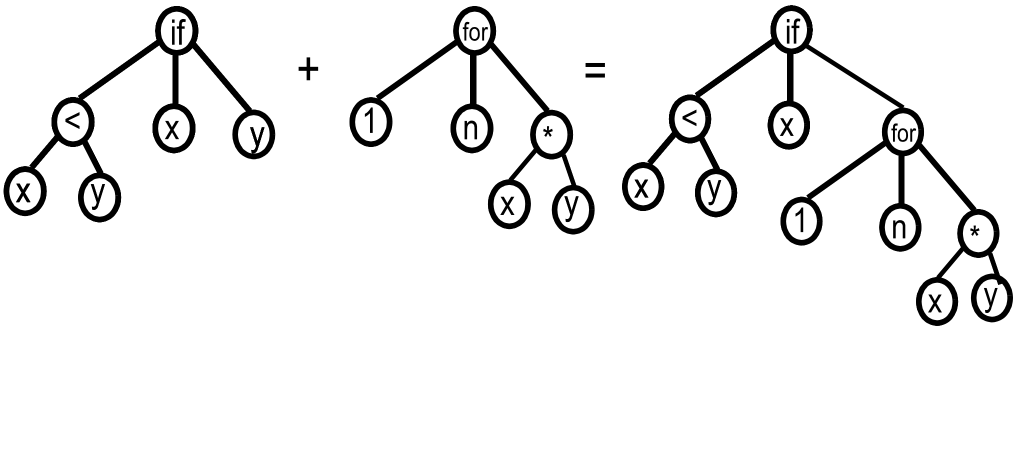
An Introduction to Genetic Programming



Chapter 1 - An Introduction to Genetic Programming

1. The Genetic Programming Algorithm

The basic genetic programming algorithm:

- Create an initial population.
- Repeat
    - Evaluate each individual in the population.
    - Select individuals to apply the genetic operators to.
    - Create new individuals by application of the genetic operators.

  Until the termination criterion has been met.
- The individual representing the best solution in a generation is returned as the result of the generation.

The repeat loop in the algorithm is referred to as a generation. A genetic programming run usually consists of a number of generations.

The following preparatory steps need to be performed

- Select the terminal and function sets.
- Specify a fitness function.
- Specify parameters such as the population size, the number of generations in a run, maximum individual size.
- Specify probabilities indicating the application rate of each genetic operator.
- Choose a selection method.
- Define the termination criteria of a run and how the result of a run will be designated.

The genetic programming run comes to an end when either a specified number of generations have been reached or when a success predicate has been satisfied. A success predicate may specify that a hundred percent solution must be found or that a “near” solution must be found.

The result designated by the algorithm is either the best individual of the last generation or the best individual over all generations. In some cases all the individuals of the last generation maybe designated as the result of a run.

2. Control Models

One of three control models can be employed by a genetic programming system

2.1 The Generational Control Model

This is the control model that is traditionally used by GP systems. There are a distinct number of generations performed by the algorithm. In each generation there is a complete population of individuals. The size of the population remains constant throughout a run of the genetic programming system. The new population is created from the old population which it replaces by applying genetic operators to the individuals.

2.2 The Steady-state Control Model

This model does not implement a fixed number of generations. A single population of a fixed size is maintained during a genetic programming run. Newly created individuals replace the individuals with poor fitness in the population. Inverse selection methods are used for choosing which members of the population to replace.

2.3 Varying Population-Size Control

In this case a single population is maintained, the size of which changes throughout a GP run. The GAVaPS is one of the many algorithms that implements this model. This algorithm enables the population size to grow and shrink according to the state of the search.

If the fitness is high then the population grows. If the neighborhood of an optimal solution is found the population size is decreased. In the neighborhood of a local optimum the size of the population is increased.

3. Functions and Terminals

Each potential solution in a GP population is represented as a parse tree. Each parse tree is composed of elements from the function set F= {f1,f2,..,fn}, where n is the number of functions, and elements of the terminal set T = {t1,t2,..,tm}, where m is the number of terminals. These are collectively referred to as primitives and form a representation language for expressing programs in. Each element of the function set takes a specified number of arguments. We refer to this as arity of the function. The set A= {a1,a2,..,an} consists of the corresponding arity for each member of the function set.

3.1 Functions

The following examples of elements that are contained in function sets are

- Arithmetic functions: +, -, *.
- Conditional operators: If-then-else.
- Variable assignment functions : ASSIGN.

- Loop statements: WHILE...DO; REPEAT...UNTIL, FOR...DO.
- Depending on each problem certain domain specific functions may be defined.

3.2 Terminals

Elements of a terminal set include:

- Variables representing the input to functions, or state variables of a system
- Single constants.
- Random ephemeral constants that remain constant throughout a genetic programming run. An example of an ephemeral constant is $\Re$ which represents a range of real numbers.
- Functions with an arity of zero that have certain side effects.

3.3 Closure and Sufficiency

The function set and the terminal sets chosen must satisfy the closure property and the sufficiency property.

3.3.1 The Closure Property

A function set satisfies the closure property if each function in the set accepts any value that the elements in the function set may return as well as any value that each member of the terminal set may take as input.

In some problems in order to satisfy the closure property some functions may be defined as protected functions. For example, the division operator cannot accept a denominator of zero. Similarly, the square root operator cannot accept negative values. In both these cases the operators are redefined as protected functions. These protected functions either return a value of one or an error message if an unacceptable input value is passed to them.

3.3.2 The Sufficiency Property

The sufficiency property is satisfied if the solution to the problem can be expressed in terms of the members of the function set and the terminal set. Extraneous functions and terminals degrade population performance.

4. The Tree Representation

Each individual in the population is represented as a parse tree as illustrated in **Figure 4.1**.

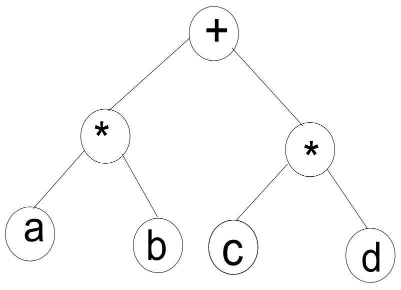


Figure 4.1: A tree representation of a program

A program is executed by firstly performing a prefix or postfix traversal of the tree. The results of the traversal is then interpreted by an interpreter built into the system or compiled and executed. A prefix traversal has an advantage over a postfix traversal in that the execution time can be reduced when executing a conditional statement such as an if-else statement.

For purposes of this course the tree structure will be used to represent each individual in a population. The AIMGP genetic programming system uses a linear structure consisting of a set of instructions that can be executed in a top to bottom or left to right manner. GP systems using a linear structure have proven to converge to a solution much quicker than those using the tree or graph structure to represent individuals. The PADO system represents each individual as a graph. The graph structure is suitable for representing iteration and recursion.

5. Initial Population Generation

The initial population essentially consists of parse trees representing each potential solution program. These parse trees are created by randomly choosing elements from a combination of the function set and terminal set. A uniform random distribution is used to make random choices.

The maximum tree size must be specified. This maximum is either the maximum depth of the tree or the maximum number of nodes a tree can be composed of. The depth of a node is the number of nodes on the path from the root of the tree to a particular node in a tree.

There are three different methods that can be used to generate the initial population: the full method, the grow method and the ramped half-and-half method.

5.1 The Full Method

This method ensures that the length of every path between an endpoint and the root of each tree is equal to the specified maximum depth.

Labels for nodes at a depth less than the maximum depth are chosen from the function set at random. Labels for nodes at a depth equal to the maximum depth are chosen from the terminal set.

5.2 The Grow Method

Trees generated using this method are of variable shape. In each individual produced the length of a path between the root and an endpoint is less than or equal to the maximum depth. Labels for nodes at a level less than the maximum depth are randomly chosen from a combination of the terminal and function set. Labels for the nodes at a level equal to the maximum depth are randomly chosen from the terminal set.

5.3 Ramped Half-and-Half

This method produces trees of various shapes and sizes. It combines the full and grow methods. A depth parameter ranging between two and a maximum specified depth is set. An equal number of trees of each depth value in the range is generated. For example, if the maximum depth is six then 20% of the trees will have a depth of two, 20% a depth of three, 20% a depth of four, 20% a depth of five, and 20% a depth of six. For each depth value 50% of the trees will be generated using the full method and 50% using the grow method.

GP systems using the ramped half-and-half method have been found to produce better results than those implementing either of the other methods.

The initial population can be seeded with sub-optimal solutions. A grammar can possibly be used for this purpose.

6. Variety

It is essential that the genetic diversity of the population is maintained in order to prevent the premature convergence of the GP system to a sub-optimal solution. The **variety** of a population is the percentage of the population for which there are no duplicates in the population. In order to promote genetic diversity it is essential that the variety of the initial population is a hundred percent.

7. Fitness Function

A fitness function must be defined. This function will be used to evaluate each individual and assign it a fitness measure. The selection methods use these fitness values to determine which individuals to choose to apply the genetic operators to.

Fitness functions are usually used to measure:

- How well the individual actually solves the problem.
- A minimization of the size of the programs induced.
- A minimization of the run time of programs induced.
- A minimization of the cost of solving the problem.

Multi-objective fitness functions can be defined to test for more than one objective, e.g. the length of the evolved program and the execution speed of the program as part of the fitness evaluation.

Four fitness measures commonly used are:

- Raw fitness
- Standardized fitness
- Adjusted fitness
- Normalized fitness

7.1 Raw fitness

A common form of fitness is the fitness error. The raw fitness error is defined as the sum of the differences between the output value of a fitness case[1] and the result of evaluating the individual using the input values as the corresponding arguments.

The result of a program can be Boolean-valued, integer-valued, floating-point valued, or symbolic-valued. For integer and floating point values the absolute values of the distances are summed.

The raw fitness is calculated using the following equation presented by Koza :

$$r(i,t) = \sum_{j=1}^{N} |s(i,j) - C(j)|$$

where: s(i, j) is the value returned by the program i for fitness case j.

N is the number of the fitness cases

C(j) is the correct value of the fitness case j.

If the program is Boolean-valued or symbolic-valued the sum of the distances is equal to the number of mismatches. If the program is complex-valued the sum of the distances for each component is computed. These are then summed.

[1] The concept of a fitness case is defined in the next section.

Depending on the problem a better raw fitness may be a smaller value, e.g. the fitness error or a bigger value, e.g. the amount found eaten by the artificial ant in the artificial ant problem.

7.2 Standardized Fitness

Standardized fitness s(i,t) describes the raw fitness so that a lower numerical value is always better. The best value of the standardized fitness is zero. In problems where a smaller fitness value means a better fitness the standardized fitness and the raw fitness are equal, i.e. s(i,t) = r(i,t).

In problems where a larger value is a better fitness the standardized fitness is calculated by subtracting the raw fitness from the maximum raw fitness value that can be attained.

The adjusted and the normalized fitness is calculated directly from the raw fitness if the raw fitness does not have an upperbound.

7.3 Adjusted Fitness

The adjusted fitness is calculated using the standardized fitness as follows:

$$a(i,t) = \frac{1}{1+s(i,t)}$$

where s(i, t) is the standardized fitness of program *i* at time *t*.

The range of the adjusted fitness is zero to one. The adjusted fitness is higher for fitter individuals of the population.

The adjusted fitness measure is only used if the method of selection of individuals for the next generation is fitness proportionate selection[2]. The adjusted fitness can be used to distinguish between a good individual and a very good individual.

7.4 Normalized Fitness

This fitness measure is used if the method of selection of individuals is fitness proportionate selection. The normalized fitness is calculated from the adjusted fitness as follows:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$$

[2]Selection methods are described in Section 9.

The normalized fitness lies between zero and one. It is larger for fitter individuals in the population. The sum of the normalized fitness values for each individual in the population is one.

8. Fitness Cases

Fitness cases are input-output pairs describing the target output of a program given the input value specified in the fitness case. An individual's fitness is usually calculated by applying the program it represents to a set of fitness cases. This is referred to as the training set. Once a solution has been derived it can be tested using a validation (also called the testing set) set.

An entire problem domain is usually very large or infinite. Fitness cases are a representational sample of the entire problem domain. In small domains, e.g. Boolean functions it may be possible to use combinations of all the arguments of a function. In cases where the number of fitness cases is large or infinite a sample of the fitness cases must be taken. Statistical methods can be employed to effectively select fitness cases.

In order to reduce the effect of using a particular set of fitness cases, different fitness cases may be chosen for each generation in a run.

9. Selection Methods

As mentioned previously selection methods are used to choose individuals to which the different genetic operators will be applied. Three methods which are most commonly used are:

- Fitness proportionate selection
- Tournament selection
- Rank selection

9.1 Fitness Proportionate Selection Method

Fitness proportionate selection is applied as follows:

- For each individual calculate the probability that the individual will be copied into the next generation using the formula $\dfrac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t))}$ where $s_i(t)$ is the adjusted fitness value of an individual.
- Create a mating pool.

- Randomly select an individual, using a uniform random distribution, from the mating pool to apply the genetic operators to. One possibility is that a number between 0.0 and 1.0 be randomly selected and select the individual with the corresponding probability.

If a GP system has employed the steady state control model inverse proportionate selection has to be performed to choose those individuals with the poorest fitness. The above algorithm is implemented where the following formula is used to calculate the necessary probabilities:

$$1.0 - \frac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t))}$$

9.2 Tournament Selection

The tournament selection method is implemented as follows:

- A group of individuals, usually two or more, are chosen at random from the population. The number of individuals in the group is referred to as the tournament size.
- The fitness value is calculated for each individual.
- The individual with the best fitness is chosen.

Selection is performed with replacement. Selection pressure measures the extent to which a selection method biases the selection process toward fitter individuals.
The higher the tournament size the higher the selection pressure.

Inverse tournament selection maybe needed if the steady-state control model is employed. Inverse tournament selection can be used to locate those individuals of the population with poor fitness. This involves choosing the individual with worst fitness in the tournament. This individual is then replaced with newly created individuals.

9.3 Rank Selection

Rank selection uses a qualitative rather than a quantitative measure to choose individuals. Each individual is assigned a rank based on their fitness measure. This method provides an easy distinction between closely clustered fitness values and in doing so enables individuals with better values to be more easily sampled. The two types of ranking commonly used are **linear** ranking and **exponential** ranking.

9.3.1 Linear Selection

The basic linear selection algorithm:

- Calculate the fitness of all the individuals in the population.
- Compute the proportional selection probability(fitness value/sum of fitness values) value for each individual.
- Sort the individuals in descending order according to their fitness values.
- Calculate the linear ranking selection probability for each individual:
- Calculate f(i) =ai +b where a and b are constants and i is the index of the individual.
- Calculate $\frac{f(i)}{\sum_{j=1}^{n} f(j)}$ for each individual. This is called the linear ranking selection probability (RSP).
- Multiply the RSP for each individual by the number of elements in the population to determine how many places each individual will have in the population of the next generation.

Different formulae can be used to calculate the RSP value.

9.3.2 Exponential Ranking

The algorithm used for linear ranking is implemented with the formula for calculating linear ranking selection probabilities replaced with the following formula for calculating exponential ranking selection probabilities.

The exponential ranking selection probabilities are calculated using the following formula $p_i(selection) = \frac{S^i}{\sum_{j=0}^{n-1} s^j}$ where i is the index of the individual in the sorted population, n is the number of elements in the population and s is the selection pressure parameter, $0 < s < 1$. As s decreases the selection pressure of the algorithm increases.

Different formulae can be used to calculate the ranking selection probabilities.

9.4 Truncation or (μ, λ) Selection

Each individual in the population reproduces a number of times, usually by mutation, until λ offspring are created, where λ > μ. The new population contains the best μ of λ offspring. (μ , λ+μ) is a variation of this method in which the μ best individuals are chosen from the union of the old population and the new offspring.

10. Genetic Operators

Genetic operators are applied to members of the population of a generation to create the population for the next generation. The most commonly used genetic operators are the reproduction, crossover and mutation operators. An application rate for each operator must be specified as part of the system parameters.

10.1 Reproduction

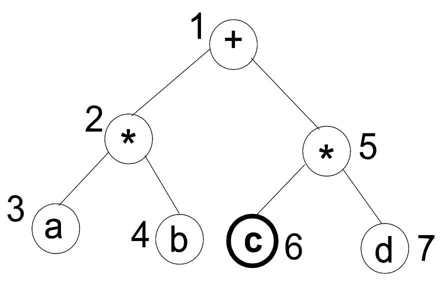
The reproduction operator is applied to a single individual. An individual is randomly selected from the population using one of the selection methods. This individual is then copied into the population of the next generation. The reproduction reduces the amount of computer time required by eliminating the need to recalculate the fitness value of the individual when it is added to the new population.

10.2 Crossover

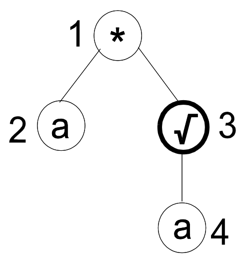
The crossover operator is applied as follows:

- Two parents are selected from the current population using one of the selection methods described below.
- A random point is selected in each of the parents using a uniform random number. This point is referred to as a crossover point. The subtrees rooted at these points are called crossover fragments.
- Both the crossover fragments are swapped generating two new offspring.

Example:

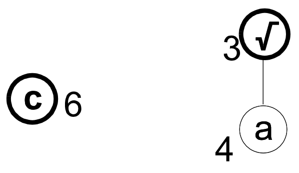


Parent 1



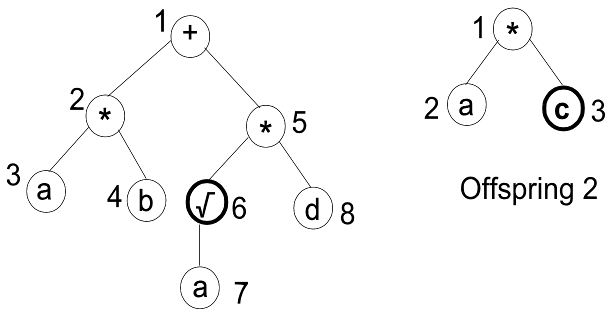
Parent 2

The nodes of each tree are numbered in a depth-first left-to-right manner. The crossover points selected are six in the first parent and three in the second parent. Thus, the crossover fragments are:



Fragment 1 Fragment 2

The corresponding offspring are:



Offspring 1

Some variations of the crossover operator include the production of just one offspring and n-point crossover (n is an integer with a floor value of one).

If terminals are located at both crossover points in both the parents then the terminals are merely swapped. If both the terminals are the same this operation has no effect and the offspring are identical to the parents. If the terminals are different then this operation has the same effect as the mutation operator.

In the case where the root of both parents are chosen as crossover points then the crossover operation has the same effect as the reproduction operation.

A maximum size, either the depth of the individual or the number of nodes, is specified. If the size of an individual produced exceeds this maximum size one of the following approaches can be taken:

- Choose only the one offspring, i.e. the one that is legal.
- Reselect the crossover points until both the children produced are legal.
- Pruning over-sized trees.
- If an offspring is over-sized replace it with one of its parents.

10.3 Mutation

The mutation operator involves making random changes to an individual in the population.

An individual is selected from the population using one of the selection methods. A mutation point is selected at random. This point can be an internal point, i.e. a function or an external point, i.e. a terminal point. The subtree at the mutation point is removed. A randomly generated subtree is inserted at the mutation point.

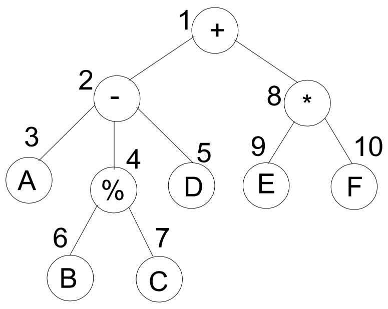
A special case of the mutation operator involves inserting a single terminal at a randomly selected position in the tree. Point mutation involves adding or deleting terminal nodes. The mutation operator can also be used to replace a function node with another legal function node.

The mutation operator is often used to maintain the genetic diversity of a population.

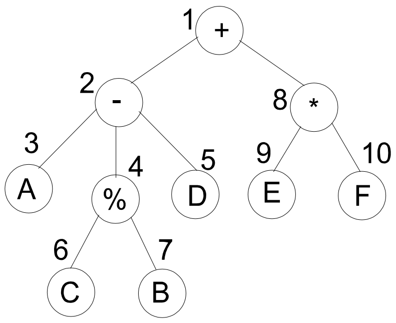
10.4 Permutation

This operator involves choosing one parent from the population using one of the selection methods described above. The permutation operator produces a single offspring. A function node in the parent selected is randomly chosen. A permutation of the arguments of the function is randomly selected. If a function has n arguments then n! permutations of these argument exists.

Example :



Suppose that four is chosen as a permutation point. The resulting offspring is:



10.5 Editing

The editing operation is used to edit and simplify programs. This operator is applied to one parent. The result of the operation is one offspring.

