

1. INTRODUCTION

In this section we present genetic programming, being the fourth member of the evolutionary algorithm family. Besides the particular representation (using trees as chromosomes) it differs from other EA strands in its application area. While the EAs are typically applied to optimization problems, GP could be rather positioned in machine learning. In terms of nature of this deferent problem types, most other EAs are used for finding some input realizing maximum payoff, whereas GP is used to seek models with maximum fit. Clearly, once maximization is introduced, modelling problems can be seen as special cases of optimization. This, in fact, is the basis of using evolution for such tasks: models are treated as individuals, their fitness being the model quality to be maximized.

The one glance summary of GP is given in Table 1.

Representation	tree structures
Recombination	exchange of subtrees
Mutation	random change in trees
Parent selection	fitness-proportional
Survivor selection	generational replacement

Table 1: Sketch of GP

2. INTRODUCTORY EXAMPLE

We will consider a credit scoring problem within a bank that lends money and keeps a track on how its customers are paying back their loan. This information about the clients can be used to develop a model describing good, respectively bad customers. Later on, this model can be used to predict customer's behavior and thereby assist in evaluating future loan applicants. Technically, the classification model will be developed based on (historical) data holding personal information along with a credibility index (good or bad) of customers. The model will have personal data as input and produce a binary output. For instance, the annual salary, the marriage status, and the number of children can be used as input. In Table 2 a small data set is shown. A possible classification model using these data might be the following:

IF (Number of children = 2) AND (Salary > 80000) THEN good ELSE bad In general, the model will look like this:

IF formula THEN good ELSE bad.

Customer id	# of children	Salary	Marriage status	Creditability
Id-1	2	45.000	married	0
Id-2	0	30.000	single	1
Id-3	1	40.000	married	1
Id-4	2	60.000	divorced	1
...
Id-10000	2	50.000	married	1

Table 2: Data for the credit scoring problem

Notice, that formula is the only unknown in this rule, all other elements are fixed. Our goal is thus to find the optimal formula, forming an optimal rule that classifies a maximum number of known clients correctly.

At this point we have formulated our problem as a search problem in the space of possible formulas \mathcal{F} , where the quality of a formula f can be defined as the percentage of customers correctly classified by the model IF f THEN good ELSE bad. In evolutionary terms we have defined the phenotypes (formulas) and the fitness (classification accuracy). In accordance with the typical GP approach we use parse trees as genotypes representing formulas. Figure 1 shows the parse tree of the formula above. This representation differs from the ones used in GAs or ES in two important

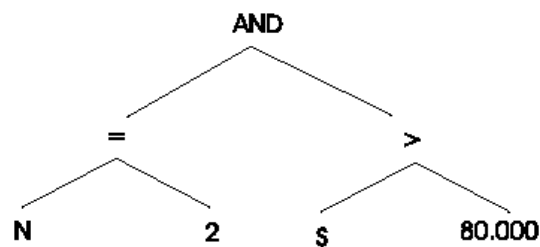


Figure 1: Parse tree

aspects:

- The chromosomes are non-linear structures, while in GAs and ES they are typically linear vectors of the type $\langle v_1, \dots, v_n \rangle$, v_i being the domain of v_i .
- The chromosomes can differ in size, measured by the number of nodes of the tree, while in GAs and ES their size, the chromosome length n , is usually fixed.
-

This new type of chromosomes necessitates new, suitable variation operators acting on trees. Such crossover and mutation operators will be discussed in the sections 4 and 5. As for selection, notice that it only relies on fitness information and therefore it is independent from the chromosome structure. Hence, any selection scheme known in other EAs, e.g., fitness-proportional with generational replacement, can be simply applied.

* Notice, that we have not defined the syntax, thus the space of possible formulas, exactly. For the present treatment this is not needed and Section 3 will treat this issue in general.

3. REPRESENTATION

As the introductory example has shown, the general idea in GP is to use parse trees as chromosomes. Such parse trees are to capture expressions in a given formal syntax. Depending on the given problem and the users perception on what the solutions must look like, this can be the syntax of arithmetic expressions, formulas in first-order predicate logic, or code written in a programming language. To illustrate the matter, let us consider one of each of these types of expressions: an

$$2 \cdot \pi + ((x+3) - \frac{y}{5+1}) \tag{1}$$

a logical formula

$$(x \wedge \text{true}) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y))) \tag{2}$$

and the following program.

```
i = 1;
while (i < 20)
{
    i = i+1
}
```

Figures 2 and 3 show the parse trees belonging to these expression.

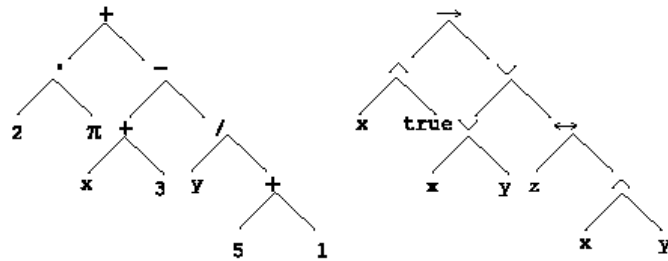


Figure 2: Parse trees belonging to the formulas 1 (left) and 2 (right).

