isolated environments, but periodically exchange a few highly fit individuals using the *island* model [2] [1] [16]. Because these individuals evolve with different underlying representations and/or with different evolutionary operations, we expect that the populations will evolve along distinctly different lines and that the exchanged individuals will bring information into their new populations that is very different from what was evolved locally.

Note that any property of an evolving population's parameter space may alter the search space or search procedure. Certainly differing program representations will present us with potentially radically different search spaces. But other properties of the population environment's operating parameters such as mutation and crossover rates, population size, or tournament size also can alter the characteristics of the search space.

So we are building an ecosystem of evolving populations. One population may differ from another in its underlying representation of individual programs, or in terms of the evolutionary pressures that are being applied, such as mutation rates and methods. In terms of the biological analogy, the difference in program representation is analogous to a difference in species and the difference in evolutionary pressures is analogous to a difference in the environment (such as temperature or available sources of food).

Of course, when the time comes to exchange individuals between populations a difference in program representations presents us with a substantial problem to overcome. After all, in biology, different species cannot inter-breed; we overcome this by using transformation algorithms to convert between representations, creating opportunities for symbiogenesis.

The rest of the paper is organized as follows: Section 2 gives a brief description of Genetic Programming and then describes the two base representations of this paper, as well as conversion algorithms between them. Section 3 describes the method of parallelization using the AutoMap and AutoLink [5] [7] [13] [14] [15] [17] MPI Data Structure Tools. Finally, section 4 discusses: 1) ways in which the methodology can benefit problem solution, and 2) plans to run the system.

2. Description of System

The GP algorithm consists of the following set of tasks: creation of initial population, evaluation of the fitness of the individuals in the populations by means of a *fitness function*, and then creation of the next generation through a set of evolution operators (such as mutation, reproduction, and crossover). The evaluation and generation stages are iterated until the problem, as defined by the fitness function, is solved, or some limit (such as the maximum allowed number of generations) is reached. This is shown in figure 1.

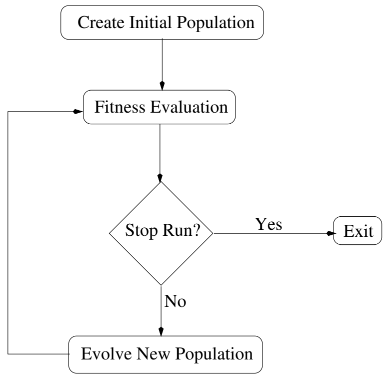

Figure 1. Simplified Algorithm for Genetic Programming

The GP methodology was originally implemented in LISP [8] where the use of S-expressions for both data and functions provides a certain simplicity of implementation. It is now implemented in a variety of languages. The programmer decides how much, if any, structure is to be imposed on the individual programs; this structure may be fixed or evolve during a run. Usually the program is a tree with multiple branches that perform different functions such as function definition, iteration, and result production. Individual trees are usually implemented as fixed length strings in postfix notation [1]. However, they may also be variable length trees implemented with pointers [4]. We chose two representations for our multi representation system: a procedural representation (PR) close to what a programmer would write, and a tree based representation (TR) implemented with pointers for maximum flexibility.

2.1. Procedural Representation (PR)

A novel feature of this work is the PR representation. This representation is modeled on conventional procedural programming languages such as C or Fortran. The program is structured into routines that make calls to other routines. Data is conveyed by means of argument lists and local variables may be used to hold intermediate results. At the bottom of the calling hierarchy are calls to built-in routines that perform basic operations such as add and subtract.

This PR was adopted for several reasons. First, since this sort of program structure is useful to human programmers, we thought it might prove to be effective for computer generated programs. Furthermore, the generated programs might be somewhat simpler for a human reader to understand. Finally, the different characteristics of the search space might provide an environment in which certain types of problems may be more easily solved.