This operation recursively applies a set of editing rules to the program. These editing rules are domain-independent.

The Universal Independent Editing Rule states that if a subtree representing a function has no side effects and is composed of only constants as arguments then the editing operation will evaluate the function and replace the function with the result of the evaluation.

Examples: (+ 1 2) will be replaced with 3
(AND T T) will be replaced withT.

Domain-specific editing rules may also be applied.

The editing operation is time consuming. A frequency parameter, $f_{ed,}$ specifies whether the editing operator must be applied:

- To every generation, $f_{ed} = 1$.
- To none of the generations, $f_{ed} = 0$.
- With a certain frequency, $f_{ed} = n$, where n is greater than1. The editing operator is applied to every generation t where t modulo n is equal to zero.

While the editing operator produces parsimonious programs from non-parsimonious programs it may reduce the genetic diversity of the population.

10.6 Decimation

The decimation operator is used when the initial population has a large number of individuals with low fitness values. In such populations the individuals with higher fitness values could end up dominating each population and hence reducing the variety of the population. The following two parameters control the application of the decimation operator:

- A percentage, Koza denotes this by p_d.
- A condition satisfying when the decimation operator must be applied.

Example:

Suppose the percentage p_d is 10% and the condition specifies that the operator must be applied during generation 0. The fitness of each individual in the population in generation 0 is calculated. Based on their fitness, 10% of the population is deleted. If the decimation operator is applied to generation 0, then the run must start with ten times the population desired for the remainder of the run. The selection of individuals is based on their fitness. Re-selection is not allowed in order to maximize the diversity of the remaining population.

10.7 The Inversion Operator

The inversion operator swaps randomly selected subtrees within an individual. Subtrees to be swapped must not be contained within each other.

10.8 The Hoist Operator

The hoist operator basically controls the size of trees generated from generation to generation. The hoist operator randomly selects a subtree of an individual and copies it into the population of the next generation.

10.9 The Create Operator

The create operator generates a new parse tree, in the same manner as the trees of the initial population were generated, as part of the population of the next generation. In this way it maintains the genetic diversity of the system.

10.10 The Compress Operator

The compress operator performs a function similar to that of the encapsulation operator. An individual is randomly selected and part of the subtree up until a certain depth is defined as a module. The nodes at the level below the cut-off depth are considered as arguments to the module. The module is named and added to the function set.

10.11 The Expand Operator

The expand operator when applied to an individual will undo the application of a previous compress operator. The expand operator can only be applied to an individual that contains at least one node that represents a function created by the compress operator. The expand operator expands such a node into the function it represents.

Chapter 2 - Genetic Programming Applications

1. General Applications

Genetic programming has been applied to a number of different application domains.

Table 1.1 lists the number of genetic programming applications in each domain during the early to late nineties (taken from Banzaf, Nordin and Keller).

Application Domain	**Number of GP systems**
Algorithms (caching algorithms, sorting algorithms, graph theory algorithms, randomizers).	8
Art (jazz melodies, musical structure, artworks)	5
Biotechnology (biohemistry, DNA sequence identification, protein core detection)	9
Computer Graphics (3D modelling , computer animation, 3D object evaluation)	7
Computing (computer security, data compression, decision trees, machine language, parallelization, software fault predication, object orientation, virtual reality)	17
Control (board games, vehicle systems)	4
Process control (process engineering, modelling different chemical processes)	5
Control robotics (sensor evolution, motion and planning, obstacle avoiding)	27
Control space craft	2
Data mining (databases, internet agents, rule induction)	6
Electrical engineering circuit design	9
Financial market (horse race predication, share prediction, trade strategies, investment behaviour)	9
Hybrids (Fuzzy logic controllers, neural network training)	9
Image processing (classification, compression, edge detection, feature extraction, image enhancement)	14

Modelling (biotechnological fed-batch fermentation, spatial interaction models)	7
Natural languages (text classification, language decision trees, language processing)	4
Optimization (database query optimization maintenance scheduling, railroad track maintenance)	7
Pattern recognition (classification, feature extraction, object classification text classification)	20
Signal processing (control vehicle systems, waveform recognition)	5

Table 1.1

2. Presenting the Results Obtained Using a GP System

2.1 Standard Specifications

When solving a problem using a GP system you initially use a particular set of system parameters. These include:

- Population size (M)
- Number of Generations
- Tournament size
- Application rates for each of the genetic operators

Note that a single run is not sufficient to determine whether the system works or not and more than one run needs to be performed. Koza suggests that multiple independent runs instead of one long run should be made.If the system does not converge initially changes must be made to these parameters. In some cases changing the seed of the random number generator may also improve system performance.

Before making changes to a GP system that does not appear to be successful the system should be run a number of times, say 10-15 times. In systems where the resources are limited it is better to use fewer generations with larger populations than a larger number of generations with smaller population sizes.

A report describing the results obtained must specify

- A description of the objective, e.g finding “a mathematical expression for a given finite sample of a sequence where the target sequence is $5j^4 + 4j^3 + 3j^2 + 2j + 1$“.
- The terminal set used.

- The function set used.
- The fitness cases used.
- The raw fitness measure.
- Hits criterion.
- The population size.
- The number of generations.
- The success predicate used.
- The method used to create the initial population.

If the main aim of the system is to find a solution the seed of the random number generator used on the successful run must also be reported.

2.2 Describing the Performance of the System

Koza [6] describes the following histograms that can be used to present the results of genetic programming runs:

2.2.1 A Hits Histogram

A hits histogram is used to illustrate the frequency of the number of hits during a generation. An example of a hits histogram is illustrated in **Figure 2.1**.

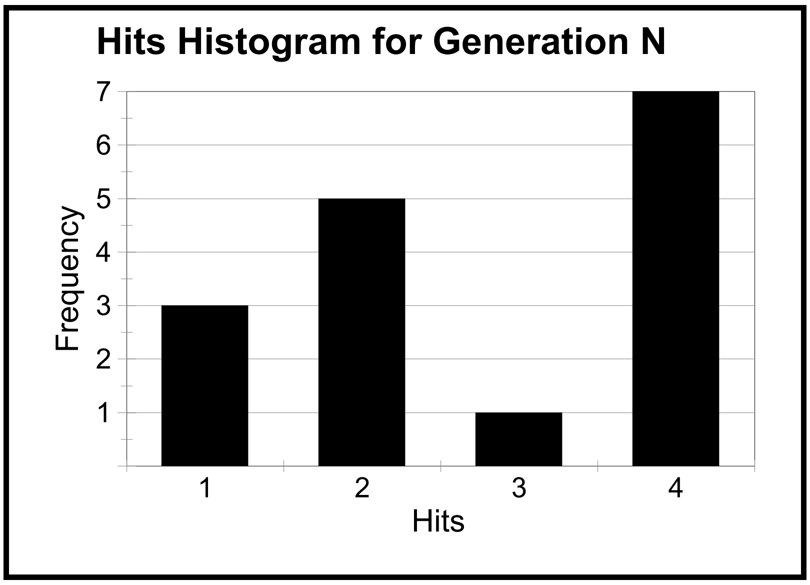


Figure 2.1

In this case four is the maximum number of generations.

2.2.2 Standardized Fitness Histogram

This histogram is used to graph the average standardized fitness obtained by the population during each generation of a run.

Figure 2.2 illustrates an example of a standardized fitness histogram. The maximum number of generations in this case is five.

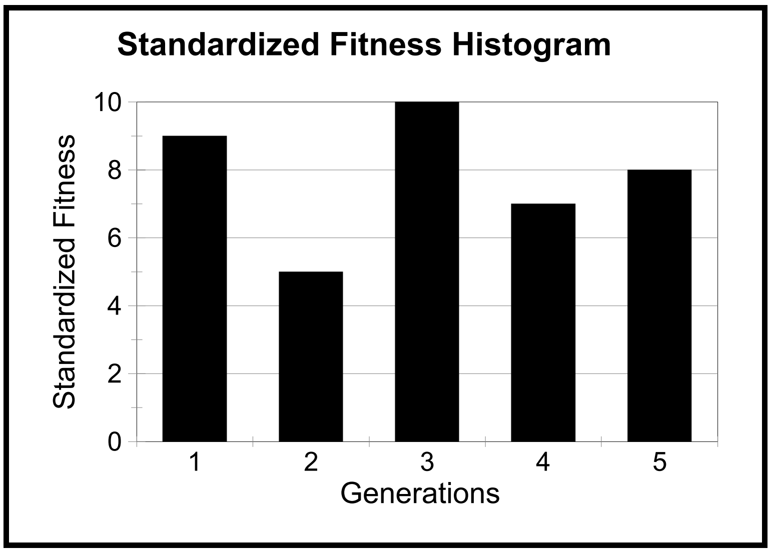


Figure 2.2

2.2.3 The Structural Complexity Histogram

The structural complexity of an individual is the number terminal and function nodes that the individual is composed of. The structural complexity histogram graphs the average structural complexity per generation of a genetic programming run. An example of a structural complexity histogram for a run of five generations is illustrated in **Figure 2.3**.

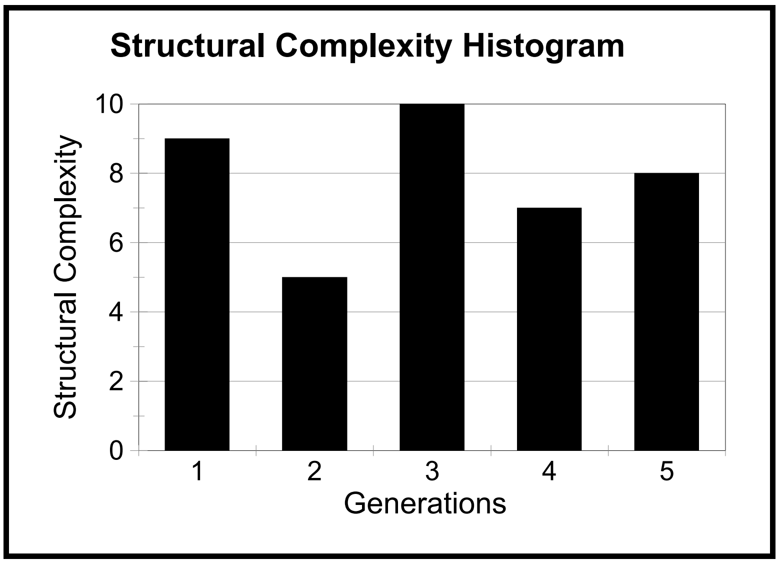
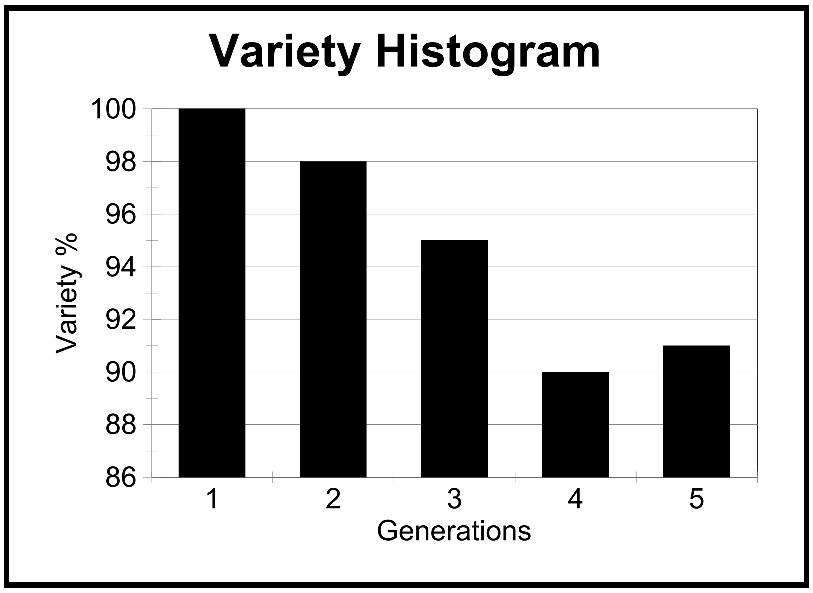


Figure 2.3

2.2.4 The Variety Histogram

The variety of a population describes the uniqueness of the population. The variety histogram graphs the variety for each generation of a genetic programming run. A variety histogram for five generations is illustrated in **Figure 2.4**.



2.2.5 Calculating the Computational Effort of a Genetic Programming Run

The probability, x, that a successful solution to a problem will be found in R independent runs of the GP algorithm is given by:

$$x = 1 - (1 - P(M, i))^R$$

where P(M,i) is the cumulative probability of success by generation i, using population M. P(M,i) is calculated by finding the total number of runs that succeeded before or on generation i and dividing this total by the total number of runs conducted.

If we fix the value of x to be 99% we can then solve the equation to obtain the number of runs R of the GP algorithm to i generations necessary such that there is a 99% chance of finding a solution in generation i. The following equation is used for this purpose:

$$R(x, M, i) = \left\lceil \frac{\log(1 - x)}{\log(1 - P(M, i))} \right\rceil$$

The number of individuals that must be examined as part of the search, i.e. the computational effort, is then calculated using: $f(x, M, i) = R(x, M, i)*M*i$

This equation is used in GP systems that use the generational control model. The following equation is used to calculate the number of individuals that will be examined in a steady-state system with a fixed population size: f(x, M, i) = R(x, M, i) * i

Example:

Given that four solutions were found by generation fourteen in sixty runs the total number of runs needed to find a solution with a 99% probability with a population size of a thousand is calculated as follows:

P(1000,14) = 4/60 = 0.07

$$R(0.99,1000,14) = \left\lceil \frac{\log(0.01)}{\log(1 - 4/60)} \right\rceil = 67$$

Thus, the computational effort is:

f(0.99, 1000,14) = 67*1000*14 = 938 000.

This means that 938000 programs have to be evaluated before a solution can be found.

3. Symbolic Regression Problems

Symbolic regression problems are essentially function-fitting problems which involve inducing a mathematical expression or function defined by a set of data points (expressed in terms of dependent and independent variables). These problems involve finding simple expressions such as x^3 and $\sqrt{x^3}$ (Kepler's law) and more complicated expressions such as $5j^4 + 4j^3 + 3j^2 + 2j + 1$,trigonometric identities, derivatives and integrals. This describes the application of genetic programming to the induction of :

- Kepler's law,
- polynomials,
- trigonometric identities.

3.1 The Derivation of Kepler's Third Law of Planetary Motion

Function to generate:	Induce the target function $\sqrt{A^3}$
Terminal set:	T = {A}
Function set:	F = {+, -, *, /, sqrt}
Number of generations:	10
Population size:	500

Raw fitness:	The sum, over the twenty fitness cases, of the absolute value of the difference between the target value produced by the derived expression and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the induced expression produces an output within 0.01 of the target value.
Bound:	0.01
Genetic operators:	Crossover - 80% Reproduction - 10% Mutation - 10%
Method of selection:	Tournament with a tournament size of two.
Initial population generator:	The grow method.

Fitness cases:

A	P
1	0.61
1	1
1.52	1.84
5.2	11.9
9.53	29.4
19.1	83.5

On every run a solution was found by the first generation with a hit value of six and a raw fitness value of 1.33E-4. The computational effort for this example is thus 500. Solutions derived by the system include:

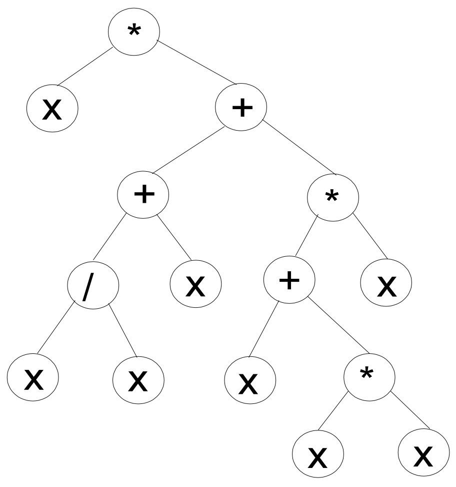
- sqrt(A*A*A)
- sqrt(A)*A
- sqrt(sqrt(A*A))*A

3.2 The Induction of a 4^{th} order Polynomial

Function to generate:	Induce the target function $x^4 + x^3 + x^2 + x$
Terminal set:	T = {x}
Function set:	F = {+, -, *, /}
Number of generations:	51
Population size:	500

Raw fitness:	The sum, over the twenty fitness cases, of the absolute value of the difference between the target value produced by the derived expression and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the induced expression produces an output within 0.01 of the target value.
Bound:	0.01
Genetic operators:	Crossover - 80% Reproduction - 10% Mutation - 10%
Method of selection:	Tournament with a tournament size of two.
Initial population generator:	The grow method with an initial tree depth of three and no limit on the size of trees created by the genetic operators.
Fitness cases:	The values of x were randomly generated in the range [-1, +1] and the corresponding target values are calculated. Twenty fitness cases are used.

Ten runs were performed. A solution was found by the ninth generation for all ten runs. The solution found on the first run was :

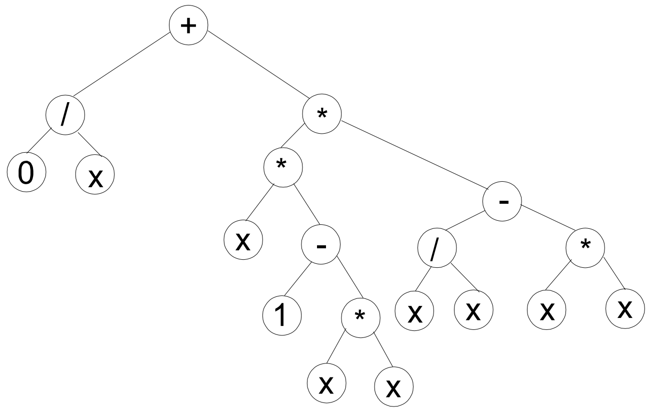


which simplifies to $x^4 + x^3 + x^2 + x$. The standard fitness of this individual is 8.60 x10E-16. The computational effort required to find a solution is 9*500 which is 4500.

3.3 The Derivation of a 5th Order Polynomial Including a Constant

Function to generate:	Induce the target function $x^5 - 2x^3 + x$.
Terminal set:	T = {x,$\Re$} $\Re$ is the $\Re$ephemeral random constant and ranges over the integers 0 and 1. When the ephemeral constant is encountered during the process of generating the initial population, or a subtree when applying the mutation operator, a random number in the integer interval [0,1] is randomly chosen and $\Re$ is replaced by this number in the tree.
Function set:	F = {+, -, *, /}
Number of generations:	51
Population size:	500
Raw fitness:	The sum, over the twenty fitness cases, of the absolute value of the difference between the target value produced by the derived expression and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the induced expression produces an output within 0.01 of the target value.
Bound:	0.01
Genetic operators:	Crossover - 60% Reproduction - 20% Mutation - 20%
Method of selection:	Tournament with a tournament size of four.
Initial population generator:	The grow method with an initial tree depth of three and no limit on the size of trees created by the genetic operators.
Fitness cases:	The values of x were randomly generated in the range [-1, +1] and the corresponding target values are calculated. Twenty fitness cases are used.

Ten runs were made. During the first run the following individual was found at generation ten with a raw fitness of 6.21E-16.



This simplifies to $x^5 - 2x^3 + x$. Of the ten runs made a solution was found on nine of the runs. During eight of the runs a solution was found by generation twenty. On one run a solution was found at generation forty seven. Thus, the maximum computational effort needed to obtain a solution is 2*500*47 is 47000.

3.4 The Derivation of Expressions Containing Real-Valued Constants

Function to generate:	Induce the target function $2.718x^2 + 3.1416x$.
Terminal set:	T = {x,$\Re$} $\Re$ is the ephemeral random constant and ranges over the real interval [-1,+1].
Function set:	F = {+, -, *, /}
Number of generations:	51
Population size:	500
Raw fitness:	The sum, over the twenty fitness cases, of the absolute value of the difference between the target value produced by the derived expression and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the induced expression produces an output within 0.01 of the target value.
Bound:	0.01
Genetic operators:	Crossover - 80% Reproduction - 10% Mutation - 10%
Method of selection:	Tournament with a tournament size of ten.
Initial population generator:	The grow method with an initial tree depth of three and no limit on the size of trees created by the genetic operators.
Fitness cases:	The values of x were randomly generated in the range [-1, +1] and the corresponding target values are calculated. Twenty fitness cases are used.

Ten runs were made. Solutions were found on eight of the ten runs. The inorder traversal of one of the solutions found with a raw fitness of 0.02 is listed below:

X* X+ X+ 0.1855 * - 0.4290 - x+ x + 0.5390 * X + X + X +X * 0.3469+ X * X +X +0.16847+X*X.

All solutions were found by generation thirty nine. Thus, of individual that need to be evaluated before a solution can be found is 3*500*39 which is 58500.

