Evolution of the Discrete Cosine Transform Using Genetic Programming

A thesis presented in partial fulfilment
of the requirements for the degree of

Master
in
computer science

At Massey University, Albany,
New Zealand

Xiang Biao Cui

2001

Acknowledgements

I would like to give my most sincere thanks to my supervisor, Dr. Martin Johnson for his supervision through out the course of the project.

I would also like to give my thanks to my wife Bihong for her sincere support.

Abstract

Image compression is a important method in image transmission, storage and manipulation. There are many successful techniques which have been developed. Most of these methods are based on some type of rule based algorithm. The cosine transform plays a very important role in image compression. It is a standard transform used by the widely used JPEG standard. Through the use of genetic programming, we successfully evolve a programmatic cosine transform based on genetic programming. The cosine transform has been heavily researched and many efficient methods have been determined and successfully applied in practice. Here, we only suggest 'another' method to do the same work. Due to the limited power of our resources, we restricted our work to a 4 point cosine transform. As a result, an approximation to the transform is evolved by the genetic programming paradigm. In theory, the 8 point cosine transform can be evolved using a similar technique.

1. Introduction to DCT (Discrete Cosine Transform)

This paper is concerned with an algorithm for a 4-point 1-D DCT based on genetic programming. We will discuss a mathematical basis of the DCT and genetic programming in general. Then we will show our results for a 4-point DCT evolved using the genetic programming paradigm.

1.1 General Overview On DCT

The cosine transform translates a set of points of data from the spatial domain to the frequency domain. The DCT has found a wide range of application in signal processing, data compression, telecommunication, image processing, feature extraction and filtering. The DCT is a very important translation method in multimedia application. This is because the DCT performs much like the Karhunen-Loéve transform for first-order Markov stationary random data. DCT algorithms can be classified into three categories based on their approach. a) indirect computation, b) direct factorization, c) recursive computation. [17]. A two-dimensional DCT can be obtained by first applying a 1-D DCT over the rows followed by a 1-D DCT to the columns of the input data matrix. Many international standards such as JPEG, MPEG and HDTV take DCT as a translation standard in coding. A great detailed research into DCT is introduced in [40].

1.2 Expression Of Forward DCT

The expression of a 2-D DCT is based on the following:

$$T(i,j) = c(i,j)\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} S(m,n)\cos\frac{\pi(2m+1)i}{2N}\cos\frac{\pi(2n+1)j}{2N}$$

In the above expression:

T(i, j) is the output matrix

S(m, n) is the input matrix

c(i, 0) = c(0, j) = 1/N

c(i, j) = 2/N (i ≠ 0, j ≠0)

N is the translation point size.

The 1-D forward DCT is expressed as:

$$T(k) = c(k)\sum_{n=0}^{N-1} S(n)\cos\frac{\pi(2n+1)k}{2N}$$

Here: c(0)=$\sqrt{1/N}$, c(k) =$\sqrt{2/N}$ (k≠0)

1.3 The Properties Of DCT

One reason that DCT is used so wide today is because of its properties:

1. The DCT compaction efficiency is close to Karhunen – Loéve transform.
2. The orthogonal property. If the forward transform is expressed as:

$$Y = TXT^t$$

The inverse DCT can be expressed as:

$$X = T^tYT$$

For the above DCT expression, the IDCT can be expressed as:

$$S(m,n) = c(i,j)\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} T(i,j)\cos\frac{\pi(2m+1)i}{2N}\cos\frac{\pi(2n+1)j}{2N}$$

3. The separable property: A 2 – D DCT can be computed by two 1 – D DCTs. This property makes the DCT very efficient. In terms of 2 – D DCT, for an 8×8 block, computing one coefficient requires 64 multiplications and 64 additions. To compute a block requires 4096 calculations. If we separate the 8×8 square into columns and rows and compute the coefficient by row and column, we need 64 multiplications and 64 additions for 1 row (or column). 8 rows and 8 columns need 1024 multiplications in total – and 1024 additions. The calculation can be reduced significantly this way. [5]
4. DCT is image independent.

1.4 DCT In JPEG Standard

JPEG takes an 8×8 pixel block as its DCT size. 8×8 pixel blocks have a suitable block size and good compression efficiency. 64×64 pixel block may be large for a normal size image and does not provide better compression. 4×4 pixel blocks reduce the compression efficiency in real applications.

1.5 Complexity Comparison Between Algorithms

Many different fast algorithms to compute the DCT have been proposed. The complexity of some well known algorithms for an 8 point 1 – DCT are shown below:

Method	Multiplications	Additions
W. Chen[7]	16	26
B. G. Lee[29]	12	29
H. S. Hou [17]	12	29
C. Loeffler [30]	11	29

2. Introduction To GA (Genetic Algorithms)

Genetic algorithms are a class of function optimization algorithms which search a space of problem solutions through simulation of the principles of biological genetics and natural selection (Goldberg. 1989). A genetic algorithm transforms a population of individual objects, each with an associated fitness value, into a new generation of the population. The Darwinian principle of reproduction and survival of the fittest and other genetic operators will guide evolution of the GA. Each individual in the population represents a possible solution to the particular problem. The fitness may be thought of as a force which causes hill climbing to reach the highest peak in the search space. The necessary topics in genetic algorithms are described as the following.

2.1 Representation Scheme

A problem's solution space should be represented in a scheme. The scheme constrains the problem domain in a GA's language. Any possible solution should be communicated with this language. In terms of conventional GA, individuals in the population are usually fixed length character strings. The representation scheme starts with selection of the string L and the alphabet size K. K is usually equal to 2. The most important part is the design of the length L. This requires mapping a possible point in the search space of the problem into a chromosome. Also, each chromosome is a point in the search space of the problem. The step of determining the representation is crucial in applying a genetic algorithm to solve problems. If the string is not long enough to represent the solution of the problem, evolution may not converge. If L is too long, more searching effort is wasted.

2.2 Fitness Measure

In nature, individuals that are better adapted to their environment are more likely to survive and their chance to reproduce is higher than those with poor abilities. A fit individual should gain higher reward. Each individual in the population represents a possible solution to the problem. Fitness measurement should reflect the level of strength for each individual. Fitness is always evaluated in the problem domain. Fitness must be able to represent not only a complete solution to the particular problem but a partial solution to the problem. A fitness function is proposed to interpret the meaning of the genotype in the context of the problem environment. The result of the fitness should represent how good the possible solution is.

There are two distinct categories in AI problem solving methods: weak and strong. A strong method is one that contains a significant amount of task specific knowledge in order to solve it. A weak method requires little or no knowledge about a task. Genetic algorithm is a weak method. A proper fitness measurement is crucial for communication between the problem solving domain and the genotype representation domain.

2.3 Genetic Operators

Genetic operators can be applied to individuals in the population to generate new or useful individuals in the population of the next generation. Reproduction, mutation and crossover are three commonly used genetic operators. Reproduction is based on the Darwinian principle of reproduction and survival of the fittest. Mutation allows new individuals to be created. Crossover creates new individuals that contain useful genotype from their parents.