Here are some of the salient features of this representation:

- There are two types of routines:
  - *Composite* routines call other routines
  - *Atomic* routines do not call other routines; these provide the basic operations of the GP system (such as addition, subtraction, etc.).
- Each routine has a formal argument list with arguments identified as input, output, or input/output. Use of these formal arguments always honor these I/O attributes.
- Each formal argument may have a specified data type or the data type may be left unspecified in which case at run-time its data type is specified by the data type of the actual argument that is passed to the routine for that formal argument.
- Each routine may have local transient variables that are scoped only within a single invocation of that routine. Local variables acquire a data type only at run-time, when the variable is created with the same type as an incoming argument.
- Each composite routine calls a sequence of other routines. Each call must specify an actual argument list that corresponds to the formal argument list of the called routine. Each actual argument is either a formal argument in the calling routine, a local variable of the calling routine, or a constant.
- Data type conversions are performed automatically whenever necessary and feasible.
- User-supplied code can be incorporated into the system in the form of additional atomic routines.

Here is a text representation of a simple program. It is presented in a C-like syntax, but it is important to remember that this is not intended to be compilable C code. Note that formal arguments and local variables are declared $void*$. This indicates that the actual data types of these items are determined only at run-time. Some comments have been added to clarify the program.

```
void PN0001 (
  void * arg000 ,   /* IN      */
  void * arg001 ,   /* IN      */
  void * arg002 ,   /* IN      */
  void * arg003     /*     OUT */
  )

  {
  void * lv000 ;   /* Like arg 2 */
  /* end of local variable list */

  add (arg000, arg001, arg003);
  mult (arg000, arg003, arg001);
  PN0002(arg001,arg003,arg002,arg003,lv000);

  }  /* end of PN0001 */


void PN0002 (
  void * arg000 ,   /* IN      */
  void * arg001 ,   /* IN      */
  void * arg002 ,   /* IN      */
  void * arg003 ,   /*     OUT */
  void * arg004     /*     OUT */
  )

  {
   /* end of local variable list */

  sub (arg000, arg001, arg004);
  div (arg001, arg002, arg003);
  add (arg003, 3.900000, arg004);

  }  /* end of PN0002 */
```

Clearly this program representation presents a variety of issues within the context of a genetic programming system. The most important issues occurs during crossover. For crossover, a branch of the calling hierarchy is easily selected, but when branches are moved from one program to another, the formal argument lists of the removed branch and the inserted branch may differ. These differences must be reconciled for the resulting program to be valid. The crossover procedure adapts the actual argument list of the removed routine to conform to the formal argument list of the newly inserted routine.

2.2. Tree Representation (TR)

In TR a program is viewed as a tree structure. A tree is composed of nodes with children (operators) and leaves (childless nodes). The children of a node are the operands of the operator at that node. A leaf node represents a value such as a constant or a variable. For example, the following program: $(5 + 3) * (8 - 4)$ will be seen as the tree shown in figure 2.

This representation has many advantages among which are:

- Ease and speed of implementation

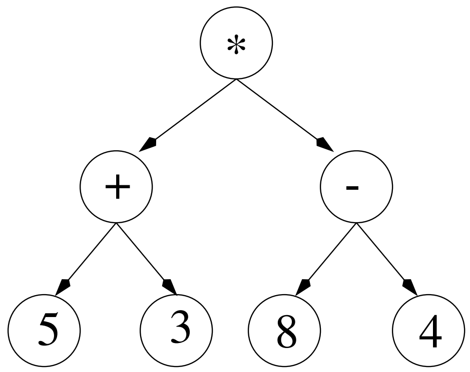

Figure 2. Example of a tree

- Ease of evolution of new programs (crossover, mutation)
- Speed of evaluation
- Ease of code maintenance
- Flexibility of user supplied code

User supplied code can be incorporated into the system in the form of new operators. The standard system provides operators for many common (and possibly uncommon) operations, but sometimes a new one will be required for a particular problem. The system gives the user facilities for providing code to perform the desired operation. The system then uses the operation in TR programs just as it uses any of the built-in operators.

2.3. Algorithms for Representation Conversion

The two representations described above share many features. They both represent an ordered sequence of operations performed on a set of incoming operands. Operands are either variables or constants. For the purpose of this presentation we will assume that a program (in either tree or procedural representation) has a single starting point (root node) and produces a single result. The algorithm conversion process consists of translating the sequence of operations specified in one representation into the other representation.

2.4. Procedural Representation to Tree Representation

The conversion of a PR program to a TR program is done by a recursive algorithm that starts at the top-level routine in the PR program. As the algorithm constructs the TR representation of that routine, it will encounter calls to lower-level *composite* routines. Sub-trees for each lower-level routine are generated as needed by recursively applying the conversion algorithm.