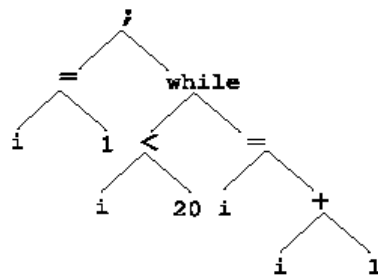


Figure 3: Parse tree belonging to the above program

These examples illustrate how generally parse trees can be used and interpreted. Depending on the interpretation GP can be positioned in different ways. From a strict technical point of view, GP is simply an variant of GAs working with a different data structure: the chromosomes are trees. This view disregards the application dependent interpretation issues. Nevertheless, such parse trees are often given an interpretation. In particular, they can be envisioned as executable codes, that is, programs. The syntax of functional programming, e.g, the language LISP, very closely matches the so-called Polish notation of expressions. For instance, the formula in equation 1 can be rewritten in this Polish notation as

$$+(\cdot(2, \pi), -(+(x, 3), /(y, +(5, 1)))) \quad (3)$$

while the executable LISP code would look like:

$$(+, (\cdot, 2, \pi), (-, (+, x, 3), (/ , y, (+, 5, 1)))) \quad (4)$$

Adopting this perception GP can be positioned as the "programming of computers by means of natural selection" [5], or the "automatic evolution of computer programs" [2]. In the following we describe genetic programming with a rather technical flavor, that is, we emphasize the mechanisms, rather than the interpretation and context specific issues. Technically speaking the specification of how to represent individuals in GA boils down to defining the syntax of the trees, or equivalently the syntax of the symbolic expressions (s-expressions) they represent. This is commonly done by defining a function set and a terminal set. Elements of the terminal set are allowed as leaves, while symbols of function set are internal nodes. For example, a suitable function and terminal set that allows the expression in equation 1 as syntactically correct is given the following table.

Function set	{+, -, \cdot, /}
Terminal set	$\mathbb{R} \cup \{x, y\}$

Table 3: Function and terminal set that allow the expression in equation 1 as syntactically correct. Strictly speaking a function symbol from the function set also must be given an arity, that is the number of attributes it takes must be specified. For standard arithmetic or logical functions this is often omitted. Furthermore, for the complete specification of the syntax a definition of correct expressions (thus trees) based on the function and terminal set must be given. This definition follows the general way of defining terms in formal languages and therefore is also often omitted.

For the sake of completeness we provide it below:

- * All elements of the terminal set T are correct expressions.

- * If f element of F is a function symbol with arity n and e_1, \dots, e_n are correct expressions, then so is

$f(e_1, \dots, e_n)$.

- * There are no other forms of correct expressions.

Note that in this definition we do not distinguish different types of expressions, each function symbol can take any expression as argument. This feature is known as the closure property in GP. In general, it can happen that function symbols and terminal symbols are typed and there are syntactic requirements excluding wrongly typed expressions. For instance, one might need both arithmetic and logical function symbols, e.g., to allow $(N = 2) \wedge (S > 80:000)$ as a correct expression. In this case it must be enforced that an arithmetic (logical) function symbol only has arithmetic (logical) arguments, e.g., to exclude $N \wedge 80:000$ as a correct expression. This issue is addressed in strongly typed Genetic Programming [10].

Before we go into the details of variation operators in GP, let us note that very often GP uses mutation OR crossover in one step. This is in contrast to GA and ES, where crossover (recombination) is performed, followed by mutation. That is, in those EA branches one uses crossover AND mutation in a (combined) step. This subtle difference is visible on the GP flowchart given in Figure 4 after Koza [5]. This chart compares the loop for filling the next generation in a generational GA with that of GP.