2.3.1 Reproduction

In the reproduction operation, an individual is selected from the population based on its fitness and then the individual is copied into the population of next generation. The selection is done in such a way that the individuals with better fitness have higher probabilities to be selected. In this way, the individuals with poor fitness have a lower probability of being selected.

2.3.2 Crossover

Crossover operates between two parents. Parents are selected in such a way that individuals in the population that have higher fitness have higher probabilities to be selected. Individuals that have lower fitness also have a chance to be selected but the chance is lower then that of a fit one. Crossover produces two offspring. Each offspring contains some genetic material from each of its parents.

In practice, single point crossover is most likely to be used. We demonstrate the operation here. Two parents have the strings as:

Parent 1	Parent 2
11001011	01001110

These parents have string length L=8, K=2. At the first step of crossover, a crossover point has to be selected. The crossover point must be selected from bit 1 to L-1(first bit is bit 0) so that the string is divided into a head and a tail. Assuming the crossover point is 5, the fragments are:

Head 1	Head 2
11001	01001

Tail 1	Tail 2
011	110

The crossover operation combines head 1 with tail 2 and head 2 with tail 1 to generate two new offspring.

Offspring 1	Offspring 2
11001110	01001011

After crossover operation, we observe that the two head strings remain on the left side of the new offspring. The tails remain on the right side of the new offspring. Tail 2 goes to offspring 1, and tail 1 goes to offspring 2.

2.3.3 Bit Mutation

The bit mutation operation is applied to a single individual within the population. The individual is again selected based on the fitness. Individuals with higher fitness have higher chance of being selected and individuals with lower fitness have lower chance of being selected. Mutation occurs on one or more bits. The mutation operation toggles a bit in the string. The figure below illustrates this.

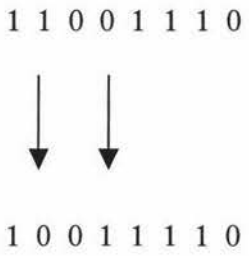

Figure 2.3.1. Mutation operation changes bit value.

The mutation operation extends the search space with new points.

2.4 Parameter Selection

Parameter setting is important because it determines the run time behaviour of the algorithm. The primary parameters for controlling a genetic algorithm are the population size and the maximum number of generations to be run. Secondary parameters, such as reproduction probability, crossover probability and mutation probability are also important. In addition, some parameters such as the selection method have to be determined.

2.5 Structure Of Genetic Algorithm

The figure below illustrates the flow chart of a conventional genetic algorithm.

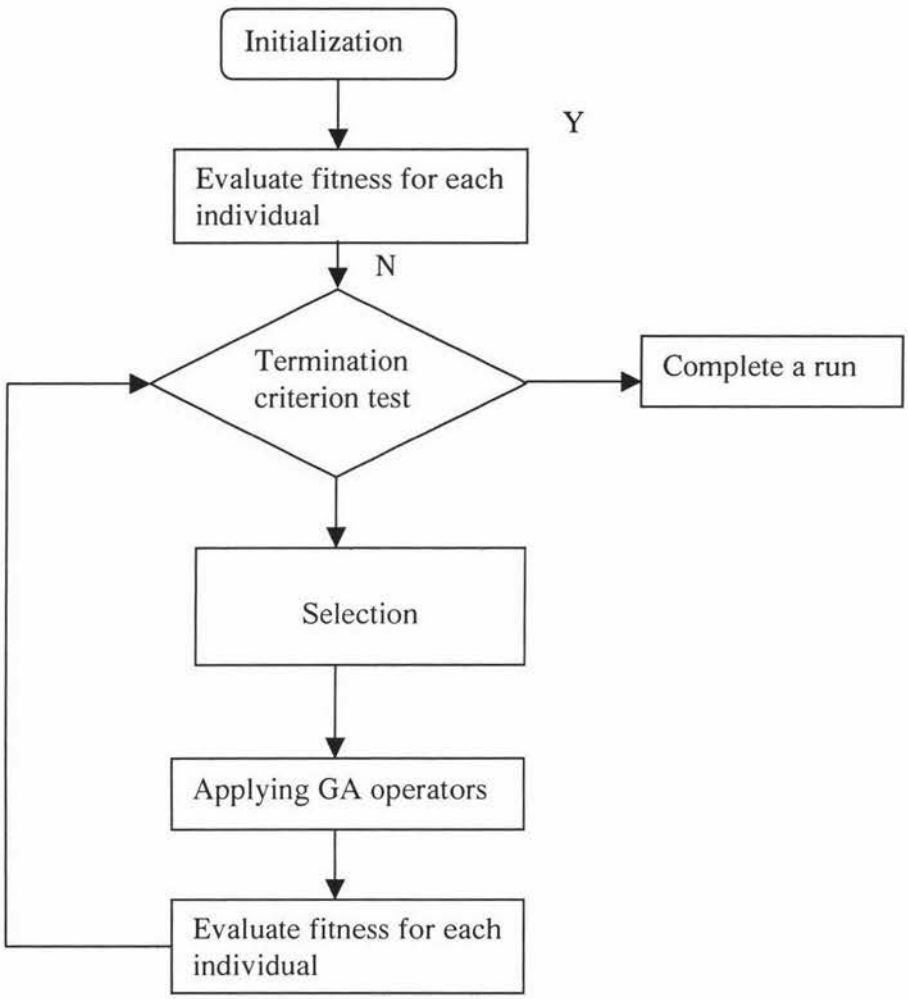

Figure 2.5.1. The flow chart of conventional GA.

The first step in performing a GA is to randomly produce an initial generation. This is called initialisation. The number of individuals within this generation is determined by the value of the population size. After initialisation, all individuals in the population have to be evaluated so that the fitness value of each individual has been calculated. If the best fitness

value satisfies the requirement of the system, the GA procession terminates. Another termination criteria may be the maximum number of generations. Once this value is reached the system is terminated.

The next step is selection. This determines the objects that genetic operators will apply to. Reproduction and mutation operate on a single individual and crossover operates on two parents. After applying genetic operators to individuals, new individuals will be generated and they will be put in the next generation. Fitness evaluation should be done after any step evolving genetic operators.

The best solution and desired solutions should be recorded at the termination stage.

2.6 An Example

Now let us consider an example of applying genetic algorithms to solve a problem. The problem is to find the maximum value of the sine curve. The formula we illustrate here is:

$$f(x) = \sin(x\pi/8) + 1$$

The x value varies between 0 and 15. Step length is set to 1. This is an optimisation problem.

To solve this problem we have to provide a concrete implementation of the steps stated above. The most commonly used representation language for the basic genetic algorithm is a fixed length binary string. We will also follow this scheme in this example. Since our step length is 1, and the total number of sample points is 16, it is possible to represent the solution as a 4 bit string. It should be stated here that the scheme representation is crucial,

the possible solution must be included in the scheme. If the precision required is higher, the number of bits in the string must be increased to satisfy this requirement. In other problem domains, different lengths of string and different coding alphabets may be adopted. It is sometimes difficult to determine the representation scheme.

To design the fitness function, we first have to describe the error between the target value and the calculation value. Since the function has a value between 0 and 2, we can calculate the error as:

$$error = 2 - (\sin(x\pi/8) + 1)$$

This makes the error value always positive.