However, it was found that using just twenty fitness cases resulted in brittle solutions and in order to increase the generality of the system at a hundred fitness cases should be used (Koza uses a thousand). When deriving expressions that include real values it is essential to test the induced solution on a validation set of data in order to ensure it is general enough. If it is not increase the size of the training set.

3.5 The Derivation of a Trigonometric Identity

Function to generate:	Induce a trigonometric identity for the function $\cos 2x$.
Terminal set:	T = {x,1}
Function set:	F = {+, -, *, /,sin}
Number of generations:	51
Population size:	500
Raw fitness:	The sum, over the twenty fitness cases, of the absolute value of the difference between the target value produced by the derived expression and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the induced expression produces an output within 0.01 of the target value.
Bound:	0.01
Genetic operators:	Crossover - 80% Reproduction - 10% Mutation - 10%
Method of selection:	Tournament with a tournament size of ten.
Initial population generator:	The grow method with an initial tree depth of three and no limit on the size of trees created by the genetic operators.
Fitness cases:	The values of x were randomly generated in the range [0, 2 π] and the corresponding target values(are calculated $\cos 2x$). Twenty fitness cases are used.

Ten runs were made of which solutions were found on eight of the runs be generation 32. A preorder traversal of one the solutions with a raw fitness value of 0.039 is listed below:

sin – + / / 1 * 1 + sin sin 1 1 x / x 1. The computational effort for this example is 3*32*500 which is 48000.

3.6 Sequence Induction

Function to generate:	A mathematical expression for a given sequence of numbers where the target sequence is $5j^4 + 4j^3 + 3j^2 + 2j + 1$
Terminal set:	T = {j, $\Re$} $\Re$ is the ephemeral random constant and ranges over the integers 0 and 5
Function set:	F = {+, -, *}
Number of generations:	51
Population size:	800

Raw fitness:	The sum, over the twenty fitness cases, of the absolute value of the difference between the target value produced by the derived expression and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the induced expression produces an output that is equal to the target value.
Bound:	0.0
Genetic operators:	Crossover - 80% Reproduction - 10% Mutation - 10%
Method of selection:	Tournament with a tournament size of four.
Initial population generator:	The grow method with an initial tree depth of four and no limit on the size of trees created by the genetic operators.
Fitness cases:	The first twenty elements of the sequence.

Ten runs were made. A solution was found on four of the ten runs. Each solution was found by generation 22. A preorder traversal of one of the solution trees is: + * * * j j j * 5 + 1 - +++ * J J * * J J 3 5 * J 4 + + * * J J J + + * 2 J + 1 2 + * J J 0 1.

The computational effort for this example is 9*800*22, i.e. 158400 individuals need to be processed in order to guarantee a solution to the problem with a probability of 99%. Increasing the population size and number of generations could possibly result in a larger cumulative probability. The use of automatically defined functions, iteration and indexed memory can be incorporated into the system to find a more efficient solution.

3.7 Inducing a Boolean Even-Parity Function

An even-parity Boolean function returns a value of true if a given binary string has an even number of true values else it returns a value of false. The even_3_parity function is the even parity function applied to a binary string of length three. In this section we will examine how genetic programming can be used to automatically induce this function.

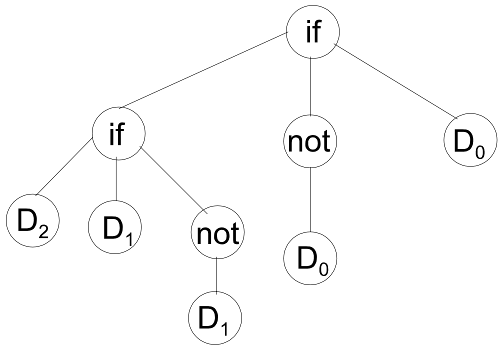
Function to generate:	The even_3_parity function.
Terminal set:	$\{D_0,D_1,D_2\}$
Function set:	F = {OR, AND, NOT, IF}
Number of generations:	51
Population size:	500
Raw fitness:	The number of fitness cases for which the induced expression produces an output that is equal to the target value.
Hits criterion:	Same as raw fitness
Genetic operators:	Crossover - 80% Reproduction - 10% Mutation - 10%

Method of selection:	Tournament selection with a tournament size of ten.
Initial population generator:	The grow method with an initial tree depth of three and no limit on the size of trees created by the genetic operators.

Fitness cases:

D0	D1	D2	Target Output
0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0
1	1	0	1
0	1	1	1
1	0	1	1
1	1	1	0

Ten runs were made. A solution was found by generation 7 on eight of the runs. One of the solutions found is :



The computation effort for this problem (requiring a 99% success rate) is 3*7*500, which is 10500.

3.8 Inducing a Boolean Symmetry Function

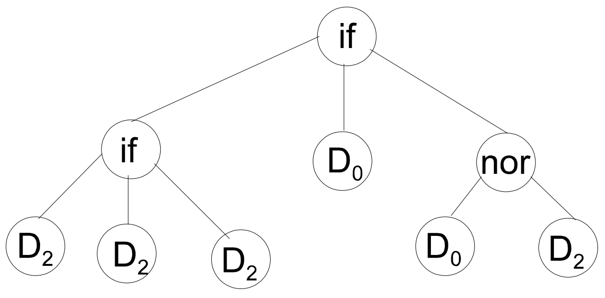
A Boolean symmetry function returns a value of true if the elements of a given binary string are arranged symmetrically, e.g. 101 and 1001 are both symmetrical binary strings while 011 and 0101 are not. The 3_symmetry function is applied to a string of length three. In this section we will examine how genetic programming can be used to automatically induce this function.

Function to generate:	The 3_symmetry function.
Terminal set:	T= {D_0,D_1,D_2}
Function set:	F = {OR, AND, NOR, NAND, IF}
Number of generations:	51
Population size:	500
Raw fitness:	The number of fitness cases for which the induced expression produces an output that is equal to the target value.
Hits criterion:	Same as raw fitness
Genetic operators:	Crossover - 80% Reproduction - 10% Mutation - 10%
Method of selection:	Tournament with a tournament size of ten.
Initial population generator:	The grow method with an initial tree depth of three and no limit on the size of trees created by the genetic operators.

Fitness cases:

D0	**D1**	**D2**	**Target Output**
0	0	0	1
1	0	0	0
0	1	0	1
0	0	1	0
1	1	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Ten runs were made. A solution was found by generation 1 on each of these runs. Thus the computational effort in this example is 500. One of the solution trees found is:



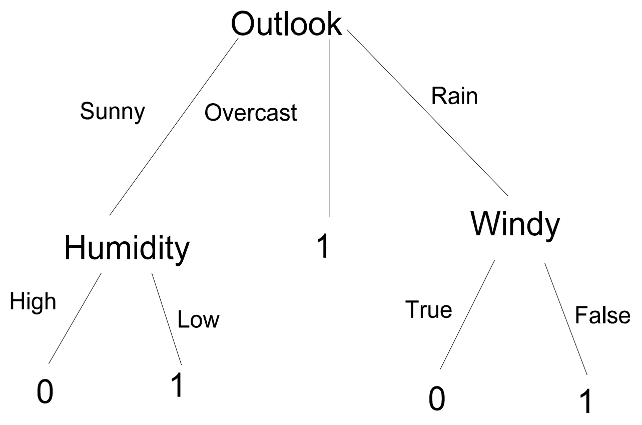
4. Decision Tree Induction

Rule-induction systems, like ID3, induce a set of rules from a given set of examples. Such systems are usually used for classification purposes. Given a number of examples representing objects with different attributes a rule-induction systems derives a set of rules to identify the different objects. In this section we will examine how genetic programming can be applied to this task. Suppose that we are given the following examples. the attributes of which are used to classify a morning as being a Saturday morning or not:

Temperature	Outlook	Humidity	Windy	Saturday Morning?
High	Overcast	High	True	Yes
High	Overcast	High	False	Yes
High	Overcast	Low	True	Yes
High	Overcast	Low	False	Yes
Medium	Overcast	High	True	Yes
Medium	Overcast	High	False	Yes
Medium	Overcast	Low	True	Yes
Medium	Overcast	Low	False	Yes
Low	Overcast	High	True	Yes
Low	Overcast	High	False	Yes
Low	Overcast	Low	True	Yes
Low	Overcast	Low	False	Yes
High	Sunny	High	True	No
High	Sunny	High	False	No
High	Sunny	Low	True	Yes
High	Sunny	Low	False	Yes
Medium	Sunny	High	True	No
Medium	Sunny	High	False	No
Medium	Sunny	Low	True	Yes
Medium	Sunny	Low	False	Yes

Low	Sunny	High	True	No
Low	Sunny	High	False	No
Low	Sunny	Low	True	Yes
Low	Sunny	Low	False	Yes
High	Rainy	High	True	No
High	Rainy	High	False	Yes
High	Rainy	Low	True	No
High	Rainy	Low	False	Yes
Medium	Rainy	High	True	No
Medium	Rainy	High	False	Yes
Medium	Rainy	Low	True	No
Medium	Rainy	Low	False	Yes
Low	Rainy	High	True	No
Low	Rainy	High	False	Yes
Low	Rainy	Low	True	No
Low	Rainy	Low	False	Yes

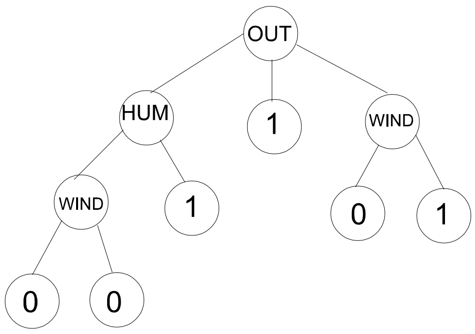
A solution found by the ID3 system using fourteen of these thirty six examples is:

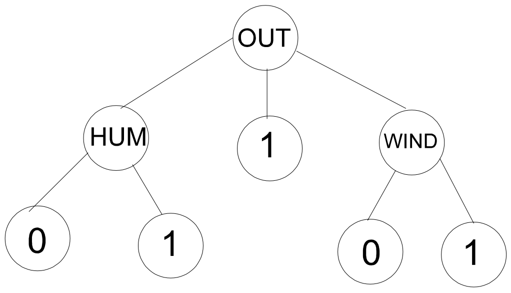


Class 0 represents all those individuals which do not satisfy the conditions for a Saturday morning while Class 1 represents all those individuals which satisfy the conditions for a Saturday morning.

Function to generate:	A decision tree to classify objects as Saturday mornings or not.
Terminal set:	{0, 1} , i.e. both the classes
Function set:	F = {TEMP, HUM, OUT, WIND}
Number of generations:	51
Population size:	500
Raw fitness:	The number of fitness cases correctly classified.
Hits criterion:	Same as raw fitness
Genetic operators:	Crossover - 60% Reproduction - 10% Mutation - 30%
Method of selection:	Tournament selection with a tournament size of ten.
Initial population generator:	The grow method with an initial tree depth of three and no limit on the size of trees created by the genetic operators.
Fitness cases:	The thirty six examples listed above.

Ten runs were made. A solution was found by generation 4 on all ten runs. Thus, the computational complexity is, with a success probability of 99%, is 500*4.
Two of the solution trees found are:





5. Developing Game Playing Strategies

This section applies genetic programming to the induction of game-playing strategies. Both sections address two-person, zero-sum games. The first section derives a minimax strategy for a particular player while the second looks at deriving a strategy for the pursuer in the pursuer-evader game.

5.1. Evolving a minimax strategy

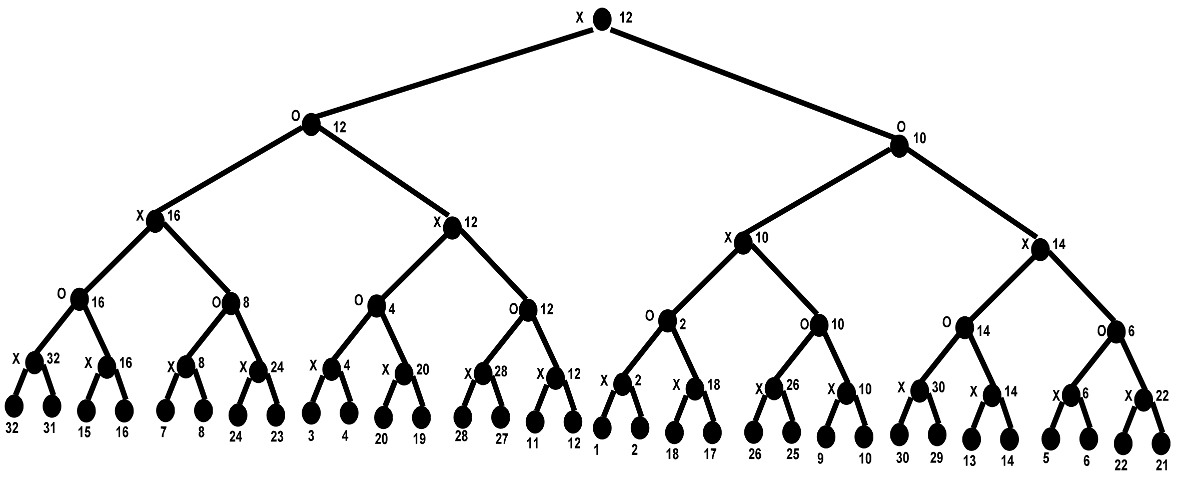


Figure 5.1.1: 32-outcome game tree (from [6])

A game-playing strategy essentially indicates to a user which is the best move to make next. When generating a game-playing strategy either the entire history up until that point is taken into consideration. Alternatively, just the current state of the game is examined. In this section we evolve the minimax strategy for the player X in the game tree in **Figure 5.1.1**. This requires the entire history to be considered. Section **2** provides an example of evolving a strategy by only examining the current state.

5.1.1 Defining Operators

Let XM1 and XM2 represent player X's first and second moves respectively. Similarly, let OM1 and OM2 represent players O's moves respectively. The terminal set consists of the possibly moves that can be made by both players, i.e. left and right. The function set is comprised of the operators CXM1, CXM2, COM1, and COM2, each of which has an arity of three. CXM1 evaluates its first child if players X has not made a first move, i.e. the move is undefined. It evaluates its second child if the first move made by X is left and executes its third child if the move made is right. CXM2 evaluates its first child if player X's first move is undefined, evaluates its second child if XM2 is left and its third child if XM2 is right. COM1 and COM2 perform the same function for the first and second moves of player O.

5.1.2 Fitness Cases

This fitness cases are comprised of the four combinations of possible moves that can be made by player O:

(Note: format (XM1, OM1, XM2, OM2))

LRRR, LLLL, LLLR, LRRL.

5.1.3 Evaluating an Individual

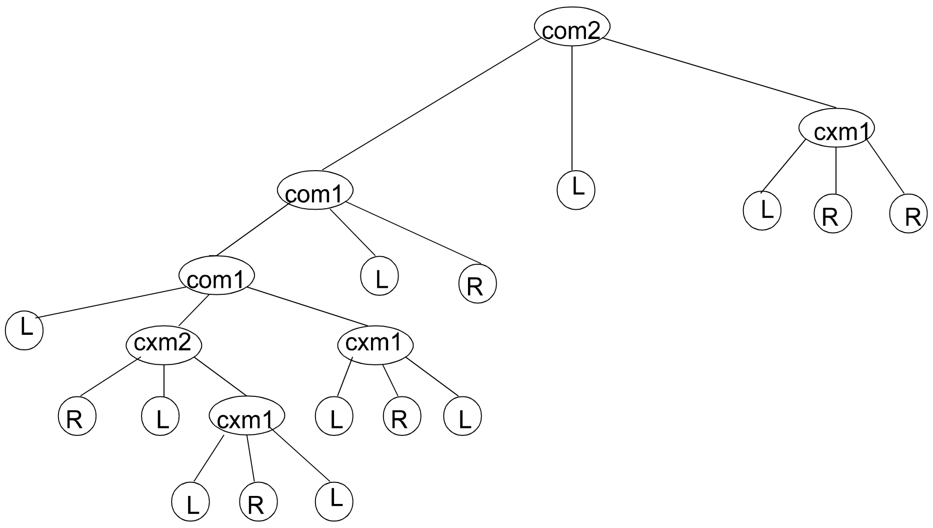
The raw fitness of an individual is the sum of the payoffs (indicated on the game-tree in Figure 1.1) obtained by the individual for each of the four fitness cases. The hits ratio is the number of fitness cases for which the individual receives a payoff at least as good as the minimax strategy.

5.1.4 Other GP Parameters

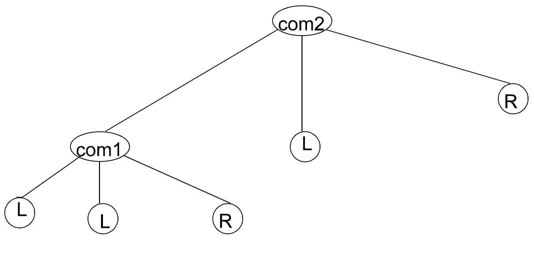
Population size:	500
Max. no. of generations:	51
Initial population *generator:*	The ramped half-and-half method with an initial tree depth of six and a depth limit of seventeen on the size of trees created by the genetic operators.
Method of selection:	Fitness proportionate selection

5.1.5 An Evolved Solution

One of the solutions evolved is:



This simplifies to:



5.2 Evolving a strategy for the purser-evader game

As the name suggests, the two players in this game are the pursuer and the evader. The pursuer is essentially chasing the evader and wins the game if the evader is caught within a specified minimum time. It is given that the pursuer moves faster than the evader. The payoff for the pursuer is the time it takes to catch the evader and the payoff of the evader is the time it remains free. Thus, the pursuer wants minimize its score while the evader wants to maximize its score. This section examines evolving a game-playing strategy for the pursuer. Note that we need to find a "best" strategy not an exact solution.

The information available at each stage of the game is the position of the pursuer and the evader. A game-playing strategy will specify the angle at which the pursuer must move in order to catch the evader. Thus, the GP system must evolve an equation (in radians) representing the angle at which the pursuer must move given the position of the evader. Note that the purser is always assumed to be at position (0,0), i.e. it is assumed that whenever the pursuer moves the axes move with it. **Figure 5.2.1** provides an example of this.

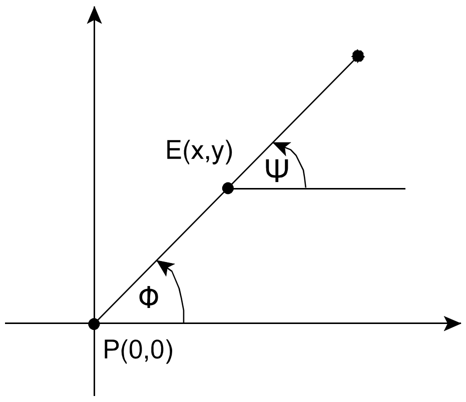


Figure 5.2.1

5.2.1 Function and terminal sets

The terminal set consists of the coordinates of the position of the evader (X, Y). In addition to this the ephemeral constant, say in the range -1 to +1, should be included to allow for the introduction of constants. Thus, T = { X, Y, $\Re$ }.

The function set consists of the standard arithmetic functions, the exponential function and the conditional operator IFLTZ . If the first argument of IFLTZ is less than zero it returns its second argument else it returns its third argument.

5.2.2 Fitness cases and evaluation

The fitness cases consists of 20 different positions of the evader on the plane, i.e. a set of (X, Y) coordinate values.

The raw fitness of an individual is average time required to catch the evader over the 20 fitness cases. An upper limit is set on the maximum time permitted. The hits ratio is the number of fitness cases for which this time limit is not exceeded. Any individual with a hits ratio of twenty is considered to be a "best" strategy.

6. Evolving Algorithms Containing Iterative Control Structures

This section examines the use of iterative operators such as the *for* loop or the *dowhile* loop. The use of iterative control structures can result in infinite and time-consuming algorithms. Methodologies for dealing with this problem is described in the following section. Section examines the use of an iterative operator in algorithms evolved for the **blocks world problem**. Section **6.3** looks at evolving solution algorithms to problems usually used to teach novice programmers iteration.

6.1 Dealing with the Halting Problem

It has been shown that it cannot be determined whether a given program terminates or not. This is referred to as the halting problem. This problem can be dealt with in a GP system by performing time-bounded executions of programs containing loops and recursion.