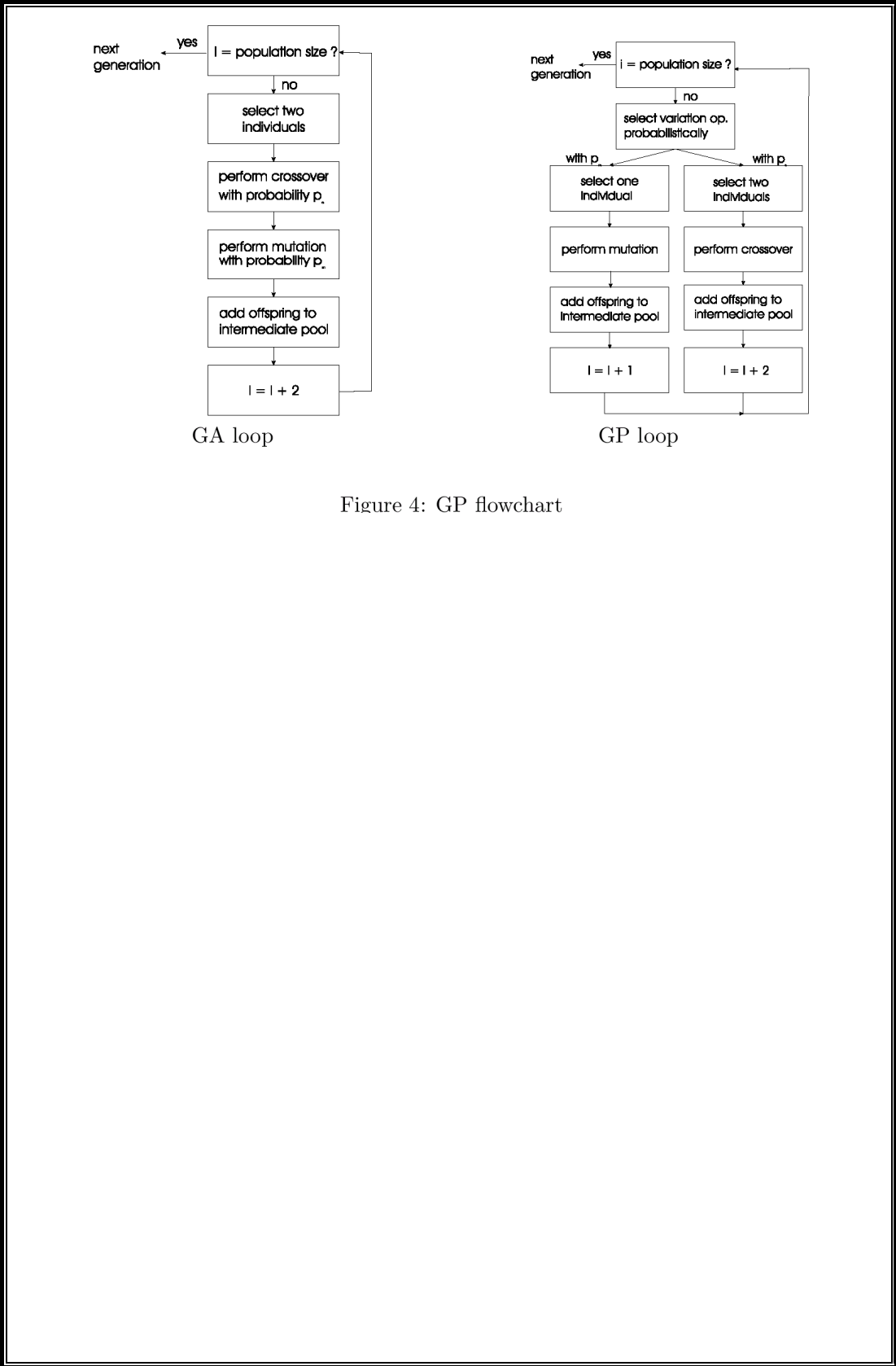


Figure 4: GP flowchart

4. MUTATION

In theory, the task of mutation in GP is the same as in all other EC branches, creating a new individual from an old one through some small random variation. The most common implementation works by replacing the subtree starting at a randomly selected node by a randomly generated tree. The newly created tree is usually generated the same way as in the initial population, see Section 8.

Note, that the size (tree depth) of the child can exceed that of the parent tree. Figure 5 illustrates how the parse tree belonging to the formula $1 - 2 + ((x + 3) - y)$ (left) is mutated into a parse tree standing for $2 - 1 + ((x + 3) - y)$

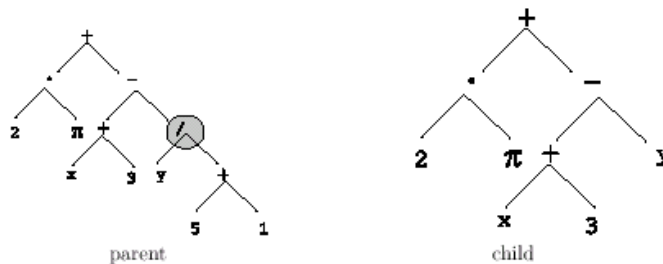


Figure 5: GP mutation illustrated: the node designated by a circle in the tree on the left is selected for mutation. The subtree starting at that node is replaced by a randomly generated tree, being a leaf here.

Mutation in GP has two parameters:

- The probability of choosing mutation at the junction with recombination,
- The probability of choosing an internal point within the parent as the root of the subtree to be replaced.

It is remarkable that Koza's classic book on GP from 1992, cf. [5], advises to set the mutation rate at 0, i.e., it suggests that GP works without mutation. More recently Banzhaf et al. suggested 5% [2]. In giving mutation such a limited role, GP differs from other EA streams. The reason for this is the generally shared view that crossover has a large shuffling effect, acting in some sense as a macromutation operator [1]. The current GP practice uses low, but positive, mutation frequencies, even though some studies indicate that the common wisdom favoring an (almost) pure crossover approach might be misleading [9].

5. RECOMBINATION

Recombination in GP creates offspring by swapping genetic material among the selected parents. In technical terms, it is a binary operator creating two child trees from two parent trees. The most common implementation is subtree crossover that works by interchanging the subtrees starting at a randomly selected node in the given parents. Note, that the size (tree depth) of the children can exceed that of the parent trees. In this, recombination within GP differs from recombination in other EC dialects.

During the development of GP many crossover operators have been offered. The most commonly used one, subtree crossover, is illustrated in Figure 6.

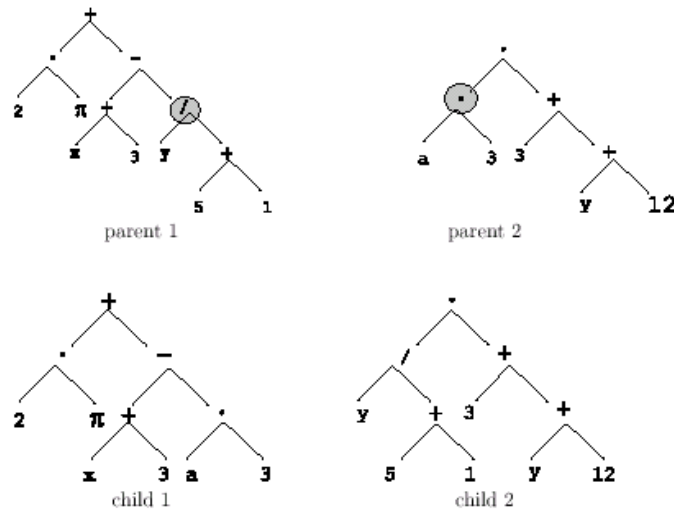


Figure 6: GP crossover illustrated: the nodes designated by a circle in the parent trees are selected to serve as crossover points. The subtrees starting at those nodes are swapped, resulting in two new trees, the children. Recombination in GP has two parameters:

- The probability of choosing recombination at the junction with mutation,
- The probability of choosing an internal point within each parent as crossover point.

Population Size	Proportion of population in fitter group (x)
1000	32 %
2000	16 %
4000	8 %
8000	4 %

Table 4: Proportion of ranked population in "_tter" sub-population from which majority of parents are selected.

6. PARENT SELECTION

GP typically uses fitness proportionate selection, however due to the large population sizes frequently used (population sizes of several thousands are not unusual), a method called over-selection is often used for population sizes of 1000 and above. In this method, the population is first ranked by fitness and then divided into two groups, one containing the top $x\%$ and the other containing the other $(100 - x)\%$. When parents are selected, 80% of the selection operations come from the first group and the other 20% from the second group. The values of x used have been found empirically by "rule of thumb" and depend on the population size as is shown in Table 4.

As can be seen the number of individuals from which the majority of parents are chosen stays constant, i.e. the selection pressure increases dramatically for larger populations.