It is quite straightforward to design the fitness function now:

$$fitness = (2 - error)/2$$

The fitness value will be 1 when error is 0. If we find the maximum value of the curve, we will have the fitness value 1.

Population size is set to 5. This is sufficient for such a trivial example. Maximum generation size is 10. If the ideal result is not found within 10 generations, we terminate the program and run it again.

We assume that we have no a-priori knowledge about this problem. Before starting the generation loop, we initialise the individuals in the population randomly. In one run, we have the following records.

Generation 1:		Generation 2:	
String	Fitness	String	Fitness
1101	0.09	0000	0.17
1101	0.09	1010	0.10
1101	0.09	1101	0.09
0000	0.17	1101	0.09
1100	0.09	1010	0.10
Total Fitness: 0.53		Total Fitness: 0.55	

Generation 3:		Generation: 4	
String	Fitness	String	Fitness
1000	0.17	0100	1.00
0101	0.72	1011	0.09
1010	0.10	0101	0.72
1010	0.10	0101	0.72
0000	0.17	1010	0.10
Total fitness: 1.26		Total Fitness: 2.63	

The total fitness value in the first generation is 0.53. No genetic operator is applied to the individuals in this generation.

After initialisation, genetic operators will be applied to the individuals. The genetic operators used in this example are reproduction, crossover and mutation. Natural selection

is simulated in selecting the genetic operator object. The individuals with higher fitness values have high chances of being selected. In practice, a very common approach called roulette wheel selection is used to perform the selection. This approach has the advantage of being simple to implement. An individual's probability for selection is given by:

$$p_k\,(selection\ ) = \frac{fitness_{\ k}}{\sum_{i=1}^{N} fitness_{\ i}}$$

Here, N is the population size. The higher the fitness value of a particular individual, the higher is its probability of being selected for the next generation. This can be illustrated by using a roulette wheel, each fitness value occupies a part of the wheel. The bigger the fitness value, the larger part it occupies. A random point is selected from the wheel. A larger part has the higher probability to be selected. In implementation, we calculate the total fitness. A random value is selected between 0 and the total fitness.

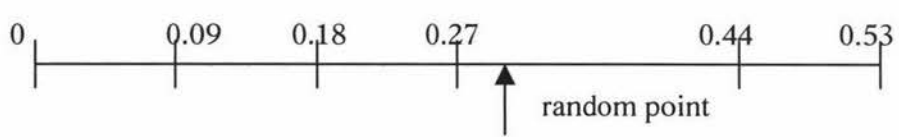

Figure 2.1 Fitness distribution for generation 1

For generation 1 in our example, we add up the fitness from individual 1 to individual 5. The partial sum is marked on the line. After a random point is selected, we observe that the left side of the random point is just the sum of the fitness from individual 1 to the random point. If we keep adding the fitness starting at individual 1, if the sum is greater than or equal to the random value, we select that individual. Note that crossover requires two parents, these are selected separately. These two parents may be the same individual.

Reproduction is performed by selecting an individual and copying it into the new population. Note that the new individual may not sit at the same position as the individual in the old generation. Fitness value does not need to be re-evaluated. A copy will reduce the calculation effort. Crossover is performed between two parents. A cross point is selected randomly between the second bit to the last bit. Mutation is performed by randomly toggling one bit in the selected individual. It produces one child, which is placed in the new population. The choice of which genetic operator to use is controlled by a probability. A weight between the three operators can be distributed.

There are two termination criterions in this example. One is the maximum generation and another one is the best fitness reached. The running result shows four generations. We can see that the total fitness increase after each generation.

3. A Short Introduction To Genetic Programming

Genetic programming as a field was introduced by John Koza[25] in 1992. Genetic programming is an attempt to answer the question: "can computers solve problems without being explicitly programmed?" That is, how can computers be made to do what needs to be done without programming. GP is based upon Holland's genetic algorithm [16]. The difference between GP and Holland's GA is that GP gives the solution to the problem in abstract syntax trees instead of a fixed – length bit string. The solution to the problem is evolved using a set of examples and the training process is a calculation of the distance between the solution and the desired target.

GP runs with an initial process. In this process, a population is created. Each individual represents an abstract syntax tree which is made up with a series of functions and terminals. The designer has to give the proper function set and terminal set so that the problem can be solved. A fitness value is set up to determine how good an individual is at solving the problem. The fitness value is designed to simulate the ability of surviving. In computer programming, an abstract syntax tree should be a computable expression. Since it is possible to calculate an abstract syntax tree for each set of input data, the fitness function can be designed to determine the fitness value based on the result of the abstract syntax tree. Training data is usually a set of input output data pairs. The fitness value should reflect the strong level of the tree toward all of the training set. In practice, a distance is used to represent how fit the tree is by comparing the output data of the training set and the result of the abstract syntax tree.

After initializing the population, the GP mechanism will take over and search the problem space. A tree can be created from the function set and terminal set while the fitness function determines how close the tree is to the problem specification. During running of the GP, new individuals are produced continually and a fitness value for an individual is attached to itself. The commonly used GP operators are reproduction, crossover and mutation.

Reproduction is a method that copies an individual in the population and places it into the new population. Reproduction copies the abstract syntax tree as well as all the information attached to the tree, including the fitness value. The fitness value is used to determine the chance of reproduction. Reproduction results in more useful abstract syntax trees appearing in the population.

The crossover operation is a sexual operator: it takes two parental programs selected based on fitness. The parental programs in genetic programming are typically of different sizes and shapes. The offspring programs are composed of sub-expressions from their parents. Often a GP program has a maximum depth to limit the computing time and ensure any program is bounded by the maximum program size. If the newly generated program size is longer than the maximum size, it is necessary to select another node from the parental programs or to reproduce the parent program. Crossover provides a mechanism to explore the search space by creating a new program that combines advantages of parents with high fitness value.

Mutation operates on one program, from this one or more nodes is selected to be modified. The selected node is replaced by a randomly generated new node. The newly generated individual should be bound to the maximum program length. After mutation, the fitness of the program is evaluated and is placed in to the population. The mutation operator helps to avoid local optima in the search space.

During the run, GP operators operate with a given probability. Individuals or parents are selected based on Darwinian evolution. Obviously, population members with high fitness values should have a high chance of survival and have a high chance of helping form the next generation than those with a low fitness value. In genetic programming, parents are selected through a use of a stochastic selection scheme. Individuals with a higher fitness value have a higher chance of being selected as a parent than those with low fitness values. The fitness value will drive the population moving from one generation to next, and like natural selection, only the strong individuals can survive. The average fitness of the next generation will be higher than the previous generation. In such a way, the population can evolve over time.

Each individual in the population during the search process represents a point in the search space. The entire population makes up the search space. Using the fitness, the exploring process moves toward the highest point which implies the desired solution to the problem. When the generation reaches the highest point in the search space, a termination criterion designed to stop the search is satisfied and the genetic programming system is completed.

There are five major steps in applying genetic programming to solve a problem.

1. determine the set of terminals.
2. determine the set of functions.
3. determine the fitness function.
4. determine the parameters and variables for controlling the run.
5. determine the method of designating a result and the criterion for termination of a run.