The basic approach of the PR to TR conversion algorithm is to scan the PR program backward to find all of the operations that contribute to the final value of the output argument of the top-level routine. As these operations are found, sub-trees are constructed and incorporated into the final TR program tree.

The conversion algorithm makes use of a data structure that is referred to as a *search-variable-set*. This is a list of variables whose values are known to contribute to an output result. As a tree representation is being constructed, this is the set of variables whose values or derivations are currently unresolved in the partially constructed tree.

This algorithm is presented below in a pseudo-code form as a single recursive procedure. This procedure takes a single PR routine and generates a list of TR trees, one for each output argument of the PR routine. All of the terminals of the generated TR trees are formal arguments of the PR routine or constants. Note that the top-level routine of a PR program is assumed to have only one output argument, so this procedure will generate a single tree when given that top-level PR routine.

```
BEGIN PR-TR-Convert (INPUT: PR routine,
                     OUTPUT: list of TR trees)

  Set each output tree to be a single node tree:
    one for each output argument of the PR routine
  Set the current-search-variable-set to the set
    of output variables of the current PR routine

  LOOP over calls in current PR routine IN REVERSE
    ORDER

    IF current call is to an atomic routine then
      Make subtree(s) corresponding to this call
    ELSE
      Invoke PR-TR-Convert ( PR sub-routine,
                             list-of-subtrees)
      (This recursive call makes a list of
       sub-trees corresponding to this call, one
       for each output arg of the called routine.)

      Substitute actual arguments in current call
        for the formal arguments in the generated
        sub-tree(s).
    ENDIF

    FOR each search-variable in
      current-search-variable-set
        IF there is a sub-tree corresponding to
          search-variable
            Substitute sub-tree for corresponding
              node in the output list of TR trees
            Remove search-variable from
              current-search-variable-set
        END IF
    END FOR

    Add each variable that is used as an input
      argument to the current call to the
```

```
    current-search-variable-set

  END LOOP over calls

END PR-TR-Convert
```

We are converting only that part of the whole program to a tree in TR which contributes to the output argument of the *top-level* procedure. This may lead to some loss of *data* in the sense of *introns* (i.e. code that does not participate in creating the final result), but it won't lead to any loss of *information* since the program as a whole has only one output argument.

2.5. Tree Representation to Procedural Representation

Conceptually, the translation from TR to PR is simple. When the tree representation is evaluated, nodes are traversed and a sequence of operations is performed. The translation process simply has to determine that sequence of operations and express each operation as a PR call. There are, however, several issues that must be considered.

Each operation in TR is represented as part of a tree; in PR it is represented as the call of a function. In TR, the tree shown in figure 3 can be converted as:

```
add(a,b,output);
```

Or it can be converted as:

```
add(a,b,tmp0);
assign(tmp0,output);
```

For ease of implementation, this last conversion has been adopted.

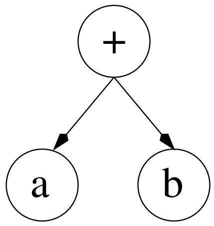

Figure 3. Simple Tree Representation (TR) program.

There are some nodes regarded as leaf nodes in our TR programs that are most appropriately treated as operators in our PR programs. A prime example of such a node is $M0$ which reads and returns memory location zero. Such a leaf node in TR will be represented in PR as a function with operand(s) that are derived as appropriate for the type of operation. For example, the TR node $M0$ will be represented as $ReadMem(0)$ in PR. To convert a program in TR to PR, it is necessary to know which leaves in a tree are true terminals, such as integers and input values, and which leaves are implied operators, such as $M0$. Leaf nodes that are to be handled as operators will be referred to as L-OP nodes.