7. SURVIVOR SELECTION

Traditional GP typically uses a generational strategy with no elitism, i.e. the number of off spring created is the same as the population size, and all individuals have a life-span of one generation. This is, of course, not a technical necessity, rather a convention. In their 1998 book Banzhaf et al. give an equal treatment to generational and steady-state GP [2] and the latest trend appears the need for elitism caused by the destructive effects of crossover [].

8. INITIALIZATION

The most common method of initialisation for GP is the so-called ramped half and half method. In this method a maximum initial depth D_{max} of trees, and the sets of functions (F) and terminals(T) are chosen, and then each member of the initial population is created using one of the two methods below with equal probability:

- Full Method: here each branch of the tree has depth D_{max} . The contents of nodes at depth d are chosen from F if $d < D_{max}$ or T if $d = D_{max}$.
- Grow Method: here the branches of the tree may have different depths, up to the limit D_{max} .The tree is constructed beginning from the head, with the contents of a node being chosenstochastically from F [T is $d < D_{max}$.

9. HOW GENETIC PROGRAMMING WORKS

Starting with a primordial ooze of thousands of randomly created computer programs, a population of programs is progressively evolved over a series of generations. The evolutionary search uses the Darwinian principle of survival of the fittest and is patterned after naturally occurring operations, including crossover (sexual recombination), mutation, gene duplication, gene deletion, and certain developmental processes by which embryos grow into fully developed organisms.

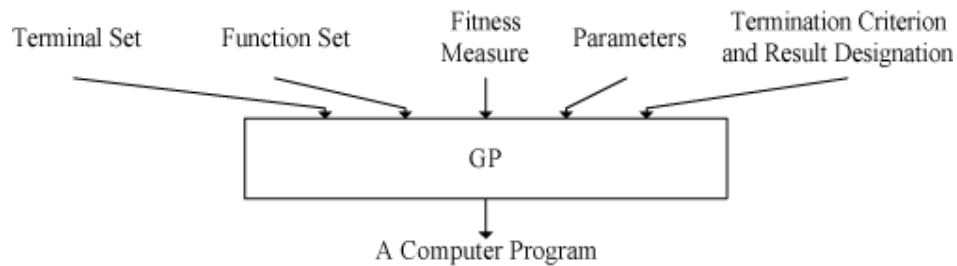
9.1 Preparatory Steps of Genetic Programming

Genetic programming starts from a high-level statement of the requirements of a problem and attempts to produce a computer program that solves the problem. The human user communicates the high-level statement of the problem to the genetic programming system by performing certain well-defined preparatory steps. The five major preparatory steps for the basic version of genetic programming require the human user to specify

- (1) The set of terminals (e.g., the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program,
- (2) The set of primitive functions for each branch of the to-be-evolved program,
- (3) The fitness measure (for explicitly or implicitly measuring the fitness of individuals in the population),
- (4) Certain parameters for controlling the run, and
- (5) The termination criterion and method for designating the result of the run.

The figure below shows the five major preparatory steps for the basic

version of genetic programming. The preparatory steps (shown at the top of the figure) are the human-supplied input to the genetic programming system. The computer program (shown at the bottom) is the output of the genetic programming system.



The first two preparatory steps specify the ingredients that are available to create the computer programs. A run of genetic programming is a competitive search among a diverse population of programs composed of the available functions and terminals.

Function Set and Terminal Set

The identification of the function set and terminal set for a particular problem (or category of problems) is usually a straightforward process. For some problems, the function set may consist of merely the arithmetic functions of addition, subtraction, multiplication, and division as well as a conditional branching operator. The terminal set may consist of the program's external inputs (independent variables) and numerical constants. This function set and terminal set is useful for a wide variety of problems (and corresponds to the basic operations found in virtually every general-purpose digital computer). For many other problems, the ingredients include specialized functions and terminals. For example, if the goal is to get genetic programming to automatically program a robot to mop the entire floor of an obstacle-laden

room, the human user must tell genetic programming what the robot is capable of doing. For example, the robot may be capable of executing functions such as moving, turning, and swishing the mop.

If the goal is the automatic creation of a controller, the function set may consist of signal-processing functions that operates on time-domain signals, including integrators, differentiators, leads, lags, gains, adders, subtractors, and the like. The terminal set may consist of signals such as the reference signal and plant output. Once the human user has identified the primitive ingredients for a problem of controller synthesis, the same function set and terminal set can be used to automatically synthesize a wide variety of different controllers. If a complex structure, such an antenna, is to be designed, it may be desirable to use functions that cause a turtle to draw the complex structure.

If the goal is the automatic synthesis of an analog electrical circuit, the function set may enable genetic programming to construct circuits from components such as transistors, capacitors, and resistors. This construction (developmental) process typically starts with a very simple embryonic structure, such as a single modifiable wire. If, additionally, such a function set is geographically aware, a circuit's placement and routing can be synthesized at the same as its topology and sizing. Once the human user has identified the primitive ingredients for a problem of circuit synthesis, the same function set and terminal set can be used to automatically synthesize an amplifier, computational circuit, active filter, voltage reference circuit, or any other circuit composed of these ingredients.

Fitness Measure

The third preparatory step concerns the fitness measure for the problem. The fitness measure specifies what needs to be done. The fitness measure is the primary mechanism for communicating the high-level statement of the problem's requirements to the genetic programming system. For example, if the goal is to get genetic programming to automatically synthesize an amplifier, the fitness function is the mechanism for telling genetic programming to synthesize a circuit that amplifies an incoming signal (as opposed to, say, a circuit that suppresses the low frequencies of an incoming signal or a circuit that computes the square root of the incoming signal). The first two preparatory steps define the search space whereas the fitness measure implicitly specifies the search's desired goal.