One of the following methods is commonly used to prevent infinite and time-consuming iterations during executions of GP individuals:

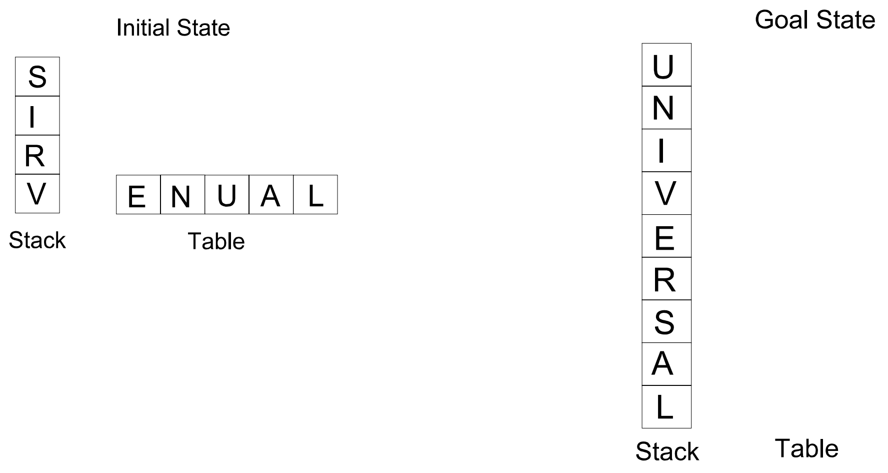
- If there are N different parse trees representing N different algorithms, all the algorithms are run at once on a particular fitness case. The algorithms are stopped when a certain time threshold is exceeded. For each individual the system maintains an average of the fitness scores the individual has obtained on the previous fitness cases, prior to the fitness case it did not terminate on. The fitness of the individual is calculated using this average. This is called the "popcorn" method.
- After the algorithms have run for a certain amount of time they are stopped and are required to return the response that they have calculated thus far. Thus, the algorithms are not required to terminate. This is referred to as an "anytime" algorithm. This algorithm is implemented as follows. For each individual the corresponding input is placed in memory. The individual is given a specified (short) period in which to run. Usually, the entire algorithm can not be executed during this time. At the end of the time period the contents of the relevant memory positions are extracted and interpreted as the answer.

- Setting a bound on the number of iterations per control structure or/and on the number of iterations per individual.

6.2 The Blocks World Problem

Description of the Blocks World Problem:

A number of blocks need to be arranged into a stack which spells out the word UNIVERSAL. Some of the blocks already exist in the stack while the rest are on a table. The initial and goal states for this problem are illustrated below:



6.2.1 Terminal Set

Koza [6] proposes that the terminal set contains the following "sensors":

- CS indicates which block is on the top of the stack.
- TB indicates which block on the stack is at the top a number of correctly ordered blocks.
- NN indicates the next block from the table that must be placed on TB on the stack. It does not matter if there are other incorrectly ordered blocks on top of TB.

The values that the terminals in the terminal set can take on are the labels of each block and the value false or null (e.g.in those fitness cases where the stack or the table is empty).

6.2.2 Function Set

Koza defines the function set to contain the following elements:

- MS - Takes the label of a block, X, as its only argument. This function firstly checks whether X is on the table or not. If X is on the table, it is moved to the top of the stack. If this is successfully performed the a value of true is returned otherwise a value of false is returned. If X is null (or false), or X is already on the stack, or the table is empty MS returns a value of false.
- MT - Takes the label of a block, X, as its only argument. This function firstly verifies that X is on the stack. If X is on the stack this function moves the first element of the stack to the table and returns a value of true. If X is already on the table or X is null or the table is empty MT returns a value of false.
- DU - Performs a do...until loop. This function takes two arguments. The first represents the code that must be executed for a number of iterations and the second is a condition specifying when the iteration must stop. One of the problems associated with incorporating iteration into a GP system is the possibility of infinite iteration or individuals running for too long. For this reason a limit of 25 is set on the maximum number of iterations a do...until loop can perform and a limit of 100 on the maximum number of iterations an individual can perform.

  This function returns a value of true if the entire loop is performed and is not terminated due to either of the limits being exceeded otherwise it returns a value of false.
- NOT - Takes a single argument and performs the function of the logical not.
- EQ - Takes two arguments and performs the function of the logical equal.

6.2.3 Fitness Cases

The fitness cases for this example are different combinations of blocks on both the stack and table. Koza [6] uses the following 166 configurations:

- Ten cases consisting of 0 to nine blocks already in the correct order.
- Eight fitness cases where there is exactly one block in the wrong order on top of the remaining correctly ordered blocks on the stack.
- A hundred and forty eight cases where 0 - 8 blocks are correctly ordered in the stack and a random number in the interval 2-8 blocks out-of-order on the stack.

6.2.4 Raw Fitness and Hits Criterion

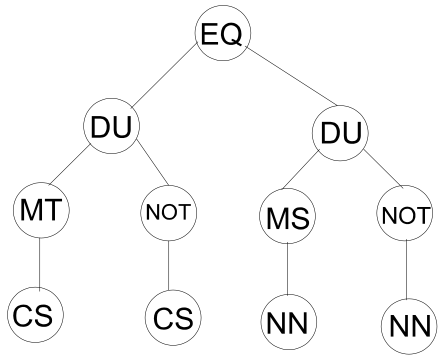
The raw fitness of an individual is the number of fitness cases for which the individual produces the goal state from the given initial state. The hits criterion is the same as the raw fitness. The algorithm terminates when an individual with 166 hits has been found.

6.2.5 Summary of GP Parameters

Function to generate:	Induce program that implements a plan to add blocks to a stack so that it spells UNIVERSAL.
Terminal set:	T = {CS, TB, CS } ,
Function set:	F = {MS, MT, DU, NOT, EQ}
Number of generations:	51
Population size:	500
Raw fitness:	The number of fitness cases for which the stack is correctly constructed.
Hits criterion:	Same as raw fitness
Genetic operators:	Crossover - 90% Reproduction - 10% Mutation - 0%
Method of selection:	Fitness proportionate selection.
Initial population generator:	The ramped half-and-half method with an initial tree depth of six and a depth limit of seventeen on the size of trees created by the genetic operators.
Fitness cases:	The 166 cases described above.

6.2.6 Solutions Evolved

Koza [6] performed one run. The following solution was found at generation 10:



6.2.7 Evolving an Efficient and Parsimonious Solution Algorithms

In addition to deriving a correct solution the fitness function can be extended to evolve efficient and/or parsimonious solution algorithms. An algorithm is more efficient than another algorithm if it achieves the same goal in a fewer number of steps. Similarly, an algorithm is more parsimonious than another if it contains a fewer number of nodes. One option would be to a weighted sum of each aspect, e.g. 75% for correctness, 15% for efficiency, and 10% for parsimony. Alternatively, a Pareto fitness, which can compares each dimension of fitness, can be used.

6.3 Evolving Solution Algorithms to Novice Iterative Programming Problems

This section looks at inducing solution algorithms to introductory level procedural programming algorithms involving the use of iteration. The use of the standard *for* operator and conditional iterative operators are examined.

A genetic programming system incorporating the use of these operators must be strongly-typed, i.e. each terminal node, function node, and function node argument must be of a specific type, e.g. Boolean, Integer. In such a system each child of the right type has to be chosen during initial population. Similarly, a subtree removed during mutation has to be replaced with a subtree of the correct type and during the crossover process only subtrees of the same type can be swapped.

In order to prevent infinite and time-consuming iterations an upper bound is set on the number of iterations that can be performed by an individual over all fitness cases.

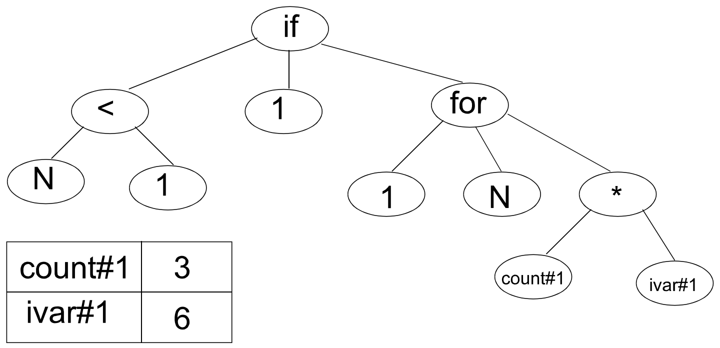
6.3.1 The Standard *for* Operator

The *for* operator defined in [11] takes three arguments. The first two arguments are of type Integer and represent a "starting" and "ending" value for the loop. The third argument can be of any type and represents the subtree to be executed on each iteration. The type of this argument is instantiated during initial population generation. Each instance of the *for* operator is defined to be of the same type as its third argument.

Each individual in the population is represented by a parse tree and a memory structure. For each *for* instance contained in an individual, two variables are maintained in the individual's memory structure. The first is called the **counter** variable. This variable is assigned the value of the first argument the *for* operator instance evaluates to and is incremented (when the *for* operators first argument is less than or equal to its second argument) or decremented (when the *for* operators first argument is greater than its second argument) on each iteration.

The number of iterations performed is the difference between the values the first two arguments of the *for* instance operator evaluates to plus one. The second variable is referred to as the **iteration** variable and stores the value of the current iteration, i.e. the value that the third argument of the *for* operator instance evaluates it. The initial value given to this variable is context-sensitive, e.g. if the operation it is involved in is addition it is given an initial value of zero, alternatively if the operation is multiplication it is given a value of one. Both these variables are added to the terminal set when evolving the subtree representing the third argument of a *for* operator instance.

The following figure illustrates a solution algorithm for the factorial example after it has been executed using an input value of 3 and target value of 6.



count#1	3
ivar#1	6

Evolved Iterative Algorithm

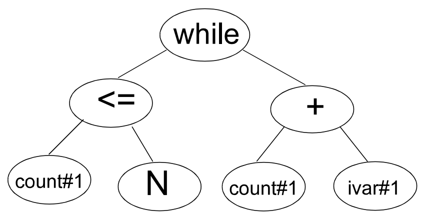
6.3.2 Conditional Loops

The references [11] describes two types of conditional loop operators, namely, the *while* operator and the *dowhile* operator. These operators takes two arguments, the first of which is Boolean. The second argument can be of any type and is instantiated during the process of initial population. Each *dowhile/while* operator instance type is the same as its second argument.

A counter and iteration variable are also maintained for each *while* and *dowhile* operator instance. These variables are added to the terminal set when generating both arguments of each operator instance. The counter variable is given an initial value of one and is incremented on each iteration. **Figure 6.3.2.1** illustrates an example of an individual containing a *while* operator instance and **Figure 6.3.2.2** depicts an individual which contains a *dowhile* operator instance. In both cases the diagrams show the effect of evaluating the individual with an input value of 2 and an output value of 3.

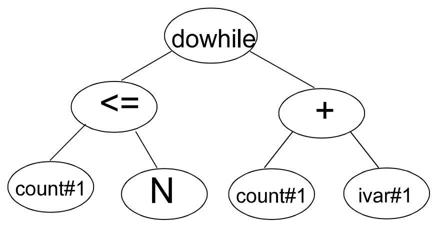
What changes need to be made to the genetic operators to cater for these operators?

Can these operators be improved in any way?



count#1	2
ivar#1	3

Figure 6.3.2.1



count#1	2
ivar#1	3

Figure 6.3.2.2

7. Evolving Recursive Algorithms

This section looks at evolving recursive functions using genetic programming. The first attempt to evolve recursive functions was made by Koza[6] and resulted in the definition of the SRF function. This is discussed in section **1**. The use of automatically defined recursion (ADR) described in [8] is presented in section **3**. Finally, the recursion operator defined in[11] to evolve novice procedural recursive functions is discussed.

7.1 The SRF Function

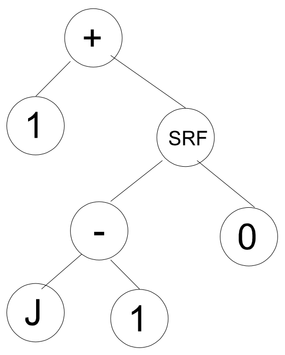
The SRF function takes two arguments (both of which are integer values). The system stores the value that the SRF outputs for each index of the sequence. Suppose that we have the following fitness cases:

Index (J)	**Target Value**
0	1
1	1
2	2
3	3
4	5

An instance of the SRF function, e.g. *SRF K D* is interpreted as follows:

- If the value of *K* is equal to the index of a fitness case that the SRF has already been evaluated on the function returns the stored value. If not the value of *D* is returned.
- The value returned is stored as the output for the index specified in the fitness case the SRF instance is currently being evaluated on.

What will the output of the following individual be ?



7.1.1 GP Parameters and Solutions

Function to generate:	Induce a recursive expression for the Fibonacci sequence.
Terminal set:	T = {J, $\Re$} where $\Re$ is the ephemeral constant in the range 0 to 3.
Function set:	F = {+,-,*,SRF}
Number of generations:	51
Population size:	2000
Raw fitness:	The sum of the absolute value of the difference between the value produced by the induced tree and the target value for each of the twenty fitness cases.
Hits criterion:	The number of fitness cases for which the tree has produced a value equal to the target value.
Bound:	0
Genetic operators:	Crossover - 90% Reproduction - 10% Mutation - 0%
Method of selection:	Fitness proportionate selection.
Initial population generator:	The ramped half-and-half method with an initial tree depth of six and a depth limit of seventeen on the size of trees created by the genetic operators.
Fitness cases:	The first twenty elements of the Fibonacci sequence.

Koza performed two runs. A solution was found by generation 30 in one of the runs and by generation 22 in the other. One of the solutions induced in prefix notation :

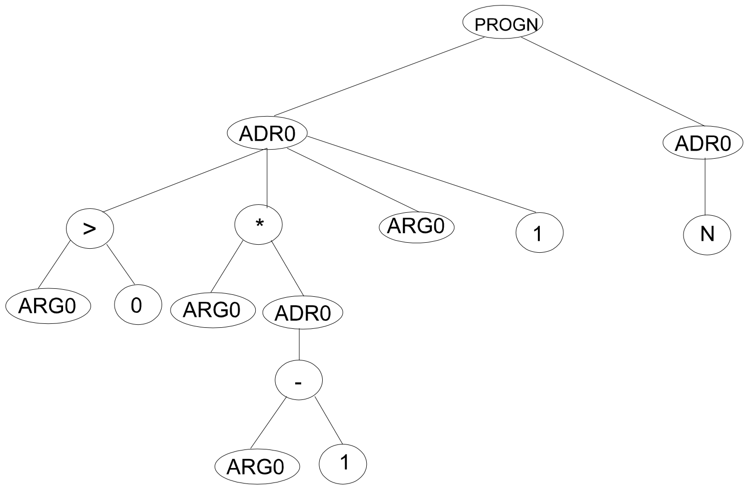
```
( + ( SRF ( - J 2 ) 0 )
    ( SRF ( + ( + ( - J 2 ) 0 ) (SRF ( - J J ) 0 ) )
          ( SRF ( SRF 3 1 ) 1 ) ) ) )
```

7.2 Automatically Defined Recursion (ADR)

Automatically defined recursions (ADR s) are described in [8]. An automatically defined recursion consists of four branches:

- A recursion condition branch (RCB) - Is executed first. The recursion continues provided this branch produces a value of a particular type, e.g. a positive integer.
- A recursion body branch (RBB) - This branch represents the subtree that is executed on each “recursion”. The ADR can be called up in this branch, i.e. it can call itself recursively.
- A recursion update branch (RUB) - Is executed after the RBB branch.
- A recursion ground branch (RGB) - Is executed once if the RCB does not return a value of the required type.

Example solution algorithm for calculating the factorial of a positive integer N:



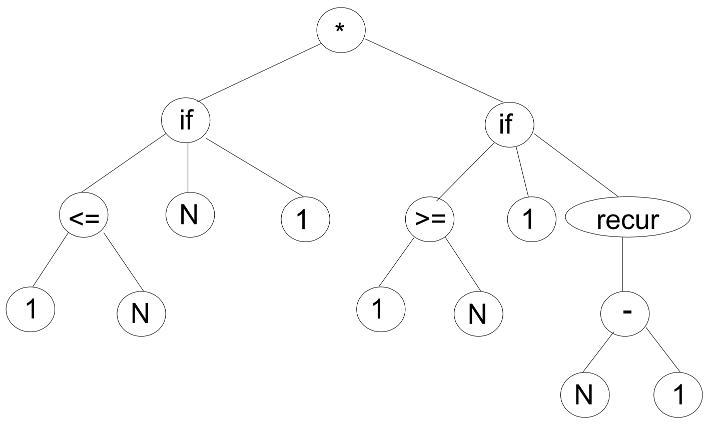
The right branch is referred to as the results-producing branch and represents the main program, while the left branch represents the recursive function in the form of an ADR. Each individual can have more than one ADR. A number of **architecture-altering** operators, e.g. recursion duplication, recursion deletion, have been defined [8] for evolution purposes. These operations will be discussed at a later stage.

7.3 The *recur* Operator

The *recur* operator is defined in [11]. This operator can be inserted at any position in an individual except the root of the tree and enables the parse tree to call itself. The arity of the operator is equal to the number of input variables (not including constants) for the problem at hand. Each argument of the *recur* operator represents one input variable of the problem. Thus, the subtrees representing each argument form the updates for each variable. In a strongly-typed GP system, the type of this operator is the same as the root of the tree.

Interpreting an instance of the *recur* operator results in the parse tree being re-evaluated with the updated values for each input variable. In order to prevent time-consuming and infinite recursions an upper bound is placed on the number of recursions that can be performed per individual. The *recur* operator returns a type-dependant value if its is not executed due to exceeding this limit.

An example solution to the factorial problem:


*
if
if
<=
N
1
>=
1
recur
1
N
1
N
-
N
1

Chapter 3 - Advanced Genetic Programming Features

This chapter examines three advanced genetic programming topics, namely memory, modularization, and architecture-altering operations.

1. Evolving Algorithms that Use Memory

Algorithms require the use of memory and usually use variables to access different memory locations. This section looks at how GP systems can induce algorithms that use memory. The different types of memory are described in section **1.1**. Section **1.2** examines the use of automatically defined storage. An application of the use of named memory is presented in section **1.3**. Section **1.4** briefly discusses an extension of the use of memory, namely, the use of and evolution of data structure algorithms.

1.1 Different Types of Memory

The first memory structure introduced was **indexed memory** defined by Teller [14]. Indexed memory is represented as an array. Each array consists of integers indexed over integers. Two functions used to access memory are added to function set: ***read*** and ***write***. The ***read*** function takes one argument, namely the index of the array. It returns the element stored at the index of the array. The ***write*** function takes two arguments, an index and an integer value to be stored in the array at the index. The *write* function returns the current value stored at the index and stores the integer value passed to it at the index position in the array. Indexed memory is used as a means of enabling the GP system to save past inputs and utilize them in processes at a later stage.

In [15] Teller describes how indexed memory can be incorporated into the GP system. Each individual in the population consists of a tree as well as an array of elements indexed from 0 to M -1 where M is the number of memory elements. The elements of the array are integers in the range 0 to M - 1. The ***read*** and ***write*** functions described above must be added to the function set. The terminals are constants between 0 and M -1 and the variables representing the system inputs. All the functions in the function set are defined to return integers in the range 0 to M-1. This ensures that all values computed are legal memory indexes. Thus, a wrapper is needed in this implementation. The choice of the number of memory elements may effect system performance.

Named memory is usually a single memory location. A *write* and *read* operator must be defined to facilitate access to this variable. Section **3** provides an example of how the use of named memory can be incorporated into evolved algorithms. Other examples of the use of named memory can be found in [8].

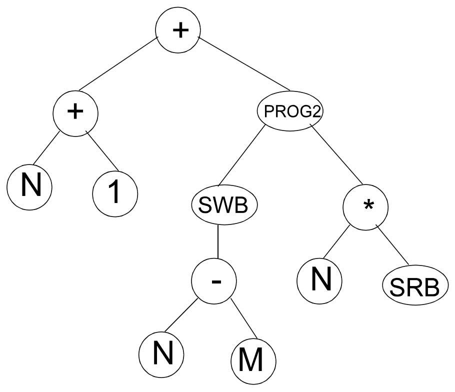
1.2 Automatically Defined Storage (ADS)

In [8] Koza introduces the concept of an automatically defined store. An ADS is comprised of two branches, namely, a storage writing branch (SWB) and a storage reading branch (SRB). The SWB branch performs the function of the *write* operator defined above while the SRB branch performs the function of a *read* operator. Each ADS has a name, dimensionality, type and size. The following types have been defined for each dimension in [8] :

Dimension	Types
0	Named memory, pushdown stack, queue
1	Indexed memory, list
2	Two-dimensional memory, relational memory
3	Three-dimensional array
4	Four-dimensional array

The dimension, type and size of each ADS instance is instantiated during initial population generation. An individual can contain one or more ADSs each of a different dimension, type and size. If the ADS type is named memory the system maintains a memory location for that instance. In this case SWB takes one argument. The value that the subtree representing this argument evaluates to, is written to the memory location. SRB does not take any arguments and returns the value written to this memory location. Similarly, if the ADS type is a pushdown stack the SWB takes one argument, the value of which is pushed onto the stack maintained by the system for that instance. When SRB is interpreted for the instance the topmost value of the stack is popped off and returned.

If the ADS type is indexed memory SWB has an arity of two and SRB has a value of one. SWB writes the result of its first argument to the index specified by its second argument while SRB returns the value stored at the index that its argument evaluates to. The following is an example of the use of an ADS. The type of the ADS instance in this individual is named memory.



