A Genetic Programming System with a Procedural Program Representation

John G. Hagedorn
National Institute of
Standards and Technology
100 Bureau Drive, Stop 8911
Gaithersburg, MD 20899-8911

Judith E. Devaney
National Institute of
Standards and Technology
100 Bureau Drive, Stop 8911
Gaithersburg, MD 20899-8911

Abstract

We describe the status of a genetic programming (GP) system that is based on a procedural program representation. The procedural representation is closely related to the high level programming languages used by human programmers; it includes features such as hierarchies of procedure calls, with arguments lists that allow multiple output values from each procedure. This representation is structurally different than previous representations used in GP and is expected to have different evolutionary properties. The system architecture is presented and specific benefits as well as problems and solutions arising from this program representation are described. Two mutation-like operations, repair and pruning, are introduced. A population visualization technique is described that includes the graphical presentation of program structure, ancestry, and fitness. This visualization tool and other instrumentation are used to investigate population diversity and fitness evolution. An unexpected benefit of the pruning operation is described.

1 INTRODUCTION

Scientists at the National Institute of Standards and Technology (NIST) do research in a wide variety of scientific disciplines that pertain to measurement science and standards. This diverse environment encompasses very discipline specific problems whose solutions often involve sophisticated computer models. Ongoing research includes the development of algorithms, optimization of models and their parameters, and pattern recognition and characterization in scientific datasets. The capability of Genetic Programming (GP) to treat whole computer codes as single functions allows us to incorporate discipline specific codes, thus enabling us to use genetic programming to ask questions relevant to these research areas. Having our own genetic programming system, functionality tailored to problems of interest to NIST scientists, will greatly assist our efforts in these areas.

In this project we are implementing a genetic programming system which uses a program representation that is modelled on the higher level programming languages typically used by human programmers, such as C or Fortran. This is in contrast to the LISP-like representation that is often used in genetic programming systems [Koza, 1992, Koza, 1994, Koza et al., 1999]. We describe our program representation as a "procedural" representation in order to highlight the structuring of the program into a hierarchy of routines, each composed of a sequence of calls to other routines.

The decision to use this program representation is motivated by two factors. First, we expect that the program structures that are useful to human programmers may prove to be useful in evolved programs. Second, the different program representation results in different characteristics of the program search space. Although the same programs can be expressed in either the LISP-like syntax or in our procedural structures, the search landscape will certainly be different. Hence the procedural representation could yield better results for some problems. Of course, by the same token, the procedural representation could yield worse results in other cases [Wolpert and Macready, 1997]. This suggests that these two representations (and perhaps others) might be used together in a single system. This idea is being explored in independent project [Devaney et al., 2001].

We have many problems to which we intend to apply this system. For example, one issue of importance for NIST is the identification and characterization of systematic or experimental measurement errors. As High Throughput Methods (combinatorial experiments) become more commonplace, the datasets become larger and automated methods become more important. We plan to use genetic programming on these datasets to extract and find functional forms for the measurement errors. Another possible problem to which we will apply this system is in the design of concrete. Algorithms exist for modeling the properties of concrete based on its composition and processing. These algorithms can be incorporated into the system, enabling us to optimize for desired physical properties.

2 RELATED WORK

Most program representations in GP can be classified into one of three categories: tree, linear, and graph representations [Banzhaf et al., 1998]: Our procedural representation does not quite fit into any of these.

The tree representation is probably the most common and easily handled program representation used in GP. Typically a program is represented as essentially a LISP S-expression [Koza, 1992]. Branching nodes on the tree are operators, with each branch being an operand. Leaves are values such as input variables or constants. In our representation, the procedures certainly form a tree based on the calling hierarchy of the procedures, but sequences of calls that occur at each node is quite different in meaning from the single operations that occur in a function in an S-expression. Furthermore, the capability of producing multiple independent outputs also distinguishs the procedure representation from the S-expression.

The linear category includes program representations in the form of sequences of tokens. There are many forms of linear representations such as bit-strings describing operations and operands [Banzhaf, 1993], Java byte code [Lukschandl et al., 2000], or even machine code [Nordin, 1994]. In our representation, the sequences of calls within a single procedure resembles a linear representation, but the hierarchical nature of procedure calls places it outside of this category.