Step one determines the terminal set which may be viewed as an input to a computer program. Step two determines the function set which is a mathematical operation or some operational functions.

The terminal set and function set make up the abstract syntax tree which makes exploring the problem possible. The last three steps correspond to the steps for a conventional genetic algorithm.

Genetic Algorithms are stochastic search techniques introduced by Holland[16] which takes the Darwinian idea of natural selection, or "survival of the fittest", and applies it to digital computation. The essence of the algorithm is that a population of individuals is evolved over generations. The individuals that are more fit have more survival chance than others.

A Genetic Algorithm starts with an initial 'population of trial solutions to a problem'. The next problem is to determine the sort of representation that GA should manipulate. The traditional simple GA, as defined by Holland represents the problem's data as chromosomes, each of which is a collection of genes. Conventional GA maps the genetic

units into binary strings, where each bit corresponds to a single gene. A group of such binary strings is the population, with each individual of the population referring to a single possible solution to the particular problem. The population of the first generation is generated randomly and the genetic operators will be applied to the individuals in the population. As in nature, after certain generations, the average fitness of the population will be stronger.

The interaction of the genetic operators and the stochastic selection based on survival of the fittest individual results in an exploration of the search space. Natural selection and the reproduction operator provide the hill climbing pressure to improve the fitness of the overall population. Reproduction provides the means of search space exploration by cloning abstract syntax trees with high fitness. Mutation maintains the diversity required to avoid local optima in the search space. Since fitness of individual abstract syntax trees is based upon the input output specification of the desired program, the result of the search is an abstract syntax tree which will satisfy the desired behavior or closely approximates it.

3.1 The Standard GP

Koza(1989, 1992) describes a genetic programming system using Lisp S-expressions. In his foundation work, he uses a tree–based representation as the genetic algorithm framework. Some basic genetic operators for genetic programming such as mutation and crossover were defined. Consider the problem with two variables a and b, output variable y is represented in the form:

$$y = (a - b) / 3$$

This expression may be represented as a tree structure in figure 3.1.

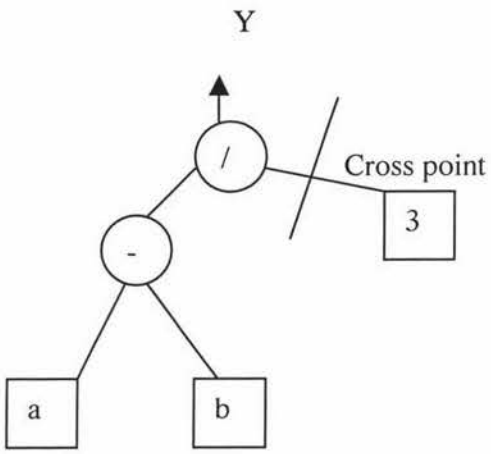

Figure 3.1: tree 1

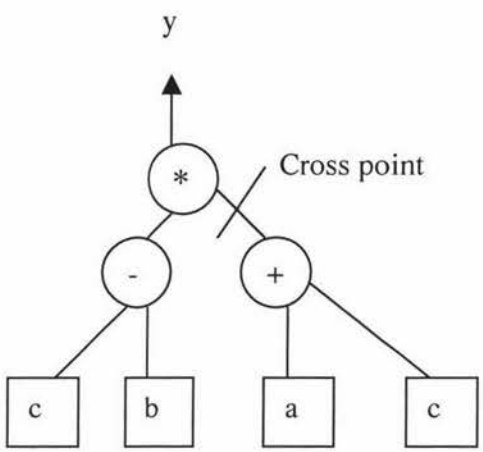

Figure 3.2: tree 2

If we have another tree structure, in figure 3.2, tree2, crossover may be applied to these trees. The cross points are as shown on the figures above. Two cross fragments are shown as follow.

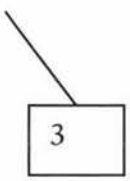

Figure 3.3 Tree Segment 1

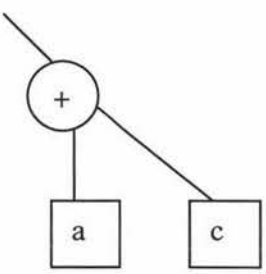

Figure 3.4: Tree Segment 2

The two cross fragments exchanged position after the crossover operator is applied. The resulting offspring are as follows:

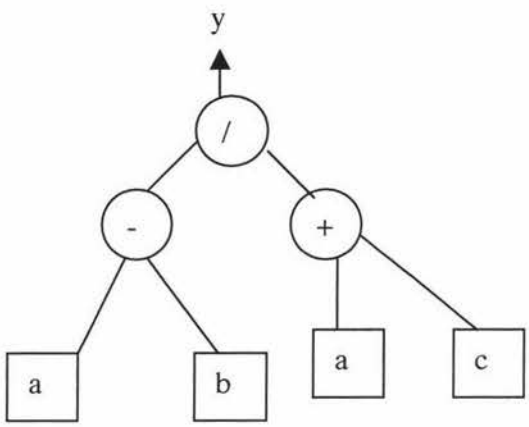

Figure 3.5: Tree 1 after crossover

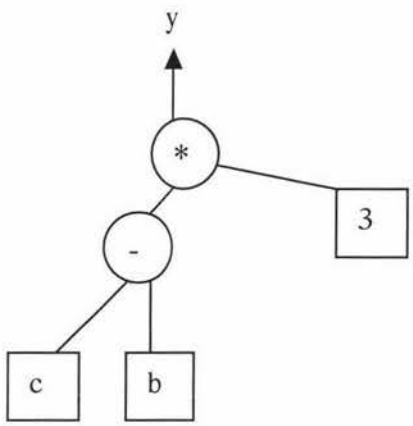

Figure 3.6: Tree 2 after crossover

In Koza's Lisp based GP, some constraints are applied to the tree structure. Two parameters, the depth and number of nodes are often defined. The mutation operation begins by selecting a mutation point. The mutation point is selected as a function or terminal. The mutation operation removes the selected point or below the selected point and then insets a randomly generated subtree at the point[25].

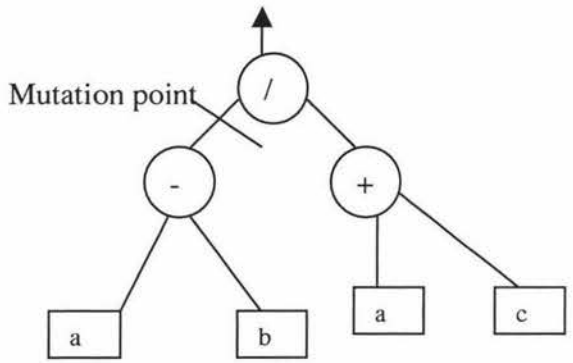

Figure 3.7: before mutation

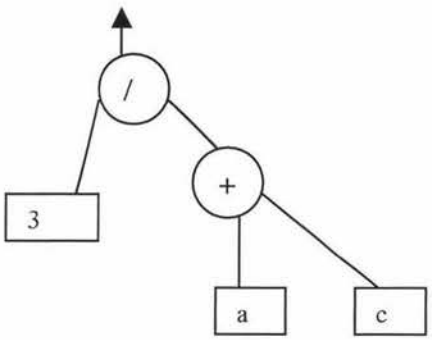

Figure 3.8: after mutation

One of the properties of genetic programming is that the tree may be generated randomly. A function may call other functions or be called by other functions. The closure property must be satisfied. Kosa defines the closure property as each of the functions in the function set be able to accept, as its arguments, any value and data type that may possibly be returned by any function or be taken by any terminal [25].