Koza [8] defines architecture-altering functions for ADSs, e.g storage creation and storage deletion. These will be discussed in detail at a later stage.

1.3 Evolving the *swap* function

This section looks specifically at the use of named memory. The *write* and *change* operators used to access a single memory location defined in [11] are described in section **1.3.1**. Section **1.3.2** specifies the GP parameters used to evolve an algorithm for the swap function.

1.3.1 The *write* and *change* operators

The *write* operator takes a single argument. Whenever an instance of the *write* operator is added to memory during initial population generation, a memory location is allocated to that instance and a variable corresponding to the location is added to the terminal set used to create the rest of the individual. In a strongly-typed genetic programming system each memory location is of a particular type. Interpreting the *write* operator results in the value that its argument evaluates to being written to the allocated memory location for that instance. The value of its argument is also returned. Interpreting a node with a variable label results in the value stored at the particular memory location being returned.

The function of the *change* operator is to change the value stored in a particular memory location. The *change* operator takes two arguments. The first is a subtree representing the new value and the second is a variable name. In a strongly-typed GP system both arguments are of the same type and the change operator is only added to memory if a memory location of the required type exists. In a standard genetic programming system the *change* operator is only inserted into a tree if a memory location already exists, i.e. an instance of a *write* operator already exists. Execution of a *change* operator instance results in the value that its first argument evaluates to being written to the memory location specified by its second argument.

1.3.2 GP Parameters and Evolved Solution

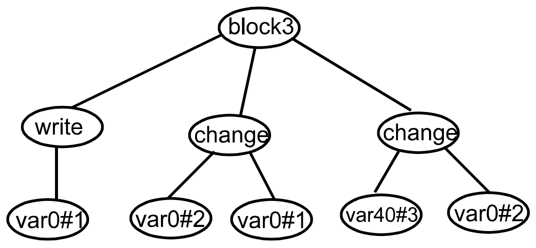
Function to generate:	Induce an algorithm for the *swap* function
Terminal set:	T = {var#01, var#02}
Function set:	F = {write, change, block3}
Number of generations:	51
Population size:	500
Fitness cases:	

Input : n1	Input: n2	Output: n1	Output: n2
100	500	500	100
315	415	415	315
475	556	556	475
230	320	320	230
345	647.89	647.89	345
574	878	878	574
45.7	109	109	45.7
123.45	187.64	187.64	123.45
345.2	765	765	345.2
56.79	89.73	89.73	56.79

Raw fitness:	The number of fitness cases for which both the output values are correct.
Hits criterion:	Same as the raw fitness
Bound:	0
Genetic operators:	Crossover - 50% Reproduction - 0% Mutation - 50% (mutation depth of 2)
Method of selection:	Tournament selection with a tournament size of 4.
Initial population generator:	The ramped half-and-half method with an initial tree depth of 3 and a node size limit of ten on the trees created by the genetic operators.

The generational control model was used. A strongly-typed genetic programming system was implemented. Each individual is comprised of a parse tree and a memory structure. A location is allocated for each input variable in the memory structure for each individual. During interpretation of each individual, the input values for the particular fitness case are written to the corresponding memory locations. After an individual is executed the values contained in its memory structure are compared to the target values to calculate its raw fitness. A solution was found on all ten runs performed.

One of the solutions evolved is illustrated below:



1.4 Data Structures

Langdon [9] extends the idea of memory use to the evolution of algorithms for linear data structures. This study involves inducing algorithms to access an array-based stack, queue, and linked-list. The data structures and evolved algorithms were then used to induce solutions algorithms to certain problems. According to Langdon this system performed better on the set of problems than the standard GP system with indexed memory.

The section examines the induction of the algorithms, namely *makenull*, *empty*, *top*, *pop*, and *push*, for an array-based, integer push-down stack. The following section defines a push-down stack. Section **1.4.2** describes the genetic programming system implemented. The solutions evolved are presented in section **1.4.3**.

1.4.1 The Push-Down stack

The push-down stack is essentially an array-based stack with the first element pushed onto the stack stored in the array at the position equal to the maximum size of the stack. As successive elements are added onto the stack the stack pointer is decremented by one. The following algorithms are used to access the stack:

- *makenull* - The stack pointer is assigned the value of the maximum size of the stack plus one so as to indicate that the stack is empty.
- *empty* - If the value of the stack pointer is not valid, i.e. not in the range of 1 to the maximum size of the stack, this method returns true otherwise it returns false.
- *top* - A copy of the element at the top of the stack is returned.
- *pop* - This method returns the element currently pointed to by the stack pointer, and increments the stack pointer by one.
- *push* - This method reduces the stack pointer by one and stores the element passed to it at this position.

14.2 The Genetic Programming System

Objective:	To simultaneously evolve the five algorithms for an integer, array-based push-down stack. Checks for stack overflow and underflow are not catered for.

Architecture: Each individual in the population is represented using a multi-tree genome. The genome consists of five parse trees representing each of the five stack operations.

Primitives:

- Constants: 0, 1
- *max* : The maximum size of the stack. This is given a value of 10.
- *arg1*: The value to be pushed onto the stack.
- Arithmetic operators: +, -
- *read* and *write* : These operators are used to access a global indexed memory structure. The indexed memory structure used consists of 63 memory cells, indexed from -31 to 31. The *read* operator takes one argument, namely the index from which to read an element. The *write* operator takes two arguments, an index and the value to be written to the particular index. If the index passed to either of these operators is invalid, evaluation of the program is terminated. The memory locations are initialised to zero prior to the evaluation of each fitness case.
- *aux*, *inc_aux*, *dec_aux* and *write_aux* : These primitives are used to access a global named memory location. *aux* returns the current value stored in the memory location. The operators *inc_aux* and *dec_aux* are used to increment and decrement the value stored at *aux* respectively. *write_aux* writes the value that it argument evaluates to the memory location accessed by *aux*. The memory location is initialised to zero prior to the evaluation of each fitness case.

Fitness Cases: Four test sequences were used. Each test sequence consists of 40 calls to each of the five stack operations and requires a value to be returned at the end of each function call. The values pushed onto the stack were randomly selected in the range -1000...999.

Raw Fitness: The number of function calls for which the same value is returned for each function call. The operations *makenull* and *push* are indirectly tested as they do not return a value. The score is automatically incremented by one for both these operations. Trees accessing an invalid memory location are penalised by stopping the evaluation and returning the fitness value calculated up until that point.

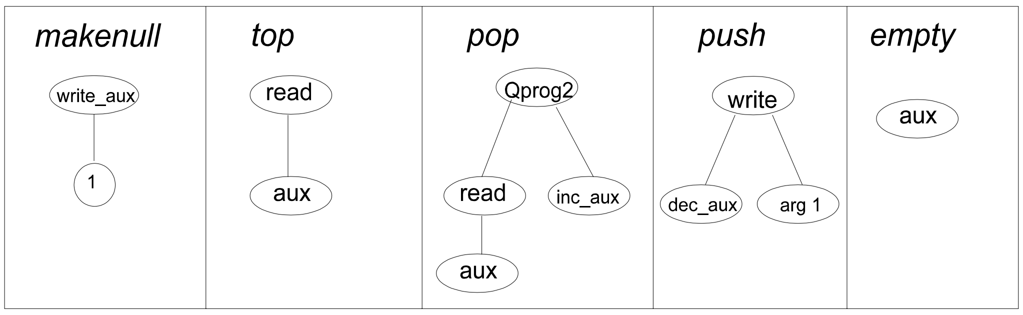
Selection: Tournament selection with a tournament size of 4.

Population Size:	1000
Generations:	101
Program Size:	<= 250
Success Predicate:	Fitness >= 160

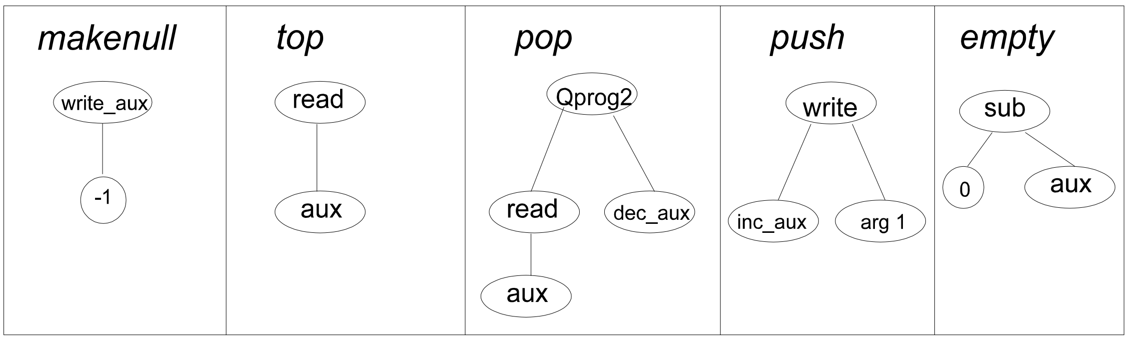
1.4.3 Evolved Solutions

This section presents two different types of solutions produced by the genetic programming system.

Solution 1



Solution 2



Notice that while the first solution evolved are algorithms for a push-down stack, the second solution evolved are methods for a push-up stack.

2. Evolving Modular Algorithms

Essentially two methodologies for evolving modular programs have been implemented thus far, namely, module acquisition and automatically defined functions (ADFs). The first section examines module acquisition while the second looks at automatically defined functions.

2.1 Module Acquisition

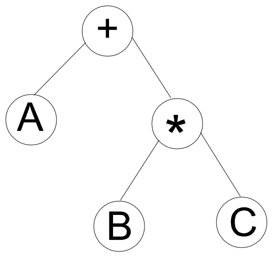
We have already studied module acquisition in the section on genetic operators. Module acquisition refers to the use of the compress and encapsulation operators. The functions of these operators are summarised below:

2.1.1 Encapsulation

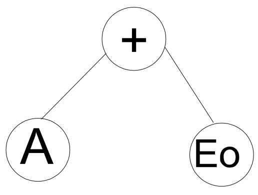
An individual is selected using one of the methods of selection.

- Select a function node at random.
- The subtree located at the selected point is removed.
- A new function is defined to represent the deleted tree. This function has no arguments.
- The new encapsulated function is labeled, E0, E1, E2,..,etc. The new function label is added to the terminal set. It represents a function without arguments.

Example: Consider the following parse tree



Suppose that the node containing the * function was randomly chosen. The function E0 is then created. The offspring produced is :



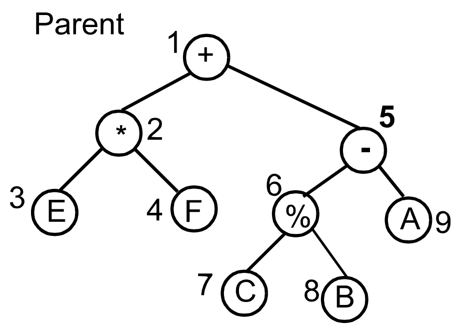
The subtree is no longer in danger of being subjected to the disruptive effects of crossover. The original parent is not changed by the application of the encapsulation operator. An intelligent encapsulation operator will choose a function node based on whether the subtree rooted at this node will contribute to the solution or not instead of randomly choosing a node.

2.1.2 Compression

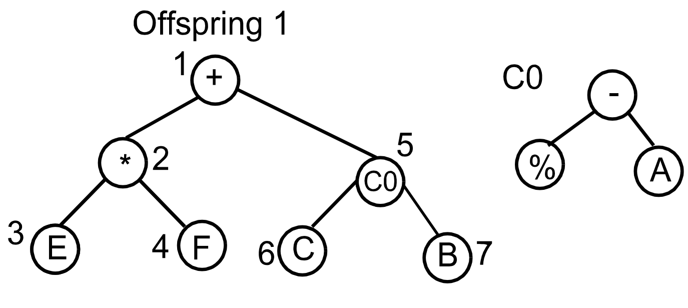
The compress operator performs a function similar to that of the encapsulation operator. An individual is randomly selected and part of the subtree up until a certain depth is defined as a module. The nodes at the level below the cut-off depth are considered as arguments to the module. The module is named and added to the function set. The following example illustrates this process.

Example:

Suppose that the following tree was chosen as a parent and the subtree rooted at node 5 was randomly chosen as the encapsulation point.



If the cutoff depth is 1, the offspring produced is:



2.2 Automatically Defined Functions

This section examines the use of ADFs as a mean of inducing modular programs. The following section describes the architecture used by Koza [7] to incorporate the use of ADFs into algorithms induced while section **2.2.1** looks at the architecture implemented by Bruce[3] for this purpose. Section **2.2.2** looks at the different ways in which functions can be called. A critical analysis of ADFs is provided in section **2.2.3**. The applications of the ADF system introduced by Koza is presented in section **2.2.4**.

2.2.1 Architecture Implemented by Koza

Koza introduces the concept of subroutines within a program by means of automatically defined functions. Automatically defined functions enable a genetic programming system to solve a problem by decomposing it into subproblems. The GP system implemented by Koza simultaneously induces a main program together with the subroutines called by the main program.

Each individual of the population is represented by a tree containing one or more function-defining (function) branches and a results-producing branch (main program). The results-producing branch can call the functions defined by the function-defining branches.

The example tree in **Figure 2.2.1.1** consists of one function-defining branch and one results-producing branch. This tree represents a program in Lisp. Koza describes the first six points in this tree as invariant and the remaining two as non-invariant. Koza defines the different types of nodes as follows:

1. The first node in the tree is the root of the tree. In this case the Lisp connective *progn* is the root of the tree and combines together the statements representing the automatically defined function and the main program.
2. The Lisp primitive *defun* which marks the beginning of each function in Lisp.
3. The name of the automatically defined function.
4. The argument list of the automatically defined function.
5. Returns value/s outputted by the automatically defined function.
6. Returns value/s outputted by the result-producing branch.
7. The body of the automatically defined function.
8. The body of the results-producing branch.

If more than one value is returned by the results-producing branch, a number of sub-branches exist under the *values* node in the results-producing branch. The actual variables of the problem (input values of the fitness case) are used in the results-producing branch and can be passed to the function defining branches.

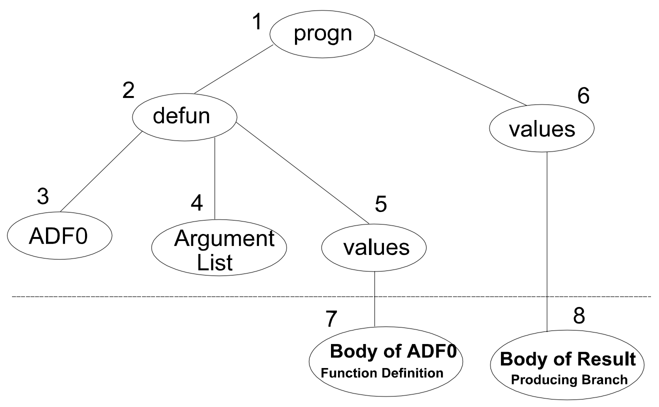


Figure 2.2.1.1

A function-defining branch may refer hierarchically to any other function that was defined before it. Furthermore, a function-defining branch may make recursive calls to itself. **Figure 2.2.1.2** illustrates an example program.

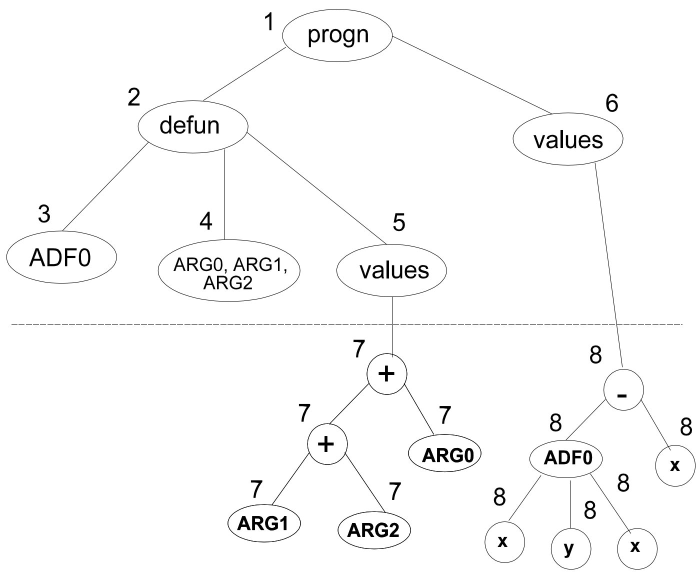


Figure 2.2.1.2

When incorporating the use of ADFs into a GP system the user has to specify the number of function-defining branches and the number of arguments that each ADF will take. If each individual contains more than one ADF the user will have to specify legal calls between functions (see section **2.2.3**). For each of the function-defining branches and the results-producing branch a terminal and a function set must be specified. Koza suggests that the number of variables of the problem should be used as an upper limit for the number of arguments that an ADF can take. The method used to generate the initial population must be edited to create individuals that are composed of a results-producing branch and one or more function-defining branches.

The crossover operator needs to be edited in order to ensure that crossover is performed on like branches, i.e. structure-preserving crossover must be applied to individuals so that the offspring produced are syntactically correct. The invariant components of each individual are not altered during crossover, i.e. structure-preserving crossover is restricted to the non-invariant points of the individual.

Each non-invariant point is assigned a type. Structure-preserving crossover is implemented as follows:

- In the first parent a non-invariant point is randomly chosen.
- A point in the second parent is randomly chosen from the points of the same type as the point chosen in the first parent.

To ensure that genetic operators are applied to the same branches in each individual, types have to be assigned to the nodes in each individual. One of two methods can be used to assign types to the nodes:

- Branch typing - There is one type for each branch. A different type is assigned to the non-invariant points of each branch.
- Point typing - Each individual non-variant point in the overall individual is assigned a type. Each type has a function set, a terminal set, a argument map, and syntactic constraints of the branch where the node is located, associated with.

Branch typing is illustrated in **Figure 2.2.1.2**. The function-defining branch is of type seven when the results-producing branch is of type eight.

Experiments conducted by Koza have indicated that the effect of using different architectures with ADFs, e.g. individuals with four ADFs with two arguments each, individuals with three ADFs with four arguments each, only effects the number of fitness evaluations and not necessarily the success of the GP system.

2.2.2 Architecture Implemented by Bruce

Bruce suggests that instead of using a single tree to represent the main program and the ADFs each individual should consist of n trees stored in a fixed-size array, one representing the main program and the n-1 trees representing the n-1 ADFs. The user has to specify the number of trees in each genotype. The user must also specify how the evaluation of each tree will contribute to calculating the overall fitness of the individual.

This representation is simpler than that implemented by Koza. As we are now dealing with separate trees and not a single tree with constrains on which branches are modifiable, the genetic operators do not need to be adapted. This representation is more general, e.g. the genetic programming system without the use of ADFs is essentially this system with the size of the genotype being one. **Figure 2.2.2.1** illustrates an example program.

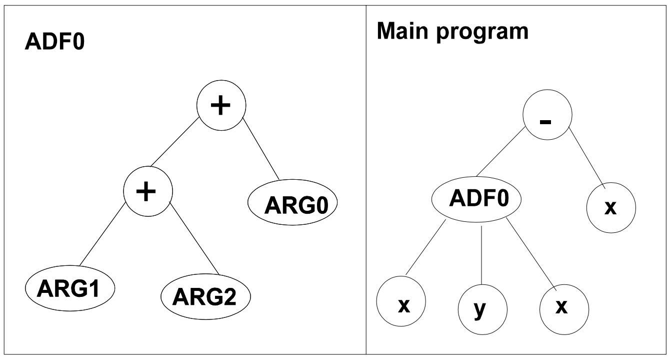


Figure 2.2.2.1

The user has to specify the size of each genotype and define legal function calls (see next section). During initial population generation each individual must be developed as an array of trees instead of a single tree. Although the generic operators do not have to be changed in the way necessary in Koza's implementation, they have to be extended to now deal with each individual containing more than one tree. A tree in the first parent is randomly selected. The tree at the same index in the second parent is chosen and the crossover operator is applied to both the selected trees.

2.2.3. Function Calls

In computer programs one function can usually call other functions that have been defined. If there is more than one function-defining branch (or tree in a genome) legal function calls can be defined as follows:

- There are no references between the function-defining branches.
- References among function-defining branches are not restricted. In this case an ADF can call itself recursively either directly or indirectly and the name of the ADF is added to its function set.
- A function may hierarchically call those functions that have been defined before it, e.g. ADFO can call ADF1 but not vice versa. These ADFs are also referred to as hierarchical automatically defined functions.

2.2.4. Critical Analysis

- The use of ADFs is not beneficial for simple problems. ADFs are beneficial for complex problems. ADFs produce parsimonious solutions for difficult problems.
- One of the disadvantages of using ADFs is that the user has to define the entire architecture prior to a simulation.
- The benefits of using ADFs increases with the difficulty of the problem.
- The use of automatically defined functions decreases the computational effort needed to find a solution as well as increases the parsimony of solutions provided that the problem is of sufficient difficulty.