Control Parameters

The fourth and fifth preparatory steps are administrative. The fourth preparatory step entails specifying the control parameters for the run. The most important control parameter is the population size. In practice, the user may choose a population size that will produce a reasonably large number of generations in the amount of computer time we are willing to devote to a problem (as opposed to, say, analytically choosing the population size by somehow analyzing a problem's fitness landscape). Other control parameters include the probabilities of performing the genetic operations, the maximum size for programs, and other details of the run.

Termination

The fifth preparatory step consists of specifying the termination criterion and the method of designating the result of the run. The termination criterion may include a maximum number of generations to be run as well as a

problem-specific success predicate. In practice, one may manually monitor and manually terminate the run when the values of fitness for numerous successive best-of-generation individuals appear to have reached a plateau. The single best-so-far individual is then harvested and designated as the result of the run.

Running Genetic Programming

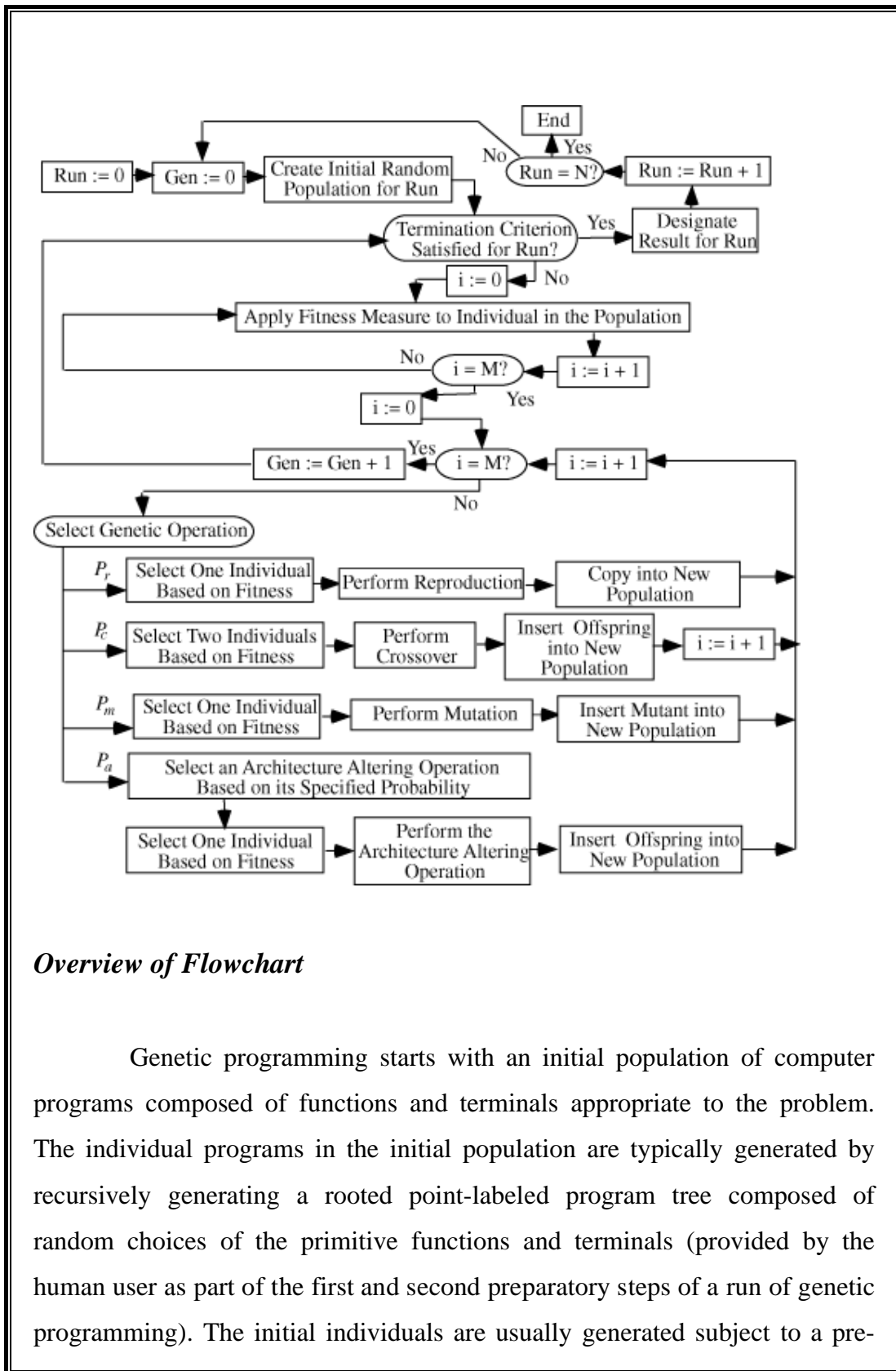
After the human user has performed the preparatory steps for a problem, the run of genetic programming can be launched. Once the run is launched, a series of well-defined, problem-independent executional steps (that is, the flowchart of genetic programming) is executed.

9.2 Executional Steps of Genetic Programming

Flowchart (Executional Steps) of Genetic Programming

Genetic programming is problem-independent in the sense that the flowchart specifying the basic sequence of executional steps is not modified for each new run or each new problem. There is usually no discretionary human intervention or interaction during a run of genetic programming (although a human user may exercise judgment as to whether to terminate a run).

The figure below is a flowchart showing the executional steps of a run of genetic programming. The flowchart shows the genetic operations of crossover, reproduction, and mutation as well as the architecture-altering operations. This flowchart shows a two-offspring version of the crossover operation.



Overview of Flowchart

Genetic programming starts with an initial population of computer programs composed of functions and terminals appropriate to the problem. The individual programs in the initial population are typically generated by recursively generating a rooted point-labeled program tree composed of random choices of the primitive functions and terminals (provided by the human user as part of the first and second preparatory steps of a run of genetic programming). The initial individuals are usually generated subject to a pre-

established maximum size (specified by the user as a minor parameter as part of the fourth preparatory step). In general, the programs in the population are of different size (number of functions and terminals) and of different shape (the particular graphical arrangement of functions and terminals in the program tree).