Graph-based GP systems represent programs as collections of nodes and connecting arcs. One such system is PADO [Teller and Veloso, 1996]. The PADO program representation is based on a directed graph; each node specifies an action and a branch-decision and is equipped with a stack and memory. After a node performs its action, the branch-decision determines which arc is taken, thus determining the next node. Another example of a GP system that uses a graph-based representation is Poli's Parallel Distributed Genetic Programming [Poli, 1997] (PDGP). As in the tree represenation, nodes are operators or values, but PDGP allows the nodes to be connected in a directed graph with fewer constraints.

In addition to these three categories, various other program representations have been used in GP. For example, much work has been done with representations that use grammars to structure programs and evolutionary operations [Whigham, 1995, Wong and Leung, 1996, Freeman, 1998]. Another interesting program representation is Yu's based on functional programming GP system, which generates programs based on the syntax of the λ *calculus* [Yu, 1999].

But the procedural representation described here is, in a sense, a hybrid of the tree and the linear representations. It has a tree structure based on the hierarchy of procedure calls, and each procedure is a linear sequence of operations. Each operation in this sequence is a call to another procedure. The tree of procedure calls ends in leaves that are built-in procedures that represent the basic operations of code, such as addition or subtraction. Our procedural representation is described in more detail below.

3 DESCRIPTION OF THE SYSTEM

3.1 ARCHITECTURE

The software is provided as a library of C routines. The user calls these routines to specify the problem to be solved, to create the initial population, and to specify aspects of the evolution. Execution of individuals, the genetic operations such as mutation and crossover, and evolution are all handled with the library routines.

The user creates a GP system for a particular problem by calling library functions for these operations:

- *Define characteristics that apply to all individual programs.* This includes specification of inputs and outputs of the top-level procedure, and characteristics such as allowable branching and depth.
- *Provide the fitness evaluation procedure.* This is done by passing a function pointer.
- *Specify problem-specific operators.* This is done by providing a function pointer and an argument list descriptor for each operator.
- *Specify various operating parameters.* These parameters include items such as population size, mutation rates, and others.
- *Initiate population creation and evolution.*

It is important to note that a problem-specific operator can encapsulate any algorithm, no matter how complex, that may be related to the problem at hand. This is one of the reasons that GP is an attractive technique for the problems we address at NIST. Domain-specific knowledge can easily be included into our GP system by turning it into one of these customized operators. The GP system can then make use of the algorithm by including the operator in the evolved programs.

3.2 PROGRAM REPRESENTATION

Each individual program within the evolving population is described as a hierarchy of procedures, which we also refer to as routines. We do not call these functions because they do not have return values. All data, both input and output, are passed through items in an argument list. Here are some important 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 within a calling routine must honor these I/O attributes.
- Each formal argument may have a specified data type or the data type may be left unspecified. If it is left unspecified, the data type is determined at run time from 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.
- Data items such as arguments or local variables may be either scalars or arrays.
- There are no global data.
- 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.
- There are composite routines that can be reused multiple times within a program structure. These are intended to serve the same purpose as automatically defined functions (ADFs) as described by Koza [Koza, 1994].

Note that this program representation enables the modeling of programs that have multiple outputs.

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, in this example, 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 PN01 ( void * arg00,   /* IN      */
            void * arg01,   /* IN      */
            void * arg02,   /* IN      */
            void * arg03    /*     OUT */  )
    {
    void * lv00 ;   /* Like arg 2 */
    /* end of local variable list */
    add (arg00, arg01, arg03);
    mult (arg00, arg03, arg01);
    PN02 (arg01, arg03, arg02, arg03, lv00);
    }  /* end of PN01 */