For example, a Boolean operation may only take Boolean values such as True or False but not number values. Therefore it must be ensured that every function within the tree structure is provided with suitable arguments. This must be the case for every possible combination of functions and terminals that the system might encounter during initialisation of a tree. The closure condition must also be satisfied when applying genetic operations such as mutation and crossover.

Providing a function argument with the wrong type is not the only example of violating the closure condition. Division by zero is another example. To overcome the problem of

division by zero, Koza [25] defines a function known as protected division. When a function, which performs protected division, encounters an argument of zero for the divisor, it produces a string *':undefined'* for its output. All other functions within the function set have to be changed so that they can deal with this result. Other examples of violated closure condition are square root and logarithm of a negative number.

3.2 Differences Between Genetic Algorithm And Genetic Programming

Angeline [2] compares and contrasts genetic programming with genetic algorithms. In this analysis he discusses three differences between genetic algorithm and genetic programming: (a) the representation of genotypes, (b) the complexity of interpretation, and (c) the use of syntax preserving crossover.

Genetic algorithms generally use a genotype, which is a fixed-length string of characters from a low cardinality alphabet. Genetic programming uses a variable sized tree representation for its genotype. It is claimed that the use of this representation makes genetic programming more powerful than genetic algorithms. Angeline concludes that in practice the representations are equivalent. In most genetic programming systems there is a maximum size to the trees that can be generated. In addition the functions used in genetic programming have fixed known branching factors. Under these constraints, the trees used in genetic programming can easily be represented as variable length strings with the union of the terminal and non-terminal symbol sets used as the alphabet. Given this equivalence of representations, genetic programming can be viewed as a genetic algorithm that operates on variable length strings built using a high cardinality alphabet.

The second area where it is claimed that genetic algorithms differ from genetic programming is in the interpretation of the representation as a genotype. The strings used in genetic algorithms are generally interpreted using a positional encoding where the meaning of an element is based on both its symbol and its position in the string. Conversely, in genetic programming the encoding has no positional component in its interpretation. A function symbol has the same meaning no matter where it may appear in the tree. This looks to be a difference, but it is only a difference in standard practice. There is no theoretical reason why a genetic algorithm could not interpret the characters in its string-based representation without regard to their position.

The final area in which there is a potential difference between genetic algorithms and genetic programming is in the operators used in the two types of systems. In genetic algorithms the crossover and mutation operators are fully general and have no component which considers the interpretation of the strings. In contrast, the crossover and mutation operators in genetic programming are carefully defined to produce syntactically correct trees after the completion of the operation. The operators know that they are manipulating trees and build their output so that it can always be manipulated by the interpretation function. Angeline concludes that the use of syntax preserving operations in genetic programming is the only real theoretical difference between genetic algorithms and genetic programming.

4. Related Work

In Koza [25](10.13), image compression is mentioned and he successfully evolves a model to compress a 30X30 bitmap image with a good solution in standard GP. Because of the vast computation necessary, only a small image was tested in Koza's work.

In Koza and other standard GP systems, an S-expression is evolved and evaluated. Most of the time is spent on evaluating each S-expression. To increase the speed of evaluating each S-expression, Fukunaga et al [12] developed a compiler for an efficient GP system. The Gnome compiler translates each S-expression into SPARC machine language codes before evaluating it. Simple arithmetic operators can be executed with a single instruction at the hardware level instead of many recursive function calls in standard LISP S-expression. In one paper [12], a predictive coder is evolved that can be used for multi-level compression(e.g. a Huffman coder). They worked on a 64X64 image and this proved much faster than the standard GP system.

Predictive coding is an image compression technique, which uses a compact model of an image to predict pixel values of an image based on the values of neighbouring pixels. A model of an image is a function, such as *PredictedModel(neighbourValue)*, which computes the pixel value at coordinate (x, y) of an image. The neighbour value is known to the predicted pixel. Linear predicted coding is a simple case of predictive coding in which the model simply takes a weighted average of the neighbouring value. Nonlinear predicted coding assigns arbitrarily complex functions to the model. The algorithm of predicted coding is described below:

Encoder (Model, Image)

```
for x=0 to xmax
     for y= 0 to ymax
          Error [x, y] = Image[x, y] -
                         PredictedModel(neighbourValue)
```

Decoder(Model)

```
for x = 0 to xmax
     for y = 0 to ymax
          Image[x, y] = PredictedModel(neighbourValue)
                         + Error[x, y]
```

Applying a model to an image results in an error between the original image pixel value and the predicted value. The error can be compressed using standard compression methods such as Huffman coding. Applying the model to the error can recover the original image.

Nordin and Bnazhaf [34] use the idea of programmatic compression to compress sound and image data. They use a chunking method to process an image that makes the system easier to converge to an expected solution. The picture was divided into 8X8 or 16X16 pixel blocks. Chunking involves two systems. The first treats all the fitness cases at the same time. The second applies programmatic compression to equally sized sub-sets of the fitness cases and evolves a solution to each of them. Programmatic compression works on a Compiling GP system(CGPS) that ensures a high speed evaluation of each fitness case. They also claimed that it is the first time to compress a real full size image(256X256) successfully with GP.

The basic idea of programmatic compression is that any system, which evolves programs or algorithms for generating data, can be viewed as a data compression system. The data that should be compressed are presented to the genetic programming system as fitness cases for symbolic regression. After choosing a function set that facilitates an accurate reproduction of the uncompressed data, the system then tries to evolve an individual program that, to a certain degree of precision, outputs the uncompressed data. If the evolved program solution can be expressed by fewer bits than the target data, then a compression is achieved.

5. Implementation Consideration

This chapter contains a discussion of many different implementation techniques that may be applied to a genetic programming system.

5.1 Representation

The parse tree representation was first used in Cramer(1985). In this representation, functions and variables are expressed as numbers instead of explicit symbols and stored in a fixed – length string. The numbers can be matched to functions according to a lookup table. For instance, an encoded program could have the form (4 (1 3) (2 1 (2 2 (0 3))))), where the parentheses denote a complete statement. The lookup table could have the following form.

Number	Function	Meaning
0	: INC VAR	Increment [VAR]
1	:ZERO VAR	Set [VAR] =0
2	:LOOP VAR FUNC	Execute [FUNC] [VAR] times
3	:SET VAR1 VAR2	[VAR1]=[VAR2]
4	:BLOCK FUNC1 FUNC2	Execute [FUNC1] then [FUNC2]

The program above can be represented as:

(:BLOCK (:ZERO V3) (:LOOP V1 (:LOOP V2 (:INC V3))))

Note that in Cramer's tree – based language structure, a subtree argument does not return a value to the calling statement but only designates a command to be executed.