Each individual program in the population is executed. Then, each individual program in the population is either measured or compared in terms of how well it performs the task at hand (using the fitness measure provided in the third preparatory step). For many problems (including all problems in this book), this measurement yields a single explicit numerical value, called fitness. The fitness of a program may be measured in many different ways, including, for example, in terms of the amount of error between its output and the desired output, the amount of time (fuel, money, etc.) required to bring a system to a desired target state, the accuracy of the program in recognizing patterns or classifying objects into classes, the payoff that a game-playing program produces, or the compliance of a complex structure (such as an antenna, circuit, or controller) with user-specified design criteria. The execution of the program sometimes returns one or more explicit values. Alternatively, the execution of a program may consist only of side effects on the state of a world (e.g., a robot's actions). Alternatively, the execution of a program may produce both return values and side effects.

The fitness measure is, for many practical problems, multiobjective in the sense that it combines two or more different elements. The different elements of the fitness measure are often in competition with one another to some degree.

For many problems, each program in the population is executed over a representative sample of different fitness cases. These fitness cases may represent different values of the program's input(s), different initial conditions of a system, or different environments. Sometimes the fitness cases are constructed probabilistically.

The creation of the initial random population is, in effect, a blind random search of the search space of the problem. It provides a baseline for judging future search efforts. Typically, the individual programs in generation 0 all have exceedingly poor fitness. Nonetheless, some individuals in the population are (usually) more fit than others. The differences in fitness are then exploited by genetic programming. Genetic programming applies Darwinian selection and the genetic operations to create a new population of offspring programs from the current population.

The genetic operations include crossover (sexual recombination), mutation, reproduction, and the architecture-altering operations. These genetic operations are applied to individual(s) that are probabilistically selected from the population based on fitness. In this probabilistic selection process, better individuals are favored over inferior individuals. However, the best individual in the population is not necessarily selected and the worst individual in the population is not necessarily passed over.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the current population (i.e., the now-old generation). This iterative process of measuring fitness and performing the genetic operations is repeated over many generations.

The run of genetic programming terminates when the termination criterion (as provided by the fifth preparatory step) is satisfied. The outcome of the run is specified by the method of result designation. The best individual ever encountered during the run (i.e., the best-so-far individual) is typically designated as the result of the run.

All programs in the initial random population (generation 0) of a run of genetic programming are syntactically valid, executable programs. The genetic operations that are performed during the run (i.e., crossover, mutation, reproduction, and the architecture-altering operations) are designed to produce offspring that are syntactically valid, executable programs. Thus, every individual created during a run of genetic programming (including, in particular, the best-of-run individual) is a syntactically valid, executable program. There are numerous alternative implementations of genetic programming that vary from the foregoing brief description.

Creation of Initial Population of Computer Programs

Genetic programming starts with a primordial ooze of thousands of randomly-generated computer programs. The set of functions that may appear at the internal points of a program tree may include ordinary arithmetic functions and conditional operators. The set of terminals appearing at the external points typically include the program's external inputs (such as the independent variables X and Y) and random constants (such as 3.2 and 0.4). The randomly created programs typically have different sizes and shapes.

Main Generational Loop of Genetic Programming

The main generational loop of a run of genetic programming consists of the fitness evaluation, Darwinian selection, and the genetic operations. Each individual program in the population is evaluated to determine how fit it is at solving the problem at hand. Programs are then probabilistically selected from the population based on their fitness to participate in the various genetic operations, with reselection allowed. While a more fit program has a better chance of being selected, even individuals known to be unfit are allocated some trials in a mathematically principled way. That is, genetic programming is not a purely greedy hill-climbing algorithm. The individuals in the initial random population and the offspring produced by each genetic operation are all syntactically valid executable programs. After many generations, a program may emerge that solves, or approximately solves, the problem at hand.

Mutation Operation

In the mutation operation, a single parental program is probabilistically selected from the population based on fitness. A mutation point is randomly chosen, the subtree rooted at that point is deleted, and a new subtree is grown there using the same random growth process that was used to generate the initial population. This asexual mutation operation is typically performed sparingly (with a low probability of, say, 1% during each generation of the run).

Crossover (Sexual Recombination) Operation

In the crossover, or sexual recombination operation, two parental programs are probabilistically selected from the population based on fitness. The two parents participating in crossover are usually of different sizes and

shapes. A crossover point is randomly chosen in the first parent and a crossover point is randomly chosen in the second parent. Then the subtree rooted at the crossover point of the first, or receiving, parent is deleted and replaced by the subtree from the second, or contributing, parent. Crossover is the predominant operation in genetic programming (and genetic algorithm) work and is performed with a high probability (say, 85% to 90%).

Reproduction Operation

The reproduction operation copies a single individual, probabilistically selected based on fitness, into the next generation of the population.

Architecture-Altering Operations

Simple computer programs consist of one main program (called a result-producing branch). However, more complicated programs contain subroutines (also called automatically defined functions, ADFs, or function-defining branches), iterations (automatically defined iterations or ADIs), loops (automatically defined loops or ADLs), recursions (automatically defined recursions or ADRs), and memory of various dimensionality and size (automatically defined stores or ADSs). If a human user is trying to solve an engineering problem, he or she might choose to simply prespecify a reasonable fixed architectural arrangement for all programs in the population (i.e., the number and types of branches and number of arguments that each branch possesses). Genetic programming can then be used to evolve the exact sequence of primitive work-performing steps in each branch.