- A GP system incorporating the use of ADFs has the lens effect, i.e. a GP system with ADFs tend to find individuals that have extreme scores.

2.3 Applications

This section presents two examples in which the GP system has been extended to include the use of ADFS.

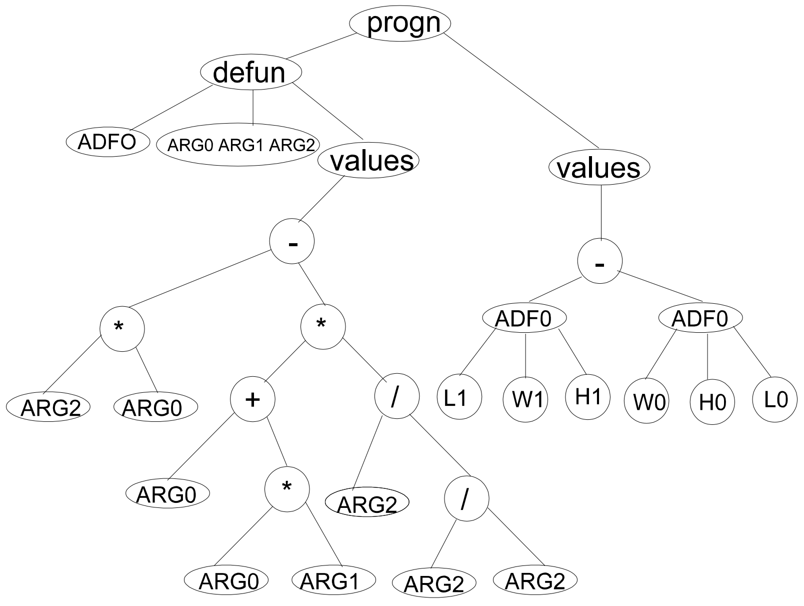
2.3.1 Function Fitting

Function to generate:	Induce a program that produces the value of the independent value D given the six independent variables L0, W0, H0, L1, W1 and H1.
Architecture of overall program:	One result-producing branch;one function-producing branch defining the ADF ADF0 with has three arguments.
Typing:	Branch typing
Terminal set for the results-producing branch:	T_r = {L0, W0, H0, L1, W1, H1 }
Function set for the results-producing branch:	F_r = {+, -, *, /, ADFO }
Terminal set for the function-producing branch:	The three dummy variables T_f = { ARG0, ARG1, ARG2}
Function set for the results-producing branch:	F_f = {+, -, *, /}
Number of generations:	51 - Different fitness cases are used for each generation.
Population size:	4000
Raw fitness:	The sum of the absolute value of the differences between the value produced by the induced program and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the value calculated by the induced program is within 0.01 of the target value.
Success Predicate:	A program that scores ten hits are found.

Genetic operators:	Crossover - 90% Reproduction - 10% Mutation - 0%
Method of selection:	Fitness proportionate selection.
Initial population generator:	The ramped half-and-half method with an initial tree depth of six and a depth limit of seventeen on the size of trees created by the genetic operators.
Fitness cases:	Ten fitness cases are used for each generation. An example set of fitness cases is:

	L_0	**W_0**	**H_0**	**L_1**	**W_1**	**H_1**	**D**
1	3	4	7	2	5	3	54
2	7	10	9	10	3	1	600
3	10	9	4	8	1	6	312
4	3	9	5	1	6	4	111
5	4	3	2	7	6	1	-18
6	3	3	1	9	5	4	-171
7	5	9	9	1	7	6	363
8	1	2	9	3	9	2	-36
9	2	6	8	2	6	10	-24
10	1	10	7	5	1	45	-155

One of the solutions found is illustrated below. This solution was found by generation 13.

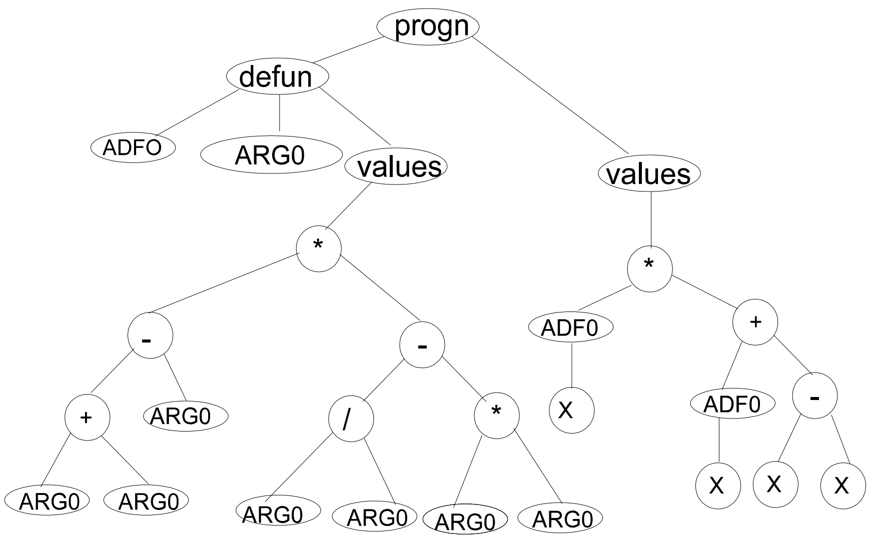


2.3.2 6th Power Polynomial

Function to generate:	Induce a program that produces the value of the independent value of the polynomial $x^6 - 2x^4 + x^2$ when given the a value of the independent variable x.
Architecture of overall program:	One result-producing branch; one function-producing branch defining the ADF ADF0 that has one argument.
Typing:	Branch typing
Terminal set for the results-producing branch:	T_r = {X, $\Re$ } where $\Re$ is in the interval [-1.0, 1.0].
Function set for the results-producing branch:	F_r = {+, -, *, /, ADFO }
Terminal set for the function-producing branch:	The three dummy variables T_f = { ARG0, $\Re$ } where $\Re$ is in the interval [-1.0, 1.0].
Function set for the results-producing branch:	F_f = {+, -, *, /}
Number of generations:	51 - Different fitness cases are used for each generation.
Population size:	4000

Raw fitness:	The sum of the absolute value of the differences between the value produced by the induced program and the target value specified in the fitness case.
Hits criterion:	The number of fitness cases for which the value calculated by the induced program is within 0.01 of the target value.
Success Predicate:	A program that scores 50 hits.
Genetic operators:	Crossover - 90% Reproduction - 10% Mutation - 0%
Method of selection:	Fitness proportionate selection.
Initial population generator:	The ramped half-and-half method with an initial tree depth of six and a depth limit of seventeen on the size of trees created by the genetic operators.
Fitness cases:	Fifty values are chosen randomly from the interval [-1.00, 1.00]

The following solution tree during generation ten of a run:



In this example the use of ADFS resulted in a decrease in the computational effort needed to find a solution. However, the average structural complexity still remained higher with the use of ADFs.

3. Architecture-Altering Operations

This section describes the architecture-altering operators defined by Koza [8]. These operators are applied to individuals that are comprised of one results-producing branch and one or more automatically defined functions (ADFs), automatically defined loops (ADLs), automatically defined iterations (ADIs), automatically defined recursions (ADRs) and automatically defined stored (ADSs). A separate set of architecture-altering operations exist for ADFs, ADLs, ADIs , ADRs and ADSs. Each set of operators contains an operator to create an instance of the particular automatically defined construct, delete an existing instance of the construct, create a new argument of an existing construct, delete an argument of an existing construct, duplicate an existing construct and an operator that duplicates an argument of an existing construct.

These six operators essentially perform the same function for all the automatically defined constructs thus we will only examine these operators for one construct only, namely, automatically defined functions. There are essentially six ADF architecture-altering operations:

- Subroutine duplication
- Argument duplication
- Subroutine creation
- Argument creation
- Subroutine deletion
- Argument deletion

3.1. Subroutine duplication

This operator changes the architecture of an individual by adding a new ADF branch to the individual, namely, a copy of an existing branch. An individual is firstly selected from the population using one of the selection methods. Selection is with replacement. If the individual does not contain any ADFs another individual can be chosen or the operator can be reduced to the reproduction operator. The branch to be duplicated is randomly chosen. A copy of the chosen branch is added to a copy of the parent. The copied branch is given a new ADF name, e.g. ADF1.

Functions calls in the results-producing branch to the copied ADF is randomly distributed between the copied function and the new function-defining branch.

The copied branch and the new branch have the same terminal set. The function set of every branch that contains the name of the copied branch is extended to also include the name of the newly created branch. The function set of the copied branch can be extended to include the name of the newly created ADF.

The following GP parameters must be specified by the user:

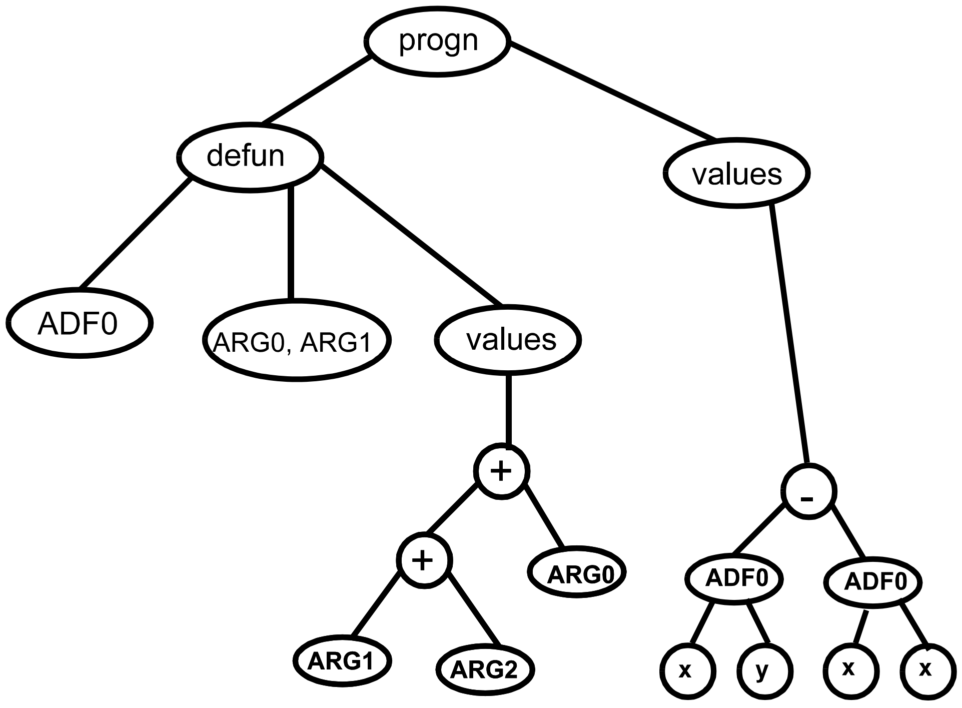
- Subroutine duplication application rate.

- The maximum number of ADFs that an individual can contain.

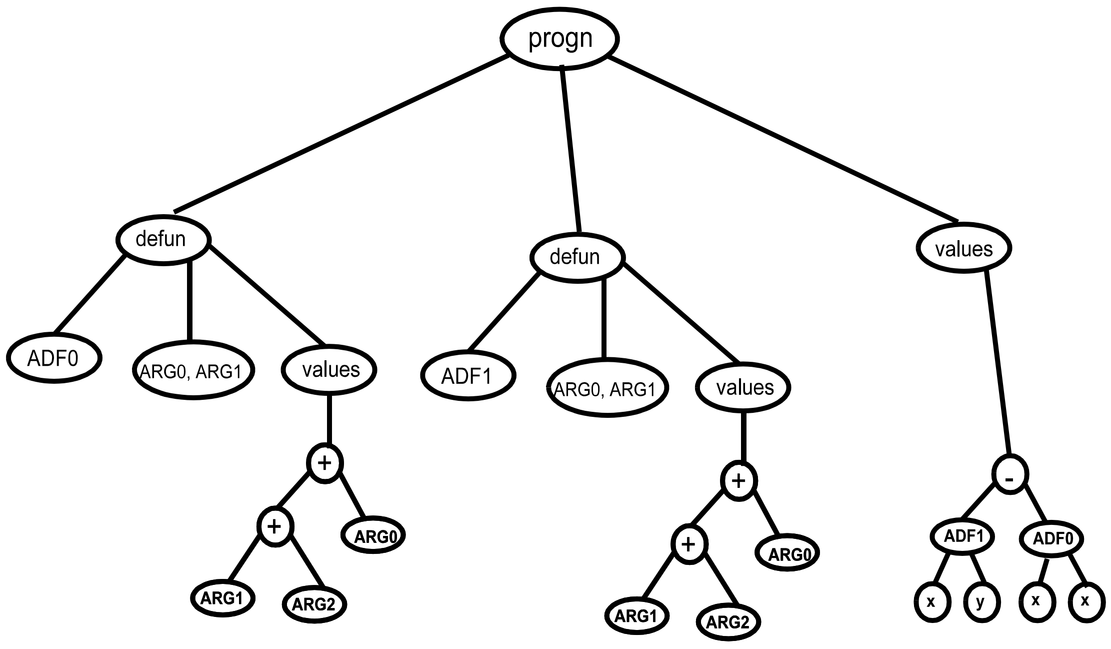
If applying this operator will result in the offspring exceeding the maximum number of ADFs permitted, the parent is copied into the next generation, i.e. the reproduction operator is applied to the chosen parent.

Example:

Parent:



Offspring



3.2. Argument Duplication

Argument duplication increases the number of arguments of a chosen ADF branch by one. A parent is firstly selected from the population using a standard selection method. Selection is performed with replacement. A copy of the parent is made. A function-defining branch is randomly selected. If the individual does not contain any ADFs another individual can be chosen or the operator can be reduced to reproduction.

An argument is randomly chosen from the selected branch. If the selected ADF has an arity of zero, reproduction is performed instead of argument duplication. A name for the new argument is added to the list of arguments of the selected branch. Each occurrence of the copied argument in the selected branch is randomly chosen to be left as is or replaced with the new argument.

The function calls to the chosen ADF in the results-producing branch need to be updated to cater for one more argument. A copy of the branch representing the copied argument represents the new argument in each function call in the results-producing branch.

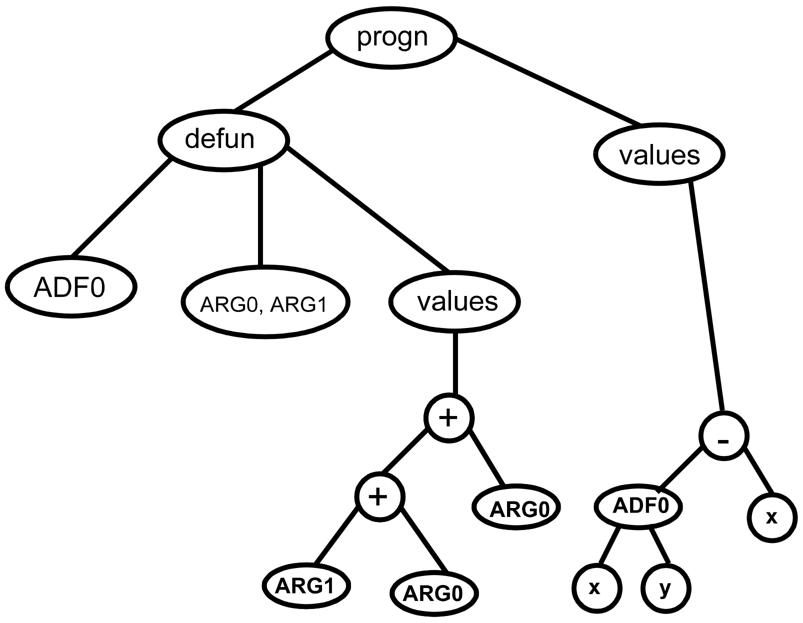
The terminal set of the chosen ADF branch is extended to include the new argument. Each function set that contains the ADF must be edited to reflect an increase of the arity of the ADF.

The user needs to specify the following GP parameters:

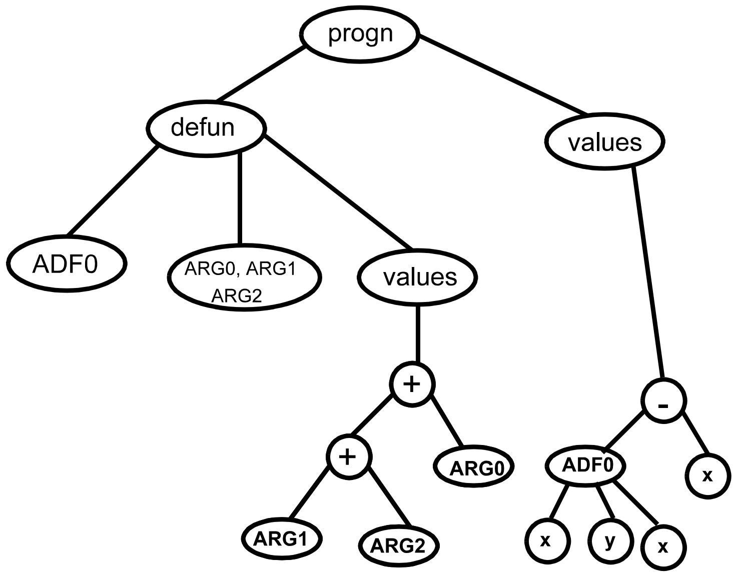
- The application rate of the argument duplication operator.
- The maximum number of arguments per ADF.
- The maximum size of the results-producing branch

Example:

Parent:



Offspring:



3.3. Subroutine Creation

The subroutine creation operator changes the architecture of a copy of the chosen parent by creating a new ADF branch. A parent is firstly selected using a standard selection method. Selection is performed with replacement. A node in the body of any of the function-defining branches or results-producing branch is randomly chosen. This node will be referred to as N. The chosen node will form the root of the body of the new ADF branch. The chosen node will also be replaced with the new ADF label. The subtree rooted at the chosen node is traversed to determine the arity of the new ADF. During the traversal each node is randomly chosen as being an argument or not of the new ADF (in the branch containing N). If a function node is chosen as an argument, its children are not visited. Zero or more points maybe chosen.

The new function branch is given a unique name, e.g. ADF1. The arity of the new ADF is equal to the number of nodes randomly chosen during the traversal. N is replaced with the name of the newly defined ADF and the arguments chosen during the traversal form the arguments of this node. The body of the ADF is a modified version of the subtree rooted at N. Each node (and its corresponding subtree) in the subtree chosen during the traversal to be an argument, is replaced by the corresponding dummy variable (e.g. ARG0, ARG1, etc).

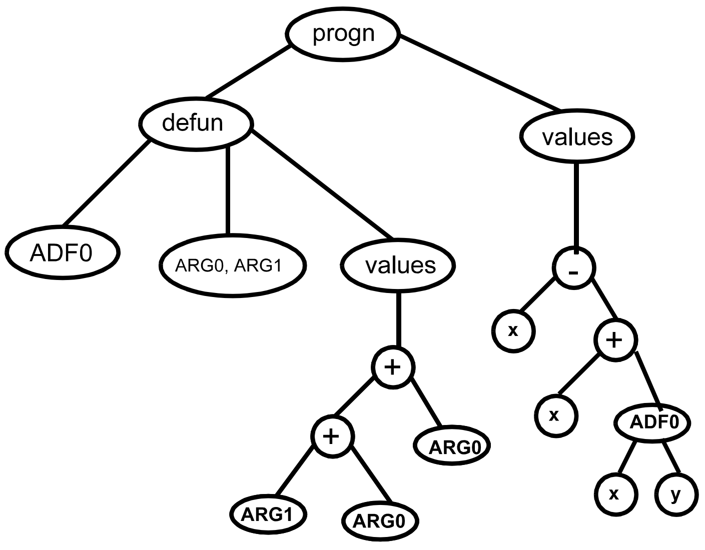
The terminal set of the newly created branch is comprised of the terminal set of the branch that contained N and the dummy variables for the newly defined ADF. The function set of this branch is the same as that of the branch that contained N. The function set of the branch that contained N is extended to include the name of the new ADF. The function sets of those branches that contain the label of the branch N was contained in, is extended to include the name of the newly created ADF.