The TR to PR conversion algorithm is presented below in pseudo-code. Nodes in the tree are traversed in the same order that they are traversed during execution. As each TR node is examined, the node is tagged with the name of a PR variable name (representing the value at that node). If that node is an operation, a PR call is created to carry out that operation. Declaration of local variables are added to the PR routine as they are needed. The L-OP nodes require a small amount of special handling. The conversion algorithm consists of the following steps:

```
BEGIN TR-PR-Convert (INPUT:  TR tree, OUTPUT: PR routine )

Initialization:
  Assign tree inputs to the names arg_0, arg_1,
    ... arg_n
  Assign tree output to the name arg_{n+1}

 Convert tree:
 LOOP over nodes in left-to-right,
  bottom-to-top order

  IF node is a true terminal
      (i.e. leaf is not an L-OP)
    Tag leaf with name assigned
      during initialization
  ELSE IF the node is a basic operation or L-OP
    Generate PR call with the following elments:
      PR routine that corresponds
         to the TR operation
      PR input arguments:
        Non-L-OP node: Use the PR variable
          names at the TR child nodes
        L-OP node: Derive appropriate PR
          variable name(s) or constant(s)
          as appropriate for this L-OP.
      PR output argument: use an available
         PR local variable name lv_j
       Add declaration of lv_j to PR
       Tag the current node with the PR variable
             name lv_j
  ENDIF

 END LOOP

END TR-PR-Convert
```

This TR to PR conversion algorithm results in a PR program that consists of a single composite routine at the top level that contains a series of calls to atomic routines.

3. Parallelization with AutoMap and AutoLink

3.1. Overview

Parallelizing the GP algorithm depicted in Figure 1 is conceptually straightforward. We used the Asynchronous Island Approach [1] [2]. This method is very efficient, sometimes reaching super-linear speed-up. The concept is to consider several populations (one per processor), and to evolve each population separately except for occasionally sending/receiving asynchronously a small number of individuals to/from each other. When generating the next population in an iteration, if the receive buffer contains a sub-population, then the programs composing this sub-population participate in the evolution process.

This was very easily done with the help of two Message Passing Interface (MPI) data-type tools that we developed: AutoMap and AutoLink [5] [7] [13] [14] [15] [17]. AutoMap is a tool that automates the process of creating data-types for use with MPI. AutoLink uses the MPI data-types generated by AutoMap; it enables sending *composed* data-types containing pointers using MPI via simple library calls. The work done by AutoMap and AutoLink on parallelizing the Genetic Programming process is equivalent to developing packing and unpacking methods for sending and receiving sub-populations. Equivalent, except that the whole process is automated. One does not have to develop and change code related to modifications of data structures from problem to problem.

In this approach, each process executes the algorithm shown in figure 4.

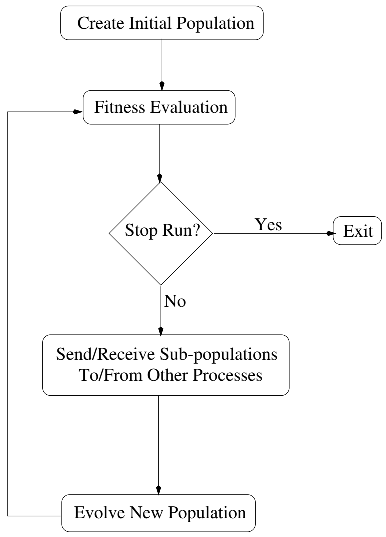

Figure 4. Simplified Algorithm for Parallel Genetic Programming

3.2. MPI Issues

The Message Passing Interface (MPI) is offered by all major computer vendors; there is also a standard for interoperability [6] among MPI implementations. Message passing is used widely on distributed memory parallel machines and clusters of computers. The MPI standard defines an easy way to work with concepts such as point-to-point and collective communications, process groups, and communication contexts. But composed data-types, such as C *structs*, can only be sent and received once they are described to the MPI library through a rather long and cumbersome series of calls. Furthermore, for dynamic data-types, it is left to the user to resolve memory references on remote processors. AutoMap and AutoLink are built on top of the MPI library to provide automated solutions to these problems.

3.3. AutoMap and AutoLink

AutoMap is designed to simplify the MPI user's task when creating composed data-types. It is a source-to-source compiler designed to read from user C data-types definition files `typedef` and `struct` entries, recognizing special directives (placed inside of C comments) and generating a set of files containing MPI data-type definition and creation procedures.

AutoLink is a library extension to MPI, designed to allow users to transfer dynamic data-types such as trees, graphs, lists, etc. via MPI. It uses AutoMap to parse the user data-type entries, then it provides the high level functions to transfer them.