However, sometimes the size and shape of the solution is the problem (or at least a major part of it). Genetic programming is capable of making all architectural decisions dynamically during the run of genetic programming. Genetic programming uses architecture-altering operations to automatically determine program architecture in a manner that parallels gene duplication in nature and the related operation of gene deletion in nature. Architecture-altering operations provide a way, dynamically during the run of genetic programming, to add and delete subroutines and other types of branches to individual programs to add and delete arguments possessed by the subroutines and other types of branches. These architecture-altering operation quickly create an architecturally diverse population containing programs with different numbers of subroutines, arguments, iterations, loops, recursions, and memory and, also, different hierarchical arrangements of these elements. Programs with architectures that are well-suited to the problem at hand will tend to grow and prosper in the competitive evolutionary process, while programs with inadequate architectures will tend to wither away under the relentless selective pressure of the problem's fitness measure. Thus, the architecture-altering operations relieve the human user of the task of prespecifying program architecture.

There are several different architecture-altering operations (described below). They are each applied sparingly during the run (say, with a probability of 1/2% of 1% on each generation). The subroutine duplication operation duplicates a preexisting subroutine in an individual program gives a new name to the copy and randomly divides the preexisting calls to the old subroutine between the two. This operation changes the program architecture by broadening the hierarchy of subroutines in the overall program. As with gene duplication in nature, this operation preserves semantics when it first occurs. The two subroutines typically diverge later, sometimes yielding specialization.

The argument duplication operation duplicates one argument of a subroutine, randomly divides internal references to it, and preserves overall program semantics by adjusting all calls to the subroutine. This operation enlarges the dimensionality of the subspace on which the subroutine operates.

The subroutine creation operation can create a new subroutine from part of a main result-producing branch thereby deepening the hierarchy of references in the overall program, by creating a hierarchical reference between the main program and the new subroutine. The subroutine creation operation can also create a new subroutine from part of an existing subroutine further deepening the hierarchy of references, by creating a hierarchical reference between a preexisting subroutine and a new subroutine and a deeper and more complex overall hierarchy. The architecture-altering operation of subroutine deletion deletes a subroutine from a program thereby making the hierarchy of subroutines either narrower or shallower.

The argument deletion operation deletes an argument from a subroutine thereby reducing the amount of information available to the subroutine, a process that can be viewed as generalization. Other architecture-altering operations add and delete automatically defined iterations, automatically defined loops, automatically defined recursions, and automatically defined stores (memory).

Genetic programming typically starts with a population of randomly generated computer programs composed of the available programmatic ingredients. Genetic programming iteratively transforms a population of computer programs into a new generation of the population by applying analogs of naturally occurring genetic operations. These operations are applied to individual(s) selected from the population. The individuals are

probabilistically selected to participate in the genetic operations based on their fitness (as measured by the fitness measure provided by the human user in the third preparatory step). The iterative transformation of the population is executed inside the main generational loop of the run of genetic programming. The executional steps of genetic programming (that is, the **flowchart** of genetic programming) are as follows:

- (1) Randomly create an initial population (generation 0) of individual computer programs composed of the available functions and terminals.
- (2) Iteratively perform the following sub-steps (called a generation) on the population until the termination criterion is satisfied:
 - (a) Execute each program in the population and ascertain its fitness (explicitly or implicitly) using the problem's fitness measure.
 - (b) Select one or two individual program(s) from the population with a probability based on fitness (with reselection allowed) to participate in the genetic operations in (c).
 - (c) Create new individual program(s) for the population by applying the following genetic operations with specified probabilities:
 - (d) **Reproduction:** Copy the selected individual program to the new population.
 - (e) **Crossover:** Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.
 - (f) **Mutation:** Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.
 - (g) **Architecture-altering operations:** Choose an architecture-altering operation from the available repertoire of such operations and create one new offspring program for the new population by applying the chosen architecture-altering operation to one selected program.

(h) After the termination criterion is satisfied, the single best program in the population produced during the run (the best-so-far individual) is harvested and designated as the result of the run. If the run is successful, the result may be a solution (or approximate solution) to the problem.

10. DEVELOPMENTAL GENETIC PROGRAMMING

Our approach to circuit synthesis using genetic programming draws on an additional analogy from nature — the process by which an embryo develops into a fully grown organism.

Gifs Showing Development of the Topology and Sizing of an Electrical

A fully developed electrical circuit can be produced by progressively executing circuit-constructing functions from an individual program tree from the population. Each circuit-constructing program tree in the population contains component-creating functions, topology-modifying functions, and development-controlling functions that grow a simple embryo into a fully developed circuit. . The process starts with a very simple embryonic circuit. In this example, the embryo consists of two modifiable wires, Z0 and Z1. The embryo is embedded in a test fixture whose input is the electrical signal VSOURCE and whose output is the voltage VOUT. The test fixture also contains a source resistor and a load resistor. The output produced by these two embryonic wires sitting in their test fixture is always zero, and uninteresting. Execution of the circuit-constructing program tree begins with the capacitor-creating C function associated with modifiable wire Z0 inserting capacitor C3 in lieu of modifiable wire Z0. The seven-point arithmetic-performing subtree of the C function establishes the numerical value of 403 nano-Farads as the sizing of capacitor C3. The next function in the tree — an S function — now becomes associated with C3. The polarity-reversing flip function F now reverses the polarity of modifiable wire Z1. The topology-modifying series, or S, function now creates a series composition consisting of two copies of the capacitor and new modifiable wire Z4. These three new

components become associated with an F, S, and L function. The inductor-creating L function inserts inductor L8 into the growing circuit. Its arithmetic subtree establishes .22 micro-Henries as the value of inductor L8. This flip function F reverses the polarity of capacitor C3. The just-flipped capacitor C3 then becomes associated with the E function. The second series function then trifurcates modifiable wire Z4 into three modifiable wires. Another inductor-creating L function converts capacitor C5 into inductor L7. The three-point arithmetic-performing subtree assigns a value of 0.41 micro-Henries to new inductor L7. The end function E at the far right of the screen then ends development of its particular path, as does this end function and other similar end functions later. Modifiable wire Z4 is flipped. Inductor L9 is inserted in lieu of modifiable wire Z6 ... and inductor L7 is changed in value to .753 micro-Henries. Continuing in the same fashion, we get a fully developed electrical circuit. Other topology-modifying functions are available, including **parallel division**. Transistors can also be inserted into a developing circuit by means of a **transistor-creating function**. Since most present-day electrical circuits are composed of transistors, capacitors, and resistors (on silicon chips), this development process is very general.