void PN02 ( void * arg00,   /* IN      */
            void * arg01,   /* IN      */
            void * arg02,   /* IN      */
            void * arg03,   /*     OUT */
            void * arg04    /*     OUT */)
    {
    /* end of local variable list */
    sub (arg00, arg01, arg04);
    div (arg01, arg02, arg03);
    add (arg03, 3.900, arg04);
    }  /* end of PN02 */
```

3.3 ISSUES ARISING FROM PROGRAM REPRESENTATION

Various problems are posed by this representation that are not present when using a S-expression representation. Many of these problems result from the use of argument lists and typed variables.

For example, in this procedural program representation, calculations may be performed and the results placed in a variable. But before this result is used, the value of the variable may be over-written by the result of another calculation. Here is a list of this and related problems:

- *Overwritten result:* The value of a variable is overwritten before it is used.

- *Unused result:* A procedure ends with a value in a local variable that has not been used.
- *Unset output argument:* A procedure ends without setting the value of an output argument.
- *Uninitialized variables:* The value of a local variable is used before it is set. (Note that all local variables are initialized to zero at the start of a routine.)

These problems are not necessarily damaging to the operation of a routine. But we discovered that they occur frequently enough that the presence of these situations seemed to have a negative impact on successful evolution. To address these issues we developed a program *repair* operation. This operation scans a program and looks for these situations and makes changes in the use of the variables in order to eliminate the situation. Clearly the *repaired* program functions differently from the original program. We apply the *repair* operation in somewhat the same way that we apply the mutation operation. At each generation a certain percentage of the individuals are selected for repair; the resulting individuals are placed in the population for the next generation.

Another source of difficulty that is caused by this procedural representation is the valid operation of crossover. Given the tree structure of procedure calls, it seems reasonable to perform crossover by exchanging procedural sub-trees between two parents. The problem arises from the fact that the argument lists of the procedures at the root of the two sub-trees may not be the same. So when we try to replace a call to a procedure with a call to a different procedure, the actual argument list for the call to the old procedure may not be compatible with the formal argument list of the new procedure. We address this problem with an algorithm that reconciles the old actual argument list with the new formal argument list. This is done by trying to preserve entries in the actual argument list, using them as *input*, *output*, or *input/output*, arguments as they were in the old call. Arguments are eliminated or added as needed.

Problems also arise from the use of typed data. Data type clashs can occur when, for example, an integer is passed as an actual argument when the formal argument is a floating point variable. Another kind of type clash occurs when scalar and array variables are mixed. We are currently addressing this problem by providing for automatic type conversion for all cases. We are also looking at the possibility of implementing strong typing.

Another problem that is not unique to our representation is program bloat. Program bloat is a common problem and techniques for reducing bloat have been studied [Langdon et al., 1999]. Program size can grow to the point of causing extreme slow-down of evolution. We address this problem by introducing a *prune* operation. This procedure goes through a program and attempts to cut off as many program branches as possible without hurting the fitness of the program. As with the *repair* operation, the *prune* operation is applied in a fashion similar to mutation.

This explicit form of pruning seems to be unlike many previous attempts to control program bloat. The typical direct approach to this problem involves applying a fitness penalty so that larger programs are regarded as less fit than smaller programs. Our form of pruning attempts to modify existing programs to make them smaller *without degrading their fitness.* Operations similar to this are sometimes used as a type of mutation, such as the *trunc* operation of Chellapilla, the *shrink* operation described by Langdon [Langdon, 1998], or the *delete subtree* operation of Angeline [Angeline, 1998]. Our form of pruning is also very similar to the reduced error pruning technique applied to decision trees [Mitchell, 1997].

4 OBSERVATIONS OF POPULATION EVOLUTION

4.1 EFFICACY OF VARIOUS OPERATING PARAMETERS

We have run a series of tests on several of the standard problems that appear in the GP literature, including the two box problem [Koza, 1994] and a symbolic regression problem. While these problems are quite simple, currently they are the examples for which we have the most data. We have run these problems many times with various combinations of operating parameters (population size, mutation rate, etc.). Each run was initialized with different seeds to the random number generator, ensuring that all runs were unique. These runs, while not exhaustive, have produced fairly consistent results. Of course, these results might not be applicable to all types of problems.