The public transfer operations are based on the MPI functions of the same name, and are preceded by `AL_` (`AL_Send` for example).

AutoLink breaks dynamic data structures into Packets for sending. A tuning tool enables optimal choice of packet size for a given communications environment. The specific algorithms developed for AutoLink are:

- Send :
  1. Graph traversal (marked)
  2. Address conversion (absolute to relative)
  3. Data transfer (using packets)
- Receive :
  1. Data reception (and memory creation)
  2. Graph links recreation (reverse address conversion)

Hence, the graph is flattened and broken into packets for sending, then reconstituted with valid references on the receiving processor. Thus GP programs are sent and received with simple AutoLink calls. The needed data structures are created by AutoMap at run initialization.

4. Discussion and Future Work

The automatic translation of arbitrary programs between these two program representations is straightforward because there are many similarities in the two representations. However, as in symbiogenesis, the benefits of combining the representations into a single system are derived from their differences. Genetic programming can be thought of as a technique for searching a solution space for an optimum. The different program representations result in different solution spaces. The program translation process can be thought of as a transformation between these spaces.

The spaces contain the same set of potential solutions, but they differ in their locality properties. The idea of locality in such a space refers to how easy it is to move from one point (program) in the space to another point in the space. In a genetic programming system a program is transformed via the genetic operations such as mutation and crossover. We think of two programs as being *near* each other in the solution space if it takes *few* such operations to transform one into the other and if the required operations are relatively likely. If the operations are particularly unlikely or if very many of them are required, then we regard the two programs as being distant from each other in the solution space. This notion of distance and locality is altered by using a different program representation.

These differences in locality result in several differences in the course of evolution in the two systems. The location of local optima will be different and the paths of easy or likely progression toward a satisfactory solution will also be quite different. One of the greatest benefits that we see for this approach is that the each of the two representations will help to prevent the other from becoming trapped at local optima. Similarly, each representation may help the other to find portions of the search space that might otherwise be difficult to enter. To use a different analogy, each representation will provide *ideas* that the other representation would never have *thought* of on its own.

As mentioned above, the benefits of using multiple representations is derived from the differences among them. The greater the differences are, the greater are the potentials for benefit. But the greater the differences are, the greater are the difficulties in translation. The practical difficulties in translation of more diverse representations could require translations that are not precise. Information may have to be altered or lost during the translation process, if only for the reason that the translation would be otherwise impractical. Yet such translations can still serve the purpose of moving potentially valuable ideas between program representations within a genetic programming system.

This work was partially inspired by an observation that was made during the development of our GP systems. We were implementing some of the problems that appear in the GP literature as a way of testing our software. We implemented the Artificial Ant problem as described by Koza [9]. In this problem, we try to evolve a program that optimizes the food-seeking behavior of a simple *ant*. Koza presents a straightforward approach that uses automatically defined functions (ADFs). The artificial ant problem and the difficulties and obstacles to its solution are analyzed in great detail by Langdon [11].

Because our system was only partially implemented at the time, we implemented the problem within a finite state machine model rather than the model described by Koza. Somewhat to our surprise, we found that our system generally solved the problem quickly and did not experience the types of difficulties described by Langdon. This highlighted the fact that the ease of finding a solution to a problem is dramatically influenced by how the problem is represented. This focussed our attention on the issue of problem representation.

The general problem of translating between problem representations that are as diverse as these two representations of the artificial ant problem is a rather difficult one. Certainly we cannot expect to have an automated translation process that will work for a variety of problems and representations. This caused us to try to solve the more limited problem of automatic translation of relatively similar representations.

The work described here is still in its early stages. As of this writing, the two representations have been implemented and are functioning independently, however their integration via the translation process described above is only partially complete. But different results observed from the tree representation and the procedural representation suggest that each has its particular strengths and weaknesses. Furthermore, parallelization even of a single representation has yielded very promising results. So we anticipate that combining these representations will provide a system that will perform better in speed and effectiveness than either of the stand-alone systems. In the future, we also hope to address the harder translations of more diverse representations with problem-specific translation procedures.