In Koza's parse tree representation, the subtree arguments in genetic programming return values to their calling statement. A function can call other functions recursively and can be called by other functions in the same way. A typical parse tree is showed in figure 5.1.1. In such a representation, a maximum length restriction is imposed on the evolved program. Without such a restriction, the dynamically generated program may increase in size uncontrolled, eventually swamping the computer's available resource. There are two common restrictions. One is the depth limitation, which restricts the size of evolving parse trees based on a user–defined maximal depth parameter. Another one is node limitation, which limits the total number of nodes available for an individual parse tree. In practice, node limitation is more used. This is because of its easy implementation and the fact that it imposes fewer limitations on the structural organisation of the evolving program.

In a parse tree representation, the primitive language definition is crucial for the solution to the particular problem. The solution must be bounded in the definition of the primitive language. Koza addresses this as the *sufficiency* property. Sometimes the representation primitive language may be taken from the programming language, but typically it is tailored by carefully consideration of domain–specific knowledge. For instance, in Koza's artificial ant problem[25], the primitive language includes elements such as move, right, left etc, which are not elements of the programming language.

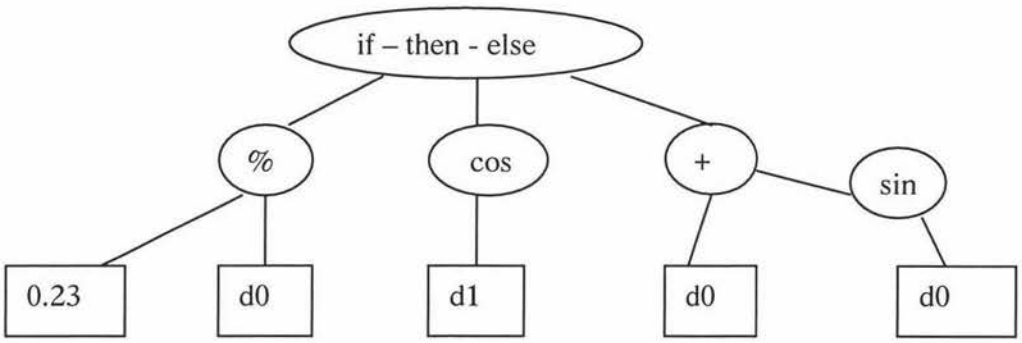

Figure 5.1.1: a typical parse tree

It is sometimes necessary to insert a constant into an evolved program. The most common way of doing this is to use a symbol to represent a constant. The constant is then usually an element of the terminal set. When evaluating an individual that contains a symbol, rather than evaluate the symbol directly, the symbol is replaced by a relative constant. For instance, in the terminal set T={ CON1, d0, d1}, d0 and d1 represent two input parameters and CON1 represents a constant. When CON1 is evaluated in the evolved parse tree, a relative constant (say 2.34) is inserted and evaluated.

5.1.1 Automatically Defined Functions

In standard genetic programming, the function set generally contains primitive functions that can be defined before the program runs. For a simple problem, this may be sufficient to solving the problem. However, in some cases, it may be difficult for a solution to be evolved. A solution to this problem arranges the problem as a hierarchical structure using

automatically defined functions. ADFs are evolvable functions (subroutines) within an evolving genetic program, which the main routine of the program can call [26].

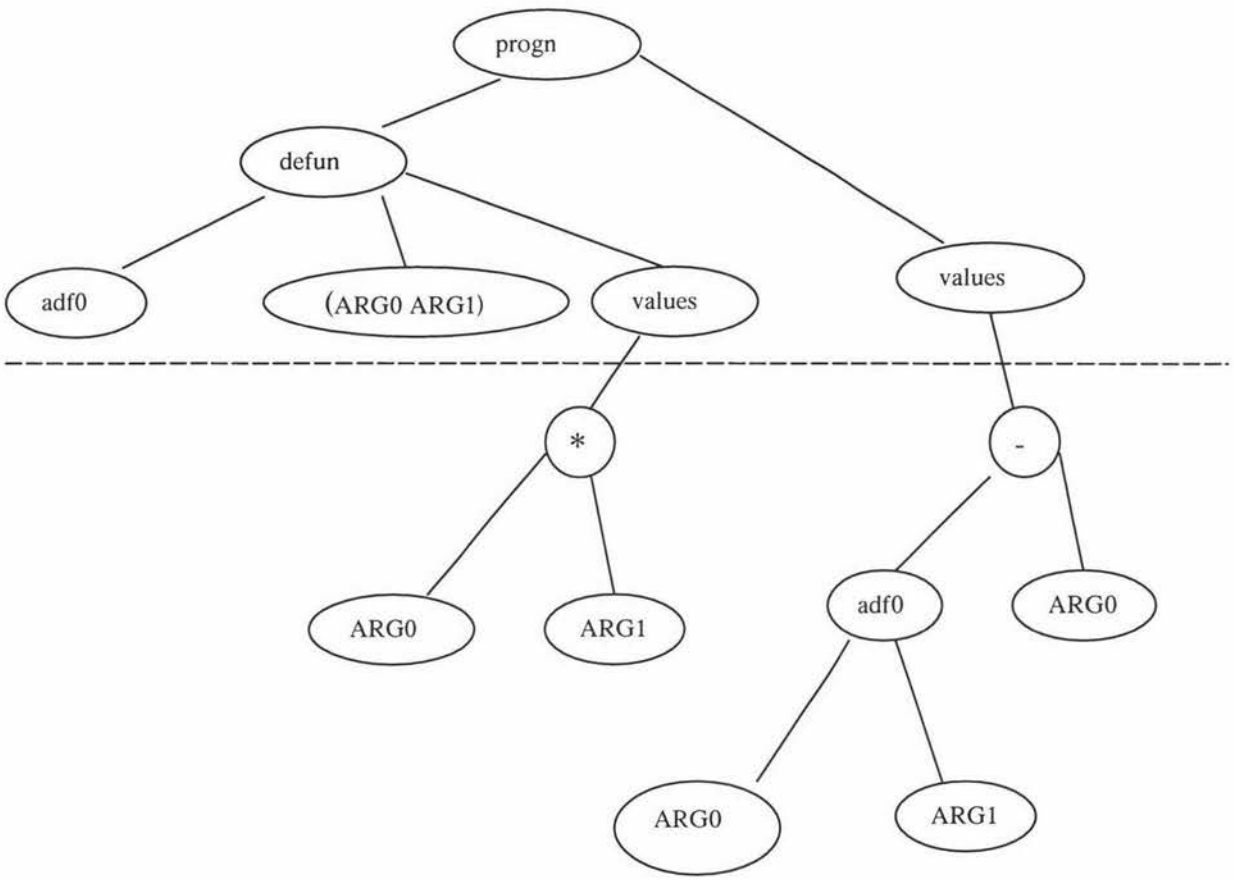

Figure 5.1.2 A program containing an ADF.