Animated Gif Showing Development of the Placement and Routing Along with the Topology and Sizing of an Electrical Circuit. A circuit's placement and routing can be synthesized at the same time as its topology and sizing by using geographically-aware circuit-constructing functions. The geographically-aware functions keep track of the geographic location of each component and wire during the developmental process. . This process starts with a modifiable wire, Z, at a particular geographic location on the circuit board or chip, and an incoming signal, V, a source resistor, and a load resistor. The circuit-constructing program tree contains component-creating functions, topology-modifying functions, development-controlling functions, and

numerical constants and arithmetic-performing subtrees that establish component values. The process starts with induction-creating function inserting L2 in lieu of a modifiable wire, while appropriately adjusting the locations of preexisting wires and components. Then, a parallel division duplicates inductor L2 and creates two new modifiable wires. Two polarity-reversing FLIP function and one development-controlling END function bring this path through the program tree to an end. A capacitor-creating function converts an inductor into a capacitor, whose component value is established by a seven-point arithmetic-performing subtree. This SERIES function divides a modifiable wire. A capacitor-creating function inserts a capacitor in lieu of a modifiable wire. A topology-modifying function creates a connection to ground, and this inductor-inserting function converts a modifiable wire into a final inductor. The result is the topology, sizing, placement, and routing of a fully developed circuit.

Design of a Complex Structure Using a Turtle

The topology and sizing of structures, such as antennas, can be automatically synthesized by means of developmental genetic programming. For example, a two-dimensional wire antenna can be constructed by a turtle that starts at a particular point facing in a certain direction, and drops metal while moving a certain distance, then turns to a new direction, moves a certain distance without dropping metal, then turns again, and moves in the new direction while dropping metal, and again turns and moves while dropping metal, yielding this final small antenna. Three-dimensional antennas can be created by employing a **flying turtle**. Other types of complex structures can be similarly created using Lindenmayer systems, the LOGO programming language, and other grammatical and developmental techniques.

11. APPLICATIONS OF GENETIC PROGRAMMING

There are numerous applications of genetic programming. For example, Genetic Programming Inc. is currently doing research and development aimed at producing human-competitive results, with emphasis on applications such as :

- “black art” problems, such as the automated synthesis of analog electrical circuits, controllers, antennas, and other areas of design,
- “programming the unprogrammable” (PTU) involving the automatic creation of computer programs for unconventional computing devices such as cellular automata, parallel systems, multi-agent systems, and the like,
- “commercially usable new inventions” (CUNI) involving the use of genetic programming as an automated "invention machine" for creating commercially usable new inventions, and
- problems in computational molecular biology, including metabolic pathways and genetic networks.

12. CONCLUSION

Genetic programming now routinely delivers high-return human-competitive machine intelligence. There are now 36 instances where genetic programming has automatically produced a computer program that is competitive with human performance, including 15 instances where genetic programming has created an entity that either infringes or duplicates the functionality of a previously patented 20th-century invention, 6 instances where genetic programming has done the same with respect to a 21st-century invention, and 2 instances where genetic programming has created a patentable new invention.

The fact that genetic programming can evolve entities that are competitive with human-produced results suggests that **genetic programming can be used as an automated invention machine** to create new and useful patentable inventions. In acting as an invention machine, evolutionary methods, such as genetic programming, have the advantage of not being encumbered by preconceptions that limit human problem-solving to well-trodden paths.

13. REFERENCES

1. Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Chapter 1 by John R. Koza (Stanford University), Martin A. Keane (Econometrics Inc.), Matthew J. Streeter (Genetic Programming Inc.), William Mydlowec (Pharmix Inc.), Jessen Yu (Pharmix Inc.), Guido Lanza (Pharmix Inc.)
2. Salon article on "**Software that Writes Software**" by Alexis Willihnganz (August 10, 1999)
3. For May 2003 *IEEE Intelligent Systems* article "What's AI done for me lately? Genetic programming's human-competitive results", visit IEEE Intelligent Systems.
4. For *the451.com* article entitled "**Re-inventing the 'invention machine'**" (April 14, 2000).
5. www.genetic-programming.com

CONTENTS

1. Introduction	1
2. Introductory Example	2
3. Representation	4
4. Mutation	8
5. Recombination	10
6. Parent selection	12
7. Survivor selection	13
8. Initialization	14
9. How Genetic Programming Works	15
9.1 Preparatory Steps of Genetic Programming	
9.2 Executional Steps of Genetic Programming	
10. Developmental Genetic Programming	29
11. Applications of Genetic Programming	32
12. Conclusion	33
13. References	34

ABSTRACT

Genetic programming (GP) is an automated method for creating a working computer program from a high-level problem statement of a problem.

Starting with a primordial ooze of thousands of randomly created computer programs, a population of programs is progressively evolved over a series of generations. The evolutionary search uses the Darwinian principle of survival of the fittest and is patterned after naturally occurring operations, including crossover (sexual recombination), mutation, gene duplication, gene deletion, and certain developmental processes by which embryos grow into fully developed organisms. There are now 36 instances where genetic programming has automatically produced a computer program that is competitive with human performance

ACKNOWLEDGEMENT

I extend my sincere thanks to **Prof. P.V.Abdul Hameed**, Head of the Department for providing me with the guidance and facilities for the Seminar.

I express my sincere gratitude to Seminar coordinator **Mr. Berly C.J**, Staff in charge, for their cooperation and guidance for preparing and presenting this seminar.

I also extend my sincere thanks to all other faculty members of Electronics and Communication Department and my friends for their support and encouragement.

ANUJ. A