Much of what we found from these tests was expected. For example, we found that larger populations were more successful on average than smaller populations. But other results were not entirely expected. We found that a tournament size of 3 is better than a tournament size of 7; however completely random selection (tournament size of 1) works surprisingly well. We found that crossover is not only unnecessary, but that it tends to hurt the system's ability to find solutions. It appears that mutation and repair (Section 3.3) are the two main agents that generate movement toward a solution. Both are essential and neither is effective without the other.

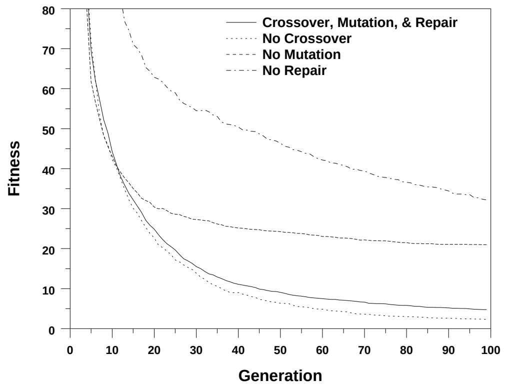

Figure 1: Average fitness for four different combinations of operating parameters. Fitness is averaged over 120 runs for each set of parameters.

Figure 4.1 shows how different combinations of operating parameters can affect fitness. Each curve is the average of 120 runs of the two box problem [Koza, 1994]. One set of runs was done with an equal proportion of crossover, mutation, and repair. Another set of runs was done with mutation and repair but no crossover, another with crossover and repair but no mutation, and another with crossover and mutation but no repair. The plot indicates that crossover has no beneficial effect and the runs without crossover actually have a somewhat better fitness on average than the runs with crossover. The runs with no mutation or no repair are clearly seen to be substantially worse that the runs in which both mutation and repair are used.

4.2 PRUNING

An important result was that we found that our pruning operation not only acts to control program bloat without damaging fitness, but sometimes acts to improve the fitness of programs on which it is applied. We have observed cases in which the pruning operation has acted as a type of mutation that produces a program that is substantially more fit that the program that it started with. At times, pruning has been observed to produce the most fit individual of the generation. This is an unexpected and intriguing result and we will be investigating this effect in more detail.

4.3 INSTRUMENTATION OF THE CODE

We have instrumented our GP system to collect information about each generated program during the evolution of the population. This information includes number and type of procedures called, usage of variables and arguments, history of genetic operations (mutation, crossover, etc.), generation of creation, and ancestry.

These data have proven useful in guiding the development of the GP system. For example, the statistics on the use of local variables and arguments motivated the implementation of the repair function described in Section 3.3.

4.4 VISUALIZATION

The data provided by the instrumentation described in Section 4.3 proved to be difficult to interpret in many situations. This motivated the implementation of a visual representation of populations and individual programs. These visualizations do not capture all aspects of the programs but we have found that they have provided interesting insights into the evolution of populations. For each individual in the population, the visualizations provide information about the fitness, ancestry, program structure, and composition.

(Note: Figures 2 and 3 can be found on-line at http://math.nist.gov/mcsd/savg/papers/g2001.html.)

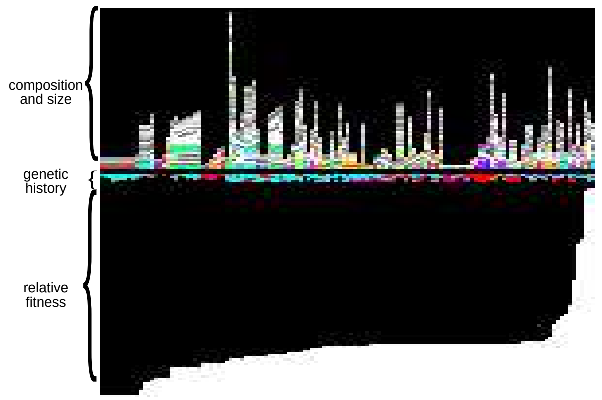

Figure 2: Visualization of a population.

Figure 2 shows a visualization of a population of 128 individuals. Although reproduced here in grey levels, the image contains colors that convey important information. Each program is represented by one vertical column. As indicated in the figure, three aspects of each program are represented. The upper part is a visual representation of the of the content of the program. Each block of color in this section corresponds to a procedure in the program. Atomic routines are each given a different grey level and composite procedures are assigned a color that indicates ancestry. In the middle section, the sequence of genetic operations that brought each individual into existence is presented. Each operation such as crossover, mutation, repair, and pruning is given a unique color. Finally, the lower portion of the image presents a normalized view of the fitness of each individual. In Figure 2, the individuals have beens sorted by fitness with the most fit individuals on the left.

These visualizations have been particularly useful in gaining insights into the preservation and loss of diversity in populations. In Figure 2 we can see groups of programs that are clearly related by structure and ancestry. Figure 3 shows the same population at a later stage of evolution. The loss of diversity is evident. Almost all members of the population are derived from only two ancestors. Furthermore, it is clear that many individuals share identical genetic histories through approximately half of the duration of the evolution, suggesting a common ancestor from that generation. Looking at the composition of the programs, it is also easy to see large segments of the population within which all members have very similar structures.