The first step of applying the automatically defined function method is to specify how the problem should be subdivided. In the example of figure 5.1.2, the root program consists of two arguments, the defun branch and the value branch. The defun branch is the function defining branch and the value branch is the result producing branch. This structure is static

across the entire program system. The function defined by the function defining branch of an individual is available for use by the result producing branch of that individual. Calling automatically defined functions is not predetermined, instead, it is determined by the evolutionary process.

Figure 5.1.2 shows the tree structure of an example program. This figure is divided into two sections, those portions of the tree above the dashed line and those portions of the tree below the dashed line. The portions of the tree above the dashed line represents the fixed structure of the procedure, which must be maintained in every individual in the population. The portions of the tree below the dashed line represent the parts of the procedure that can be evolved to solve the problem.

In order to use automatically defined function, two considerations that are different from standard genetic programming must be taken into count. First, the procedure that creates the initial population must be modified to generate individuals with the required fixed structure. Second, the genetic operators must be modified to maintain the fixed structure of the tree and to operate only on the modifiable areas within a tree.

To avoid the possibility of infinite recursion between the defined procedures, the genetic material which may be used in each evolvable branch of the tree is separately defined so that procedures defined within the main procedure may not call each other, but may only be called within the values branch of the main procedure.

Koza gives the functions that an automatically defined function can perform.

- Perform a calculation similar to that which a human programmer might use
- Perform a calculation unlike anything a human programmer would ever use
- Redundantly define a function that is equivalent to a primitive function that is already present in the function set of the problem
- Ignore some of its dummy variables
- Be entirely ignored by every potential calling branch
- Define a constant value
- Return a value identical to one of the dummy variable(so that the automatically defined function redundantly defines a terminal that is already present in the terminal set of the problem
- Call another automatically defined function with a subset of or a permutation of its dummy variables

Several sets of experiments have been performed to compare the performance of genetic programming systems that use automatically defined functions with those that do not [26]. The results of these experiments show that the use of automatically defined functions increases the power of a genetic programming system in terms of the difficulty of problems that it can solve. The technique also improves the speed at which solutions can be obtained (measured in terms of the number of individuals searched) for all but the most trivial of problems. In addition, the use of automatically defined functions generally results in more

compact solutions than those found without the technique. The reuse in the main routine of the subroutines evolved by the system allow for this reduction in program size.

5.2 Selection

Selection in genetic algorithms is the process of choosing which individuals in the population will be allowed to propagate their genetic material through application of a genetic operator. The selection operator does not create any new solutions, instead it selects relatively good solutions from a population. The takeover time is defined as the speed at which the best solution in the initial population would occupy the complete population by repeated application of the selection operator alone[9]. Selective pressure is the phenomenon that highly fit individuals have a high probability of being selected. If the takeover time of a selection operator is large, the selective pressure of the operator is small. If the selection operator has a large selective pressure, the population loses diversity quickly and may converge to a local optimum. If the selection operator has a small selective pressure, the population will converge slowly and reduce the ability to search the entire space. A different selection operator applies a different selection pressure to the population. The sections below will discuss a number of common used selection operators.

5.2.1 Proportional Selection

Proportional selection assigns to each individual a reproductive probability that is proportional to the individual's relative fitness. Proportional selection was first introduced by Holland [16] and has been used widely in many implementations of evolutionary

algorithms. In this scheme, the probability of selection of a single individual i in the population is characterized by the equation below:

$$p_i(selection) = \frac{f(fitness_i)}{\sum_{n=1}^{N} f(fitness_n)}$$

Here $f(fitness)$ is a mapping function that maps the value obtained by user defined fitness function to a non negative real value, which is proportional to the fitness value of individuals. N is the population size.

The fitness mapping function is crucial to proportional selection. An individual with the same fitness value will have a different selection probability when different mapping functions are used. For example, consider two individuals with fitness 1 and 2 respectively, the second individual has twice the selection probability of the first one. But if the mapping function adds 1 to all individuals, the fitness of first individual will change to 2 and the second one will change to 3. In this case, the second individual has only 1.5 times the selection probability of the first one.

In some genetic algorithms it is necessary to select an unreasonable individual to replace some individual in the population in order to increase the diversity. In this case, a possible method is to give a high reward to the poor individuals. This is the inverse proportional selection.

$$p_i(selection) = 1.0 - \frac{f(fitness_i)}{\sum_{n=1}^{N} f(fitness_n)}$$

In this case, the mapping function has the same form as the normal mapping function but poorly performing individuals have a high probability of being selected.

The scatter of fitness value may lead the population toward premature convergence. At the beginning of a run, an individual with a high fitness value will be preferred and produce more offspring. The similar structure of the string will occupy most of population. This will cause the search to stop at a local optimum.

5.2.2 Windowing

Windowing is a method used to avoid a local optimum by evaluating the fitness function as a time–varying transformation. It can be expressed as:

$$\Phi_i = f(fitness_i) - \beta(t)$$

Here Φ_i is the windowing fitness mapping function. $f(fitness_i)$ is the normal fitness mapping function that maps the raw fitness value to a non negative value. $\beta(t)$ represents the worst value seen in the last few generations. The number of generations used is given by the windowing size w. w is typically between 2 and 10.

Since $\beta(t)$ normally improves over time. In the later stages of search, windowing provides great selection pressure. However, poorly performing individuals may occasionally arise through mutation or recombination.

5.2.3 Sigma Scaling

Sigma scaling is based on the distribution of fitness values over the entire current population. When windowing scaling is applied, one problem is that particularly poor individuals may push the baseline very low. Sigma scaling is defined as follows:

$$\Phi_i = \begin{cases} f(fitness_i) - (\overline{f}(t) - c\,\sigma_f(t)) \\ 0 \end{cases}$$

If $f(fitness_i) > (\overline{f}(t) - c\,\sigma_f(t))$, the upper formula is applied, otherwise the lower formula is applied. $\overline{f}(t)$ is the mean objective value of the population. $\sigma_f(t)$ is the standard deviation of the objective values in the current population. c is a constant (e.g. c=2).

The sigma scaling method relies on the average fitness of the population, this may lead the premature convergence to a super fit individual.

5.2.4 Tournament Selection

In tournament selection, a group of q individuals is randomly chosen from the population. They may be drawn from the population with or without replacement. This group takes part in a tournament. That is, a winning individual is determined depending on its fitness value. Tournament selections are held with a group of size q, called the tournament size. q is usually between 2 and 10.

Tournament selection can be implemented very efficiently because no sorting over the entire population is required. Tournament selection also has the advantage of simple implementation. In proportional selection, a scaling technique will affect the result of the

selection. However, tournament selection only relies on the fitness of the tournament group, no scaling technique is needed. Tournament selection is also suitable for a parallel evolutionary scheme. Since the tournament group is a local parameter rather than from the global population. Evaluations are performed on the local tournament group.
Selection pressure is modified in tournament selection by varying the tournament size. Tournament size is equal to 1 corresponds to no selection at all. It is just a random selection from the population. As the tournament size increase it becomes more likely that the fittest individual will be chosen. For many applications in genetic programming, tournament size take the range between 6 to 10.