The user needs to specify the following GP parameters:

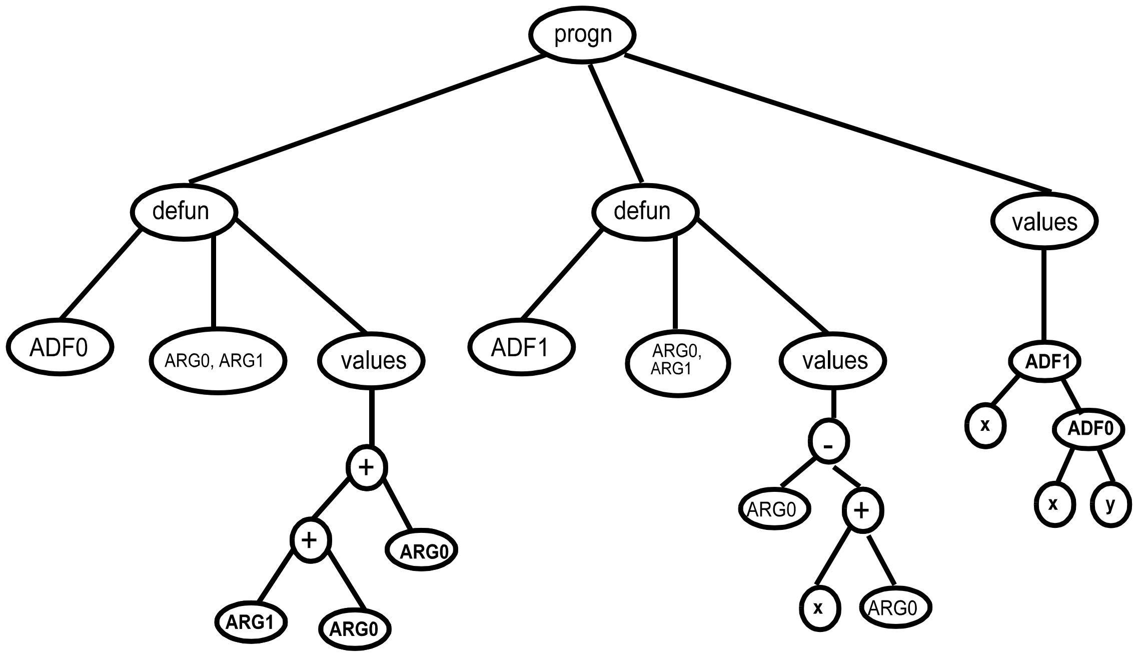
- Application rate of the subroutine creation operator.
- The maximum number of ADFs per individual.
- The maximum size of an ADF
- The maximum number of arguments per ADF.
- The minimum number of arguments per ADF.

Example:

Parent:



Offspring:



3.4. Argument Creation

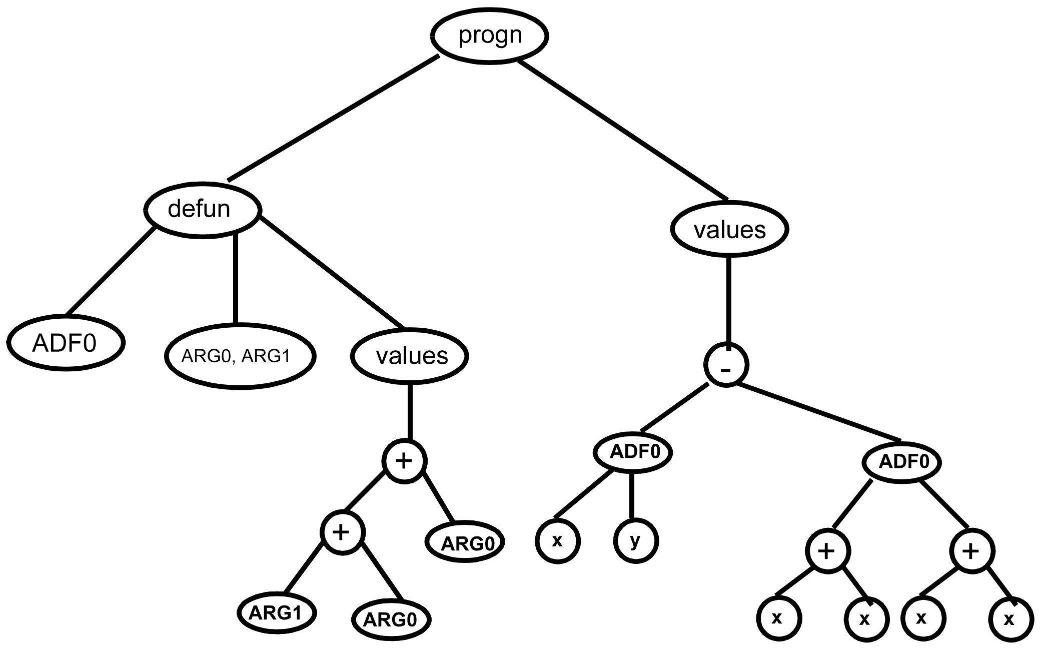
The argument creation operator creates a new argument for an existing ADF in a copy of the chosen parent. A parent is chosen using a standard selection method. Selection is performed with replacement. A node is selected in the body of a function-defining branch. This node will be referred to as N. An argument is added to the argument list of the branch containing N. N , and the subtree rooted at N, is replaced with the new argument label. Every occurrence of the ADF, that contains N, in the individual must be updated to include another subtree representing the new argument. The additional subtree consists of the subtree rooted at N, with the dummy variables replaced with the subtrees representing each dummy variable for that particular occurrence of the ADF. The terminal set of the branch containing N is extended to include the new argument.

The user has to specify the following GP parameters:

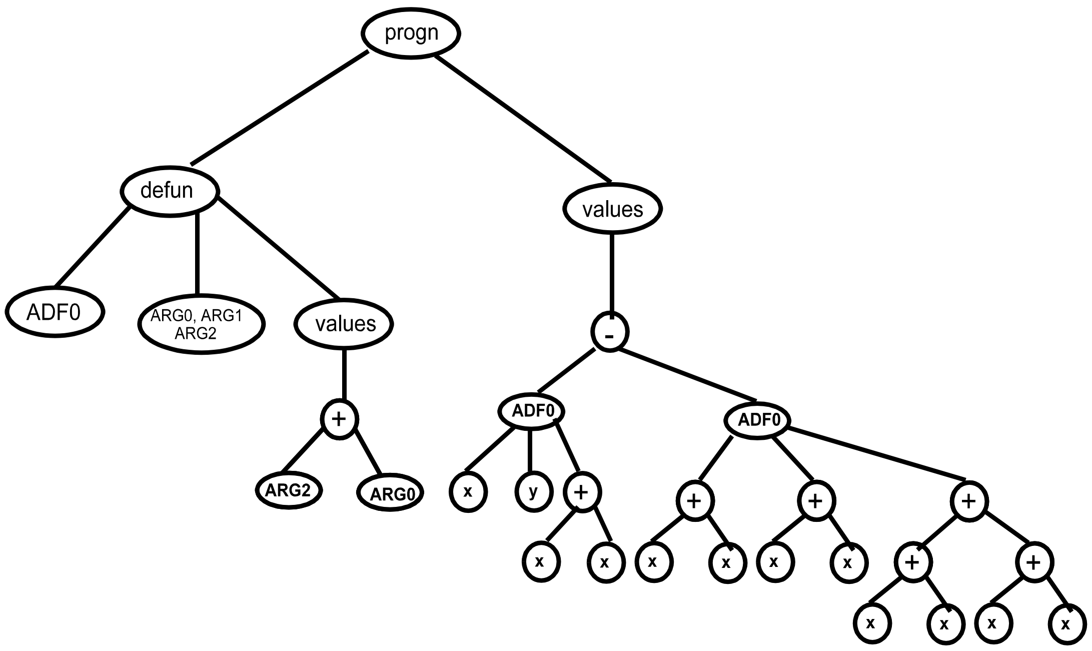
- The application rate of the argument creation operator.
- The maximum number of arguments per individual.
- The maximum size of the results-producing branch.

Example:

Parent:



Offspring:



3.5. Subroutine Deletion

This operator deletes an existing ADF branch from a copy of the chosen parent. A parent is selected, with replacement, using a standard selection method. If the selected parent does not contain any ADFs or this operation will result in the offspring containing less than the minimum number of ADFs (minimum default is 1) another parent is selected or reproduction is performed instead. A function-defining branch is randomly selected to be deleted.

Once the branch is deleted function calls to the deleted ADF must be distributed between the remaining ADFS. One of three methods can be used for this purpose, namely, subroutine deletion by consolidation, subroutine deletion with random regeneration and subroutine deletion by macro expansion.

In order to perform subroutine **deletion by consolidation** the chosen parent must have at least two ADF branches. A remaining ADF is randomly chosen to replace the function calls to the deleted ADF. If the chosen ADF has the same number arguments as the deleted ADF each function call is replaced with the label of the chosen ADF. If the number of arguments of the replacement ADF is less than that of the deleted ADF each function call is replaced with the label of the chosen ADF and the subtrees representing the additional arguments are deleted.

On the other hand if the number of arguments of the replacement ADF is greater than that of the deleted ADF the additional subtrees can either be randomly generated or the subtrees representing the other arguments can be copied.

Subroutine deletion with random generation firstly replaces function calls to the deleted ADF with the label of a randomly chosen remaining ADF.

Subtrees for each of the arguments of the replacement ADF is randomly generated using the same process as initial population generation.

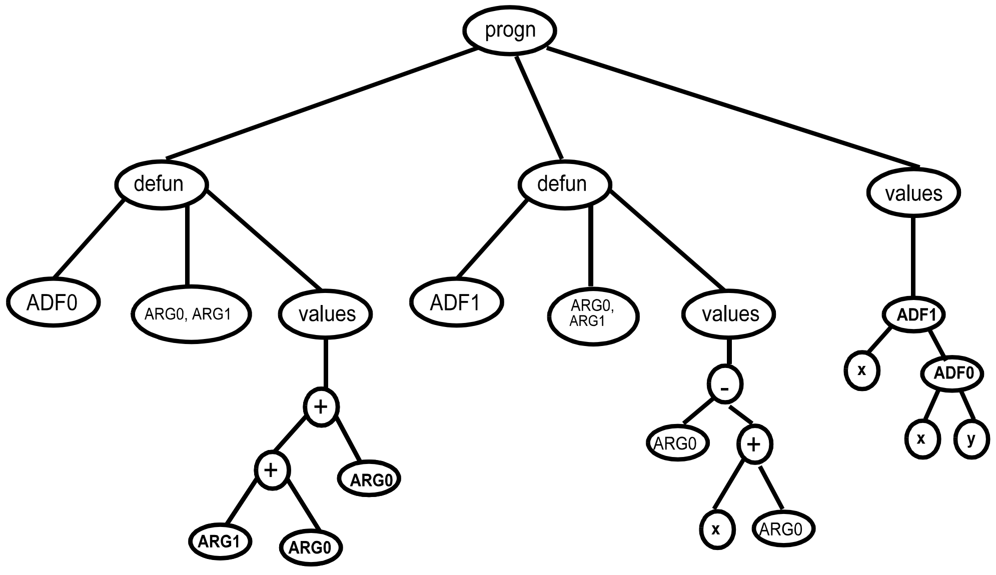
During **subroutine deletion by macro expansion** each function call of the deleted ADF is replaced with the body of ADF with the dummy variables replaced by the subtrees representing them in the function call.

The user needs to specify the following GP parameters:

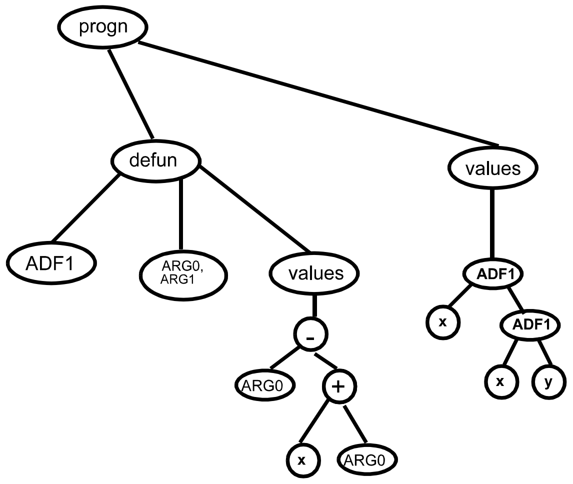
- Application rate of the subroutine deletion operator.
- The minimum number of ADFs per individual.
- Method to use to replace function calls to the deleted ADF, i.e. subroutine deletion by consolidation, subroutine deletion with random regeneration, or subroutine deletion by macro expansion.

Example 1: Subroutine deletion with consolidation - Replacement ADF has the same number of arguments as the deleted ADF.

Parent:

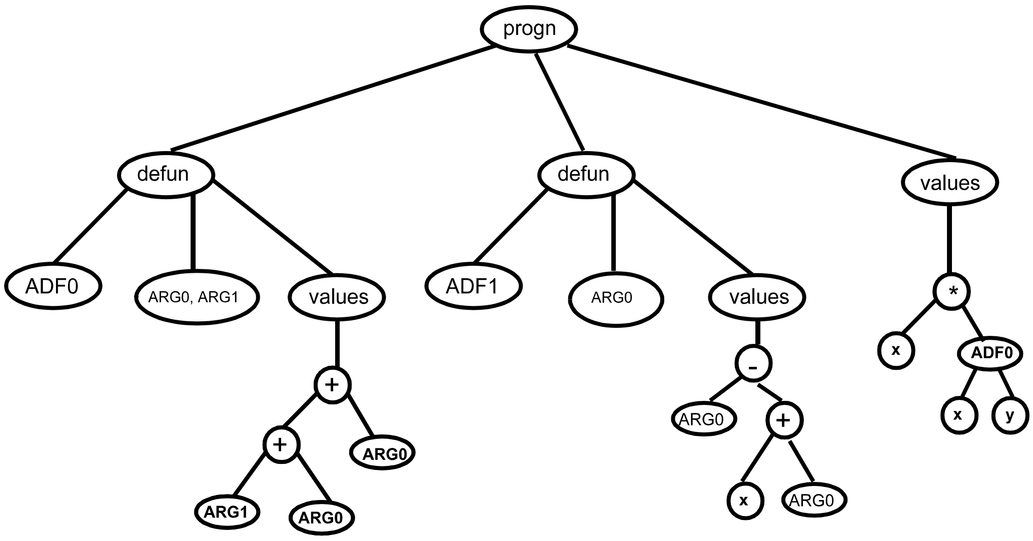


Offspring:

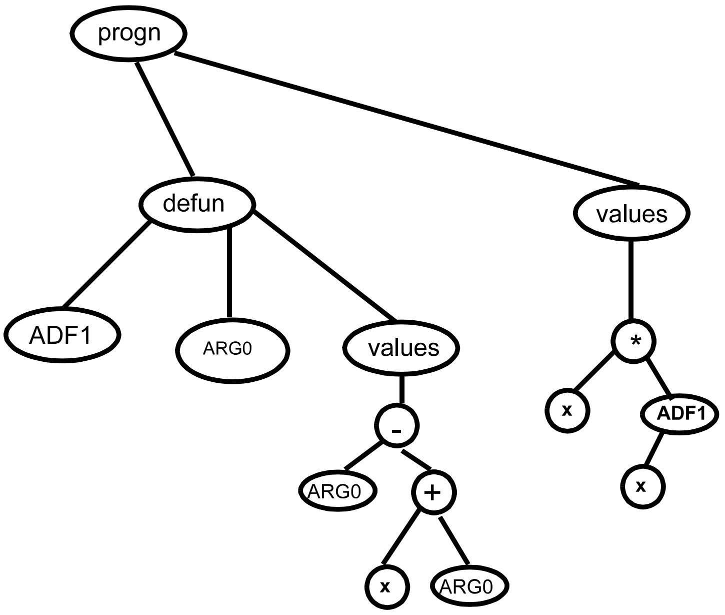


Example 2: Subroutine deletion with consolidation - Replacement ADF has less arguments than the deleted ADF.

Parent:

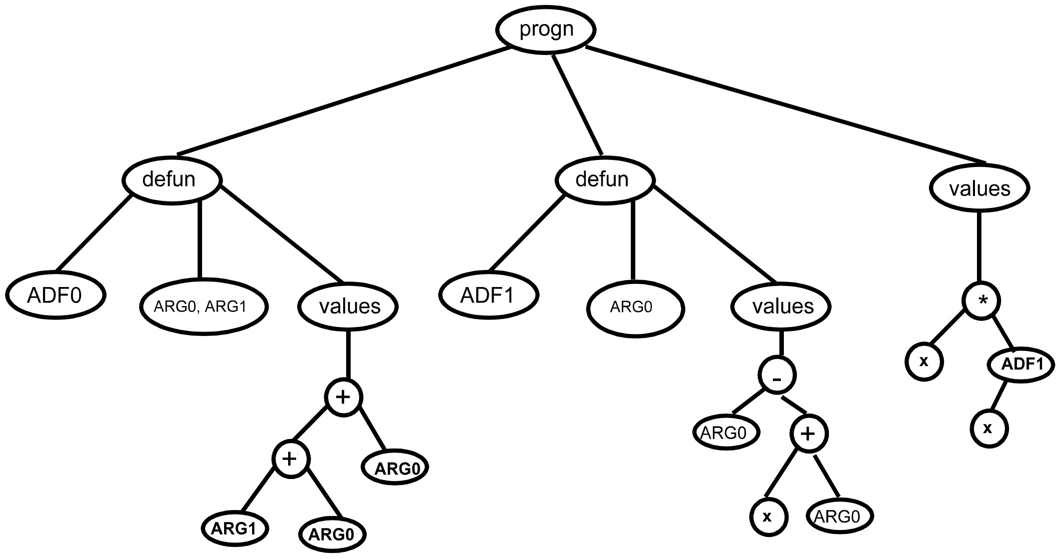


Offspring

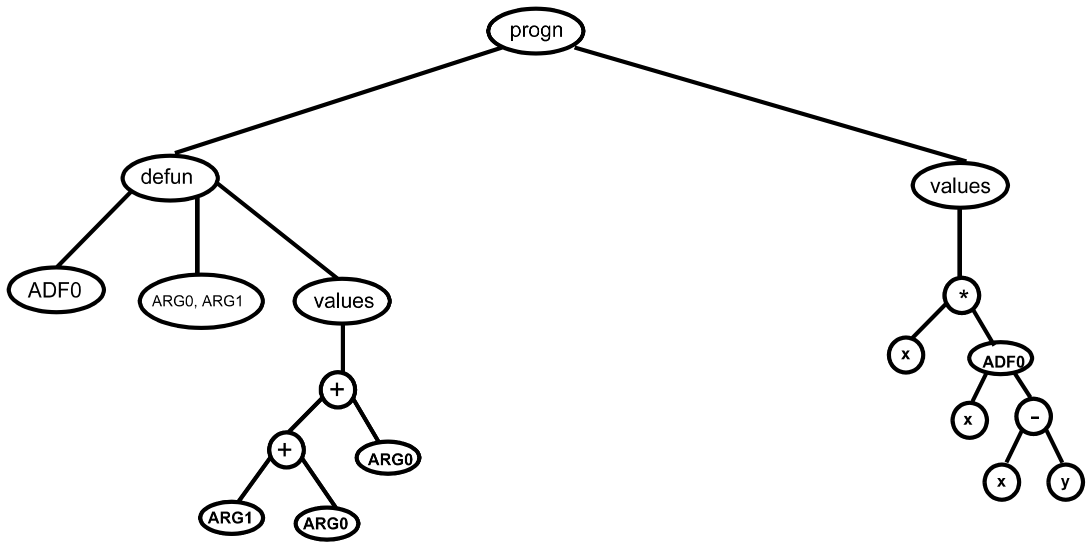


Example 3: Subroutine deletion with consolidation - Replacement ADF has more arguments than the deleted ADF.