5. Disclaimer

Certain commercial products are identified in this paper in order to adequately describe work related to connecting software. Such identification is not intended to imply recommendation or endorsement by the National Institute of

Standards and Technology, nor is it intended to imply that the identified products are necessarily the best available for the purpose.

References

[1] Andre, D., Koza, J. R., "A parallel implementation of genetic programming that achieves super-linear performance", In Hamid R. Arabnia, editor, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications , volume III, pages 1163-1174, Sunnyvale, 9-11 August 1996. CSREA.

[2] Bennett, F. H. III, Koza, J. R., Shipman, J., Stiffelman, O., 1999, "Building a parallel computer system for $18,000 that performs a half peta-flop per day", In Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E. (editors). 1999. GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA. San Francisco, CA: Morgan Kaufmann. Pages 1484 - 1490.

[3] Bloedorn, E., Michalski, R. S., "Data-Driven Constructive Induction", IEEE Intelligent Systems, 30-37, March/April, 1998.

[4] Devaney, J. E. "Experience with MPI: 'Porting pvmmake to mpimake' and 'Parallel Genetic Programming'", MPI Developers Conference, June 22-23, 1995, Notre Dame, IN.

[5] Devaney, J.E., Michel, M., Peeters, J., Vrielink, K., "AutoLink: An MPI C Library For Sending and Receiving Dynamic Data Structures", technical report, April, 1997, http://www.itl.nist.gov/div895/savg/auto/.

[6] George, W. L., Hagedorn, J. G., Devaney, J. E., "IMPI: Making MPI Interoperable", with appendix I by IMPI Steering Committee, "IMPI: Interoperable Message-Passing Interface", Protocol Version 0.0, January, 2000, http://impi.nist.gov/IMPI/, Journal of Research of the National Institute of Standards and Technology, May-June 2000.

[7] Goujon, D., Michel, M., Peeters, J., Devaney, J.E., "AutoMap and AutoLink: Tools for Communicating Complex and Dynamic Data-structures Using MPI", Lecture Notes in Computer Science, Volume 1362, pp 98-109, 1998, Springer-Verlag.

[8] Koza, J. R., "Genetic Programming", MIT Press, Cambridge, MA, 1992.

[9] Koza, J. R., "Genetic Programming II", MIT Press, Cambridge, MA, 1994.

[10] Koza, J. R., "Genetic Programming III", Morgan Kauffman, Cambridge, MA, 1999.

[11] Langdon, W. B., Poli, R., "Why ants are hard", Proceedings of the Third Annual Genetic Programming Conference, University of Wisconsin, Madison, Wisconsin, July 22-25, 1998.

[12] Margulis, L., "Symbiotic Planet: A New Look at Evolution", Basic Books, New York, 1998.

[13] Michel, M., Devaney, J. E., "A Generalized Approach for Transferring Data-Types with Arbitrary Communication Libraries", Proceeding of the Workshop on Multimedia Network Systems (MMNS'2000) at the 7th International Conference on Parallel and Distributed Systems (ICPADS '2000), July 4-7, 2000, at Iwate, Japan.

[14] Michel, M. Schaff, A., Devaney, J. E., "Managing data-types: the CTRBA approach and AutoLink, an MPI solution", Proceedings of the Message Passing Interface Developer's and User's Conference, March 10-12, 1999, Atlanta, GA.

[15] Michel, M., Devaney, J. E., "Fine Packet Size Tuning with AutoLink", Proceedings of the International Workshop on Parallel Computing (IWPP '99), September 21-24, Aizu, japan.

[16] Tanese, R., "Distributed Genetic Algorithms for Function Optimization", PhD Dissertation, Department of Electrical Engineering and Computer Science, University of Michigan, 1989.

[17] Vrielink, K. H. J., Baland, E. C., Devaney, J. E. "AutoLink: An MPI Library for Sending and Receiving Dynamic Data Structures", International Conference on Parallel Computing, University of Minnesota Supercomputer Institute, october 3-4,1996.

[18] Winston, P. H., "Artificial Intelligence", Addison-Wesley, Reading, MA, 1993.

[19] Wnek, J. Michalski, R. S., "Hypothesis-Driven Constructive Induction in AQ17-HCI - A Method and Experiments", Machine Learning, 14:(2), 139-168, Feb., 1994.