As a result of these visualizations, we feel that loss of diversity, as illustrated so dramatically in Figure 3, can be a substantial problem. The qualitative assessments of the population via visualization has motivated and is guiding our efforts in this area.

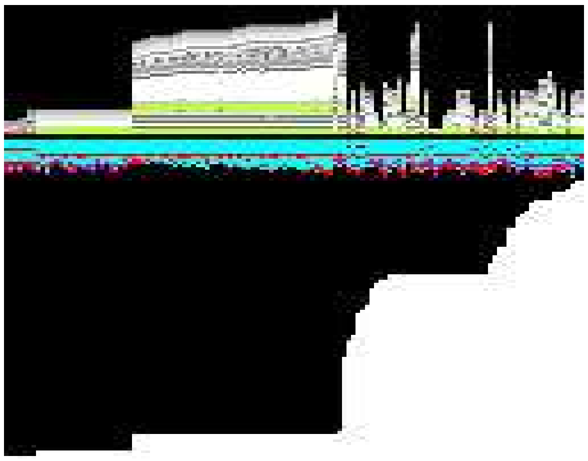

Figure 3: A population showing a substantial loss of diversity.

5 FUTURE WORK

Our GP system, based on a procedural program representation, provides a foundation on which we are continuing to build. There are many directions in which we would like to extend the system. We intend, for example, to provide greater flexibility in the the program structures allowed. One possibility in this area is to allow an individual within the population to consist of a group of program branches. The programs within a group would co-evolve and it would be the responsibility of the fitness evaluator to apply the multiple branches in concert to the desired problem.

Another area that we would like to enhance and investigate is the use of data types. Our program structures enable us to give data type and scalar/array attributes to data items in the programs. But this is a feature of the system that remains largely unexploited and unexplored. Finally, we intend to do substantial development in the areas of diversity and parallelization as described in the following sections.

5.1 MEASURES OF DIVERSITY

Diversity is considered to be an important factor in the success of program evolution [Koza, 1992, Banzhaf et al., 1998]. Langdon in particular has done an extensive investigation into the underlying mechanisms that affect diversity and has performed statistical analyses of actual evolutions [Langdon, 1998].

The visualizations described in Section 4.4 have provided important qualitative information on the preservation and loss of diversity in evolving populations. On the basis of these visualizations, we have become convinced that loss of diversity is a critical issue in the failure of GP runs. But without a more quantitative measure of diversity, we will have difficulty assessing the success of any effort to preserve diversity.

We have begun implementing a diversity measure based on a the measurement of similarity between pairs of programs in the population. We are using Levenshtein or edit distance [Sankoff and Kruskal, 1983] as this measure of similarity. This measure was used by O'Reilly [O'Reilly, 1997] as a tool for understanding the effect of crossover on population diversity. O'Reilly describes two diversity measures based on edit distance (BI-POP distance, and BI-POOL distance) that are based on the selection of a small number of individuals from the collection of best individuals and calculating distances from these individuals to other members of the population. We are implementing these metrics as well as another of our own design.

We should note that our procedural program representation presents a few problems in the implementation of the algorithm for calculating edit distance. Our composite nodes and the information contained in argument lists do not easily fit the schemes described by Sankoff and Kruskal [Sankoff and Kruskal, 1983]. We are addressing this problem by simplifying the representation before calculating the edit distance. We simply regard each program as a string of tokens; each token corresponds to a procedure in the program. All composite procedures are represented by the same token, and each atomic function is represented by a unique token. They are arranged in execution order. Edit distance between two programs is then computed by the conventional algorithm for calculating edit distance between two strings. Although this approach, of course, discards lots of information, it also retains a great deal of information about the structure and content of each program. We believe that the simplification of the data and the resulting simplification of the algorithm will give us a diversity measure that is both useful and reasonably easy to compute.

We are also pursuing another approach to gauging diversity that is based on tracking the ancestry of individuals. Our instrumentation enables us easily to determine how many ancestors from a given previous generation are represented in the current population. We believe that diversity is likely to be better when more individuals have descendants in subsequent generations. If this is so, then this measurement of ancestry could prove a powerful indicator of diversity.

5.2 PARALLELIZATION

We are planning to parallelize our GP system using the asynchronous island model [Bennett III et al., 1999, Andre and Koza, 1996]. For portability among the various computing platforms at NIST, we are basing our parallelization on the Message Passing Interface (MPI) [MPI Forum, 1998]. MPI is an industry-standard library of message passing routines available on a wide variety of platforms. With the definition of the Interoperable MPI standard [George et al., 2000], we anticipate that it will be increasingly feasible to distribute our GP runs across heterogeneous collections of machines.