Parent:

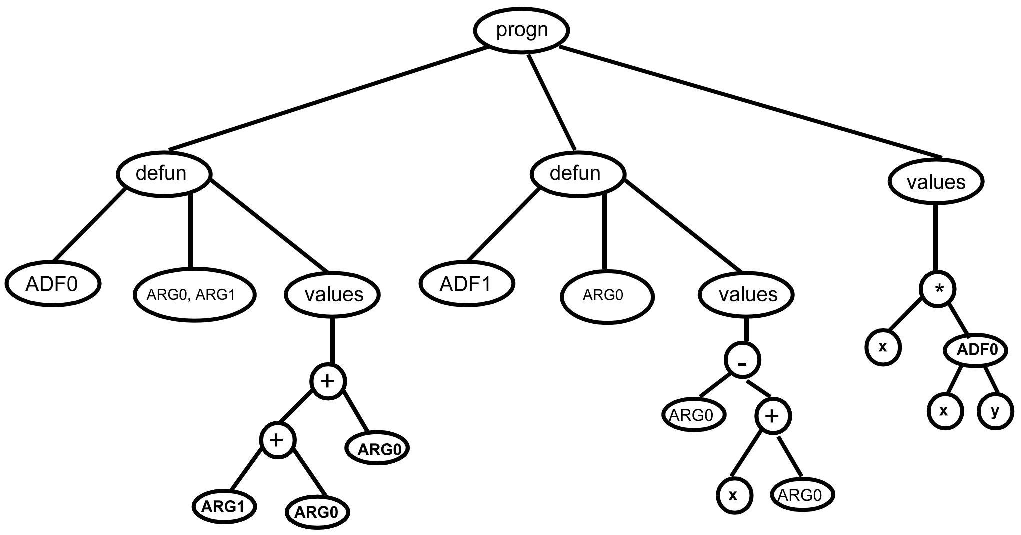


Offspring:

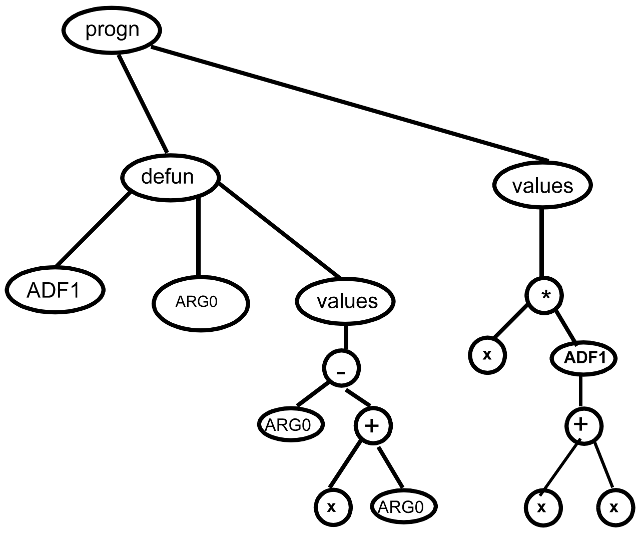


Example 4: Subroutine deletion with random regeneration

Parent:

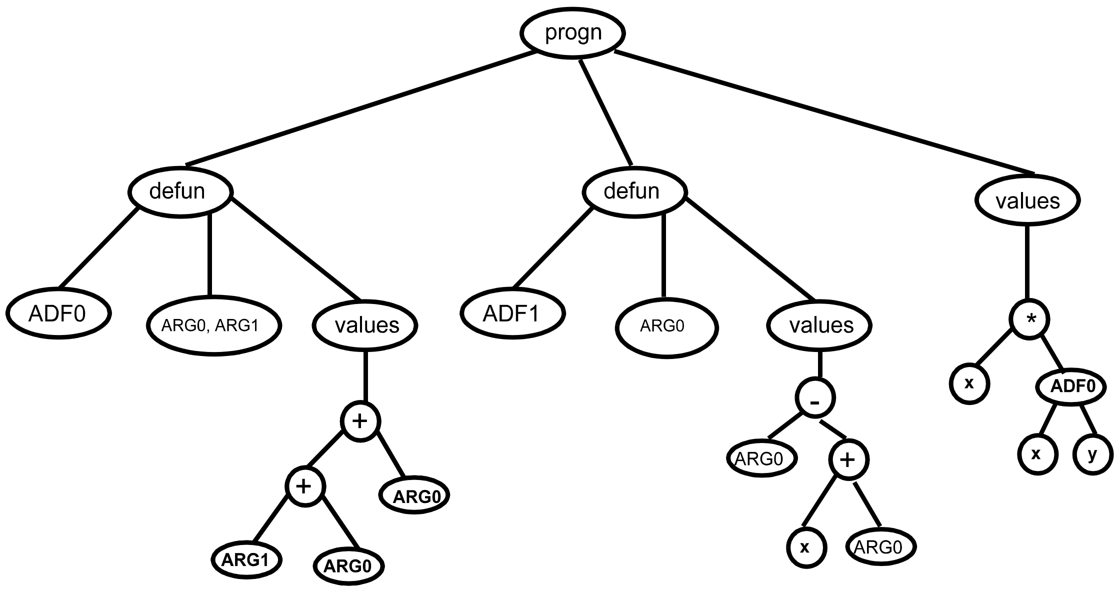


Offspring**:**

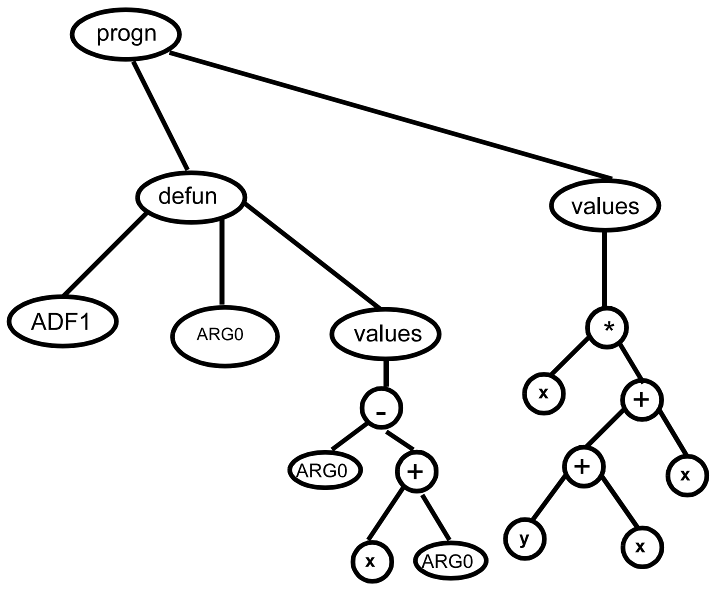


Example 5: Subroutine deletion with macro expansion

Parent:



Offspring:



6. Argument Deletion

The argument deletion operator removes one of the arguments of an ADF branch in the copy of the selected parent. A parent is selected, with replacement, using a standard selection method. An ADF branch in the copy of the parent is randomly selected.

If the selected parent does not contain any ADFs another parent is selected or reproduction is performed instead. An argument of the chosen ADF is randomly selected. The chosen argument is deleted from the argument list of the ADF. In each function call to the ADF, the subtree representing the deleted argument is also deleted. For each occurrence of the argument in the body of the ADF the argument is replaced with an existing argument. One of the three methods is used for this purpose, namely, argument deletion by consolidation, argument deletion with random regeneration, or argument deletion by macro expansion.

During the process of **argument deletion by consolidation** an argument of the ADF is randomly chosen to replace the deleted argument. This argument replaces every occurrence of the deleted argument in body of the ADF.

The **argument deletion with random regeneration** replaces each occurrence of the deleted argument with a newly generated subtree. The terminal and function set used to create the subtree is that of the branch from which the argument was deleted.

The process of **argument deletion by macro expansion** makes a copy of the branch containing the argument to be deleted for each function call to the ADF. These copies form new ADF branches and are given unique ADF labels.

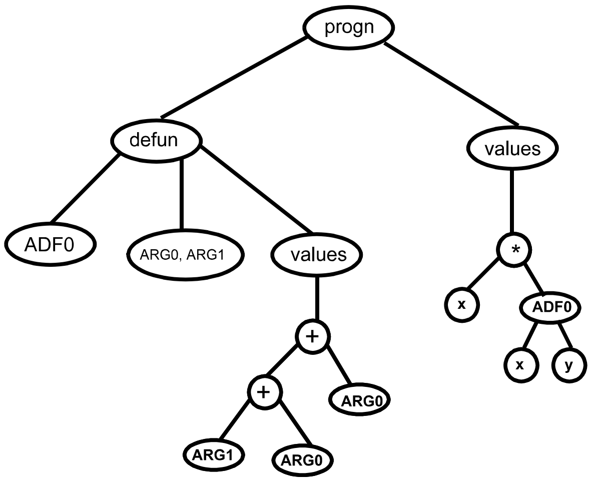
Each function call of the ADF that contains the argument to be deleted is replaced with the label of one of the newly created ADF branches. The subtree corresponding to the deleted argument in the particular occurrence replaces the argument label in the body of the corresponding newly created ADF branch.

The user has to specify the following GP parameters:

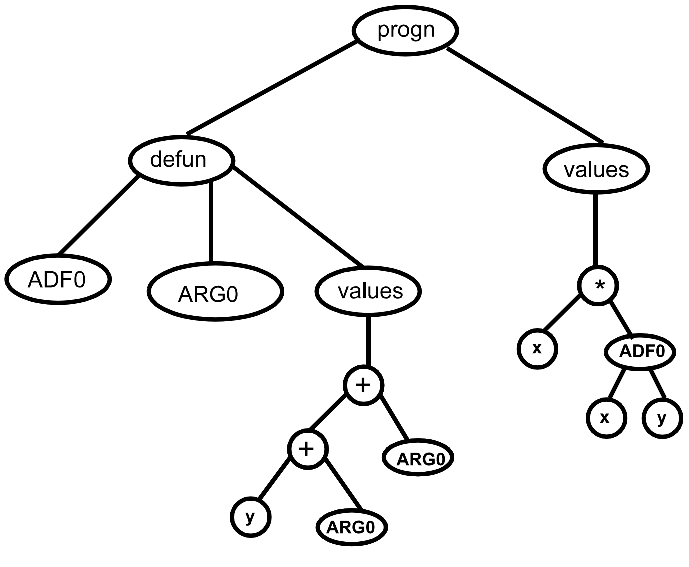
- Application rate of the argument deletion operator.
- The minimum number of arguments that an ADF branch must contain.
- The methodology used to replace occurrences of the deleted argument in the body of the ADF, i.e. subroutine deletion by consolidation, subroutine deletion with random regeneration, or subroutine deletion by macro expansion..

Example: Argument deletion by macro expansion

Parent:



Offspring:



Chapter 4 - Problems Associated with Genetic Programming Systems

There are two main problems associated with genetic programming. The first is its susceptibility to converge prematurely to a local optimum. This is discussed in section **1**. The second is the production of bloat and introns as a result of the evolutionary process. Bloat and introns are discussed in section **2.**

1. Premature Convergence

In order for the genetic programming system to find a solution it has to converge to individuals of a particular structure with the fitness of the individuals improving from one generation to the next. In some cases the algorithm converges to an overall structure that appears to be fit but cannot be improved on successive generations and will not lead to a solution. In this case we say that the genetic programming algorithm has converged prematurely to a local optimum. This section discusses the possible causes of premature convergence and describes preventative measures that can be taken.

1.1 Lack of Diversity

One of the main causes of premature convergence is the lack of diversity of the initial population and/or the population on successive generations. These populations may not contain the structures necessary for a solution to be found. This is especially so if the population contains a number of duplicates or similar individuals.

One can ensure that the initial population is genetically diverse by not allowing duplicate individuals. This restriction can also be placed on successive generations, however it should be remembered that too diverse a population on successive generation can result in the genetic programming algorithm not converging at all. Thus, this restriction is often placed in the initial population only.

During the evolutionary process higher mutation application rates and the use of the creation operator can re-introduce important building blocks into the population when the local search is not successful.

Some studies ([5] and [10]) check on the diversity of the population after N generations. The diversity is measured by introducing a similarity index to determine how similar individuals in the population are to each other. If the percentage of similar individuals exceeds a specified threshold a percentage of dissimilar individuals are introduced into the population.

1.2 Selection Variance or Noise

Most of the choices made during the evolutionary process, e.g. choosing nodes during initial population generation, is fairly random. These choices result in the GP algorithm converging to a particular structure.

The random choices made could result in the algorithm converging prematurely. However, if a consecutive run is made for the same seed, different random choices would be made on this run which could lead to a solution being found. Thus, one way to deal with selection noise or variance is to perform multiple consecutive runs for the same seed. An alternative to this is to develop a number of populations in parallel. Instead of performing a number iterations sequentially to find the global optimum, a population can be divided into a number of subpopulations which run in parallel to speed up the process. The subpopulations communicate by means of interbreeding.

1.3 Destructive Crossover

Crossover has been described as being both constructive and destructive. One of the destructive effects of crossover is that it breaks up good building blocks that could form part of a solution. Another destructive effect is that it may insert a good building block into an individual that does not make proper use of it.

Experiments conducted by Banzhaf et. al. [2] have revealed that crossover is destructive to offspring approximately seventy five percent of the time. The closer a tree is to a solution the more susceptible it is to the destructive effects of crossover. Experiments conducted indicate that the percentage of destructive crossover remains high until the end of a run.

In addition to the terms constructive and destructive crossover, the term neutral crossover is defined to describe a crossover operator that produces offspring that have a fitness value that is within +-2.5 of the fitness value of its parents.

Due to the destructive effects of crossover the application of the crossover operator must be monitored during a GP run. Two methods which be used to measure the effect of crossover:

- The average fitness of the parents must be compared with the average fitness of the offspring.
- The fitness of one parent is compared with the fitness of one child. However, this approach presents the problem of how the child and parent should be chosen.

Improvements of the crossover operator include:

- The Macromutation Operator

  The macromutation operator, also known as “headless chicken crossover”, is an improvement over the standard crossover operator. The macromutation operator selects a single parent. It then generates a new individual randomly. The selected individual is crossed over with the new individual to produce a single offspring. If the offspring has a fitness value greater than or equal to the fitness of its parent it forms part of the new population, else it is discarded.

The macromutation operator has not been widely tested and has been successfully applied to only one problem domain.

- Brood Recombination

  Brood selection reduces the destructive effects of crossover. Brood selection requires that a brood size N be specified. Two parents are selected. Standard crossover is applied to the parents N times creating N pairs of offspring. The offspring is evaluated for fitness and sorted according to their fitness values.

  The best two offspring are selected as the result of brood selection and the rest of the offspring are discarded. One of the disadvantages of this method is the time required to evaluate all 2N offspring.

- Researchers are of the opinion that the problems experienced with the crossover operator can be attributed to the fact that there are differences between the biological crossover and GP crossover. Biological crossover is only applied to two individuals that are homologous with respect to their function and structure. However, GP crossover is applied to any two individuals.

- Intelligent Crossover

  An intelligent crossover operator is used in the PADO system developed by Teller et. al. [15]. This operator learns how to choose good crossover points. The intelligent crossover operator resulted in an increase in the performance of the PADO system.

  Another form of the intelligent crossover described by Banzhaf et. al. [2] involves calculating a performance evaluation for each subtree. This performance value is used to determine which subtrees to insert into other trees, and which to replace. This form of the crossover operator improved the performance of the system “substantially”. Banzhaf et. al. [2] state that the result of using intelligent crossover is not as good as that obtained using brood selection.

- Context-Sensitive Crossover

  The crossover operator is not context preserving. The application of a strong context preserving crossover operator in combination with the standard crossover operator in the system developed by D’haeeleer resulted in an improvement in system performance.

- Explicitly Defined Introns

  This approach involves adding introns to an individual to reduce the destructive effects of crossover. These introns are called EDIs, explicitly defined introns and usually take the form of real or integer values inserted between every two nodes in an individual.

1.4 Cloning

The use of crossover together with a depth limit can result in the crossover operator merely producing clones of the parents and hence causing the GP process to stagnate. Research has proven that crossover produces more duplicates shortly after each new improved individual is found. Two types of crossover reduce the variety of a population. Crossover which swaps terminals and crossover which replaces whole trees. Where variety is low both these types of crossover lead to a further production of clones. If crossover produces copies of parents at a high rate this leads to a reduction of the variety of the population and the eventual extinctions of certain primitives. The cloning of parents is found to prevail when the crossover operator is applied to trees which are short or identical to their parents. Furthermore, if the trees involved in crossover have repeated subtrees than the chance of producing clones is greater.

Proposed solutions to the cloning problem include:

- Do not use the reproduction operator.
- An increase in selective pressure.
- Detect when an offspring is identical to one of its parents. This information can be used to reduce the GP run time or increase the variety. The first of these is achieved by copying the fitness value of parents for offspring that are clones instead of re-evaluating the offspring. The latter is achieved by not allowing the production of duplicates.

2. Introns and Bloat

In the previous section the concept of explicitly defined introns was described. In addition to these artificially created introns, an individual can also contain introns created naturally as part of the evolutionary process.

Although these introns help to reduce the destructive effects of crossover during the early and middle stages of a GP run, they grow exponentially towards the end of a run. This exponential growth of introns is called bloat and causes the GP algorithm to stagnate.

Introns are pieces of code that do nothing and have no effect on the fitness of an individual. The following are examples of introns:

(NOT(NOT(X)), (IF (2==1)...X), (+ X 0).

Two types of non-functional code commonly found are:

- Code which does nothing.
- Code which is not executed, e.g. an alternative in an if-statement.

The emergence of introns has been described as a protective response to the destructive effect of the crossover operator. An increase in the destructive effects of the genetic operators results in an increase in the growth of introns.

Three types of function introns exist:

- Local introns - These are functions that have no effect except to pass on the values that were passed to them.
- Hierarchical introns - These are functions which contain one or more arguments that will be ignored, e.g. an if-else statement.
- Sibling (horizontal) introns - These functions undo the effect of a function represented by a sibling tree. e.g. when a subtree has its contributions to a memory location overwritten by another subtree after it.

Research in this area has lead to the discovery of a number of ways of dealing with this problem of bloat:

- The use of the editing operation during a run.
- An increased application of the mutation operator has been found as a solution to the problem of intron growth. Systems applying the mutation operator stagnated a lot later, the number of introns were reduced and better overall solutions were found.
- Due to introns and bloat the solutions produced by a GP system often contains redundant code. In order to reduce the amount of redundant code in programs generated a measure of parsimony can be incorporated into the fitness function by penalizing an individual for the number of nodes that make up an individual. Koza states that parsimony can be incorporated into the fitness function. However, the problem with this is deciding what weighting to give the different objectives represented by the fitness function. He is of the opinion that the removal of extraneous code too early in a run can result in the reduction of the diversity of the population. Thus, Koza suggests incorporating parsimony into a fitness function late in a run or alternatively after a certain number of solutions have been found.

  The application of the editing operator to remove non-functional code will be able to reduce the amount of non-functional code but will not be able to remove it completely. Soule has found that the use of a penalty function is the most effective means of removing bloat.

Koza suggests that the best-of-run program is simplified during a post-run process.

- Koza describes the use of automatically-defined functions as a means of producing more parsimonious programs.

Sources of Information

- Genetic Programming FAQ
  http://www.cs.ucl.ac.uk/research/genprog/gp2faq/gp2faq2.html(FAQ)
- What is genetic Programming ? -
  http://www.genetic-programming.com/gpanimatedtutorial.html
- The Genetic Programming Tutorial Notebook
  http://cair.kaist.ac.kr/gp/Tutorial/tutorial.html
- Genetic Programming
  http://vivaldi.ece.ucsb.edu/projects/GP/aboutgp.html
- Genetic Programming Homepage
  http://www.genetic-programming.org/

References

1. Andre D.,Teller A., A Study in the Response and the Negative Effects of Introns in Genetic Programming, in the proceedings of the First Conference on Genetic Programming, editor J.R. Koza, AAAI Press, pg. 12 - 20, 1996.
2. Banzhaf W., Nordin P., Keller R.E., Francone F.D., Genetic Programming - An Introduction - On the Automatic Evolution of Computer Programs and its Applications, Morgan Kaufmann Publishers, Inc., 1998.
3. Bruce W. S., The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs, Phd Dissertation, School of Computer and Information Sciences, Nova Southeastern University, 1995.
4. Hooper D.C., Flann N. S. , Improving the Accuracy and Robustness of Genetic Programming through Expression Simplification, nin Genetic Programming, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, MIT Press, 1996, pg. 428.
5. Keller R.E., Banzhaf W., Explicit Maintenance of Genetic Diversity in Genospaces, http://1s11-www.cs.uni-dormund.de/people/keller/publication.html, 1994.
6. Koza J. R., Genetic Programming I : On the Programming of Computers by Means of Natural Selection - John R. Koza, MIT Press,1992.
7. Koza J.R., Genetic Programming II, Automatic Discovery of Reusable Programs, MIT Press, 1994.
8. Koza J. R., Bennett III F. H., Andre D., Keane M. A., Genetic Programming III Darwinian Invention and Problem Solving, Morgan Kaufmann Publishers, 1999.
9. Langdon W.B., Genetic Programming and Data Structures, Genetic Programming + Data Structures = Automatic Programming!, Kluwer Academic Publishers, 1998.
10. Mawhinney D., Preventing Early Convergence in Genetic Programming By Replacing Similar Programs, http://citeseer.nj.nec.com.mawhinney00preventing.html, 2000.
11. Pillay N., Using Genetic Programming for the Induction of Novice Procedural Programming Solution Algorithms, in ACM Proceedings of the 2002 Symposium on Applied Computing (SAC2002), pp. 578-584, ACM Press, March 2002.
12. Soule T.,Foster J.A.,Dickinson J., Code Growth in Genetic Programming, in Genetic Programming 1996, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, MIT Press, 1996, pg. 215-223.
13. Teller A. , Learning Mental Models, in the proceedings of the 1993 International Simulation Technologies Conference, OmniPress, 1993.
14. Teller A., Genetic Programming, Indexed Memory, The Halting Problem and Other Curiosities, in proceedings of the 7th Annual Florida AI Research Symposium, IEEE,pg. 270 - 274, 1994.
15. Teller A., Veloso M., PADO: A New Learning Architecture for Object Recognition, in Symbolic Visual Learning 1996, Oxford University Press, pg. 81 - 116, 1996.