The relatively complex data structures used in our procedural representation could make passing individual programs via MPI difficult. This problem is addressed by AutoMap and AutoLink [Goujon et al., 1998, Michel and Devaney, 2000]. These MPI-based tools were developed at NIST to facilitate the passing of complex C data structures. AutoMap is a tool that automates the process of creating data-types for use with MPI, and AutoLink enables the sending of these *composed* data-types containing pointers using MPI via simple library calls. All MPI data type creation, data packing, unpacking, and pointer referencing and dereferencing is handled automatically. One of the large benefits of AutoMap and AutoLink is that changes to data structures that might occur from problem to problem do not require recoding. These tools greatly simplify the passing of our procedurally defined programs among processes in a parallel processing environment.

Acknowledgments

We would like to acknowledge valuable input and support from Barbara Cuthill and John Hewes, and financial support from the NIST Advanced Technology program. We would also like to thank Bill Langdon for his important suggestions for improving this paper.

References

[Andre and Koza, 1996] Andre, D. and Koza, J. R. (1996). A parallel implementation of genetic programming that achieves super-linear performance. In *Proc. International Conf. on Parallel and Distributed Processing Techniques and Applications*, volume III, pages 1163–1174. CSREA.

[Angeline, 1998] Angeline, P. J. (1998). Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806.

[Banzhaf, 1993] Banzhaf, W. (1993). Genetic programming for pedestrians. In Forrest, S., editor, *Proc. 5th International Conf. on Genetic Algorithms*, page 628. Morgan Kaufmann.

[Banzhaf et al., 1998] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann.

[Bennett III et al., 1999] Bennett III, F. H., Koza, J. R., Shipman, J., and Stiffelman, O. (1999). Building a parallel computer system for $18,000 that performs a half peta-flop per day. In *Proc. of the Genetic and Evolutionary Computation Conf.*, volume 2, pages 1484–1490. Morgan Kaufmann.

[Devaney et al., 2001] Devaney, J., Hagedorn, J., Nicolas, O., Garg, G., Samson, A., and Michel, M. (2001). A genetic programming ecosystem. In *Proceedings 15th International Parallel* *and Distributed Processing Symposium*, page 131. IEEE Computer Society. Available online: <http://math.nist.gov/mcsd/savg/papers/bio.ps>. Accessed 16 May 2001.

[Freeman, 1998] Freeman, J. J. (1998). A linear representation for GP using context free grammars. In *Genetic Programming 1998: Proc. Third Annual Conference.*, pages 72–77. Morgan Kaufmann.

[George et al., 2000] George, W. L., Hagedorn, J. G., and Devaney, J. E. (2000). IMPI: Making MPI interoperable. *J. Res. Natl. Inst. Stand. Technol.*, 105(3):343–428.

[Goujon et al., 1998] Goujon, D., Michel, M., Peeters, J., and Devaney, J. E. (1998). AutoMap and AutoLink: Tools for communicating complex and dynamic data-structures using MPI. In *Lecture Notes in Computer Science*, volume 1362, pages 98–109. Springer-Verlag.

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

[Koza, 1994] Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

[Koza et al., 1999] Koza, J. R., Andre, D., Bennett III, F. H., and Keane, M. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufman.

[Langdon, 1998] Langdon, W. B. (1998). *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming.* Kluwer.

[Langdon et al., 1999] Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999). The evolution of size and shape. In *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press.

[Lukschandl et al., 2000] Lukschandl, E., Borgvall, H., Nohle, L., Nordahl, M., and Nordin, P. (2000). Distributed java bytecode genetic programming. In *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 316–325. Springer-Verlag.

[Michel and Devaney, 2000] Michel, M. and Devaney, J. E. (2000). A generalized approach for transferring data-types with arbitrary communication libraries. In *Proc. Workshop on Multimedia Network Systems, 7th International Conf. on Parallel and Distributed Systems*, Iwate, Japan.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill.

[MPI Forum, 1998] MPI Forum (1998). MPI-2: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 12(1-2).

[Nordin, 1994] Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code. In *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press.

[O'Reilly, 1997] O'Reilly, U.-M. (1997). Using a distance metric on genetic programs to understand genetic operators. In *Late Breaking Papers, 1997 Genetic Programming Conference*, pages 199–206. Stanford University.

[Poli, 1997] Poli, R. (1997). Evolution of graph-like programs with parallel distributed genetic programming. In Back, T., editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353. Morgan Kaufmann.

[Sankoff and Kruskal, 1983] Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley.

[Teller and Veloso, 1996] Teller, A. and Veloso, M. (1996). PADO: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81–116. Oxford Univ. Press.

[Whigham, 1995] Whigham, P. A. (1995). Grammatically-based genetic programming. In Rosca, J. P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

[Wong and Leung, 1996] Wong, M. L. and Leung, K. S. (1996). Evolving recursive functions for the even-parity problem using genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press.

[Yu, 1999] Yu, G. T. (1999). *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming.* PhD thesis, University College, London.