One interesting variant of tournament selection is the notion of competitive fitness, which is used to evolve game playing programs [11]. In games, which are non-trivial, it is extremely difficult to generate a correct, heuristic solution for specifying fitness. Competitive fitness requires the members of the tournament to actually play against each other in order to determine the fittest individual. Fitness is determined with the record of each individual in the competition. The method has three advantages. First, it does not require the user to have the detailed domain knowledge required to write the complete and correct fitness function. Second, a well defined fitness function does not provide for good differentiation between poorly fit individuals during the initial generations of the algorithm. Competitive fitness notes these distinctions by computing fitness in a way that improves fitness over time. Finally, since an individual may encounter many different strategies in a tournament, competitive fitness rewards those individuals that are able to compete well against a multitude of strategies.

In those cases where an inverse selection scheme is required to determine which member of a population is to be replaced, inverse tournament selection can be used. In this scheme, the individual with the worst fitness in the tournament is selected for replacement.

5.2.5 Rank – Based Selection

Hancook [15] summarised the different ranking selection that may be applied in practice. The basic idea is to sort the population from best to worst in the current generation, and then assign a reproductive or survival probability to each individual that depends on the rank ordering of the individuals. Linear ranking, non-linear ranking, (μ, λ) selection and (μ+λ) selection are methods of rank–based selection.

5.2.6 Linear Ranking

The most common form of ranking is linear ranking. In this scheme, the assignment function is a linear function that assigns individuals a fitness value which depends on the position of individuals in the queue instead of the actual fitness. If the index of the individual with highest fitness is μ-1 and the index of the individual with worst fitness is zero, the selection probability for individual i is defined as follows:

$$\Phi_i = \frac{\alpha_{rank} + \left[rank(i) \Big/ (\mu - 1) \right] (\beta_{rank} - \alpha_{rank})}{\mu}$$

where i is the index of individuals. α_{rank} is the number of offspring allocated to the worst individual. β_{rank} is the expected number of offspring to be allocated to the best individual during each generation. μ is the population size. The sum of the selection probability can be calculated:

$$\sum_{i=0}^{\mu-1} \Phi_i = \frac{1}{2}(\beta_{rank} + \alpha_{rank})$$

The total probability is 1. Thus, $1 \leq \beta_{rank} \leq 2$, and $\alpha_{rank} = 2 - \beta_{rank}$. If β_{rank} is set to 2, the worst string has no chance of reproduction. In principle, β_{rank} could be increased beyond 2 to achieve higher selection pressures, but then several of the worst strings would be given negative fitness values. These could be truncated to zero, but then the remaining fitness values would need rescaling to give the correct total number of offspring.

As an example, consider the population whose fitness is presented in table 5.2.1. The proportional selection probability is computed by dividing each fitness value by the sum of the fitness values. For linear ranking the population is sorted in decreasing order. The probability of each individual is computed using the formula given above with μ=10, and $\beta_{rank} = 1.5$ and $\alpha_{rank} = 0.5$.

The comparison results are listed in table 5.2.1

index	fitness	Probability with proportional selection	Probability with linear ranking selection
1	0.92	0.1226	0.1500
2	0.91	0.1213	0.1388
3	0.88	0.1173	0.1277
4	0.84	0.1120	0.1166
5	0.82	0.1093	0.1055
6	0.77	0.1026	0.0944
7	0.73	0.0973	0.0833
8	0.69	0.0920	0.0722
9	0.63	0.0840	0.0611
10	0.31	0.0413	0.0500

Table 5.2.1: comparison of proportional selection and linear ranking selection

Linear ranking provides increased selection pressure over proportional selection schemes. This is due to the weakness of proportional selection schemes in the latter portions of their search. As the average fitness of the population in a proportional selection scheme approaches the global optimum, the deviation between fitness values decreases. This drives selection probability of each member of the population in the limit to $1/\mu$. Thus proportional schemes fail to reward individuals with marginally better fitness values in the final stages of the search. Linear ranking provides a fixed advantage throughout the entire search process to those individuals with better fitness, thus capitalizing on minute differences in fitness in the later stages of the search process. Empirical experiments show that linear ranking improves selection pressure up to five orders of magnitude over proportional selection [3].

5.2.7 Exponential Ranking

Another ranking–based selection is exponential ranking. It is similar to linear ranking in that it uses an assignment function to determine the numbers of offspring. It is different from linear ranking in that the assignment function is not a linear function. The selection probabilities might be proportional to the square of the rank:

$$\Phi_i = \frac{\alpha + \left[rank(i)^2 \Big/ (\mu - 1)^2 \right](\beta - \alpha)}{c}$$

where $c = (\beta - \alpha)\mu(2\mu - 1) \Big/ 6(\mu - 1) + \mu\alpha$ is a normalization factor. α , β has the same meaning as in linear ranking and usually, $0 < \alpha < \beta$. Modifying the value of α and β will change the selection pressure. Increasing β and decreasing α will increase the selection pressure and vice versa.

5.2.8 (μ, λ) And (μ + λ) Selection

In (μ, λ) selection, each individual in the population is allowed to reproduce *k* offspring in the next generation. $\lambda = k\,\mu$, where μ is the best individuals selected in the current generation, and *k* is the expected number of the offspring of μ in the next generation. For $k > 1$, the best individuals will produce many offspring while some poor performing individuals have no chance to breed. This selection method results in a very strong selection pressure and is not commonly used in genetic programming.

In ($\mu + \lambda$) selection, the individuals that are going to be parents are selected from the union of the best μ individuals from the old generation and best λ individuals from offspring. The ($\mu + \lambda$) method always retains the best individuals unless they are replaced by superior individuals.

In these two methods, the best individuals always win against competitors. Selection pressure is high and takeover time is very low.

5.3 Genetic Search Operator

In evolutionary computing, an individual in the population represents a solution to the particular problem. A fitness value is adopted to measure how good the individual is. As evolving progresses, a new individual is generated by selecting one or more fit individuals and applying a genetic operator to the selected individuals. The mutation operator creates a new individual by changing one individual while the crossover type operator generates a new individual by combine some parts of two individuals.

5.3.1 Mutation

Mutation is one of the commonly used genetic operators in evolutionary computing. In mutation, one allele of a gene is randomly replaced by another to yield a new structure [16]. Mutation has the benefit of maintaining diversity in the population to avoid premature convergence. In a fixed–length genetic algorithm, mutation is necessary because at the early stage of a run, the string may contain some undesired bit that should be replaced by

another. As evolution is carried out, the population may retain similar genetic material, the desired genetic material may not exist in the population, mutation provides a mechanism to generate a new allele and put it into the population.

Koza [25] describes a mutation operation in which an entire subtree of the selected tree's copy is replaced with a randomly generated subtree. The depth of the subtree is controlled on a similar way to the random creation of the initial population. However, Koza argues that mutation is not important in genetic programming.

First, in genetic programming, particular functions and terminals are not associated with fixed positions in a fixed structure. Moreover, when genetic programming is used, there are usually considerably fewer functions and terminals for a given problem than there are positions in the chromosome in the conventional genetic algorithm.

Second, in genetic programming, whenever the two crossover points in the two parents happen to both be endpoints of trees, the crossover operation operates in a manner very similar to point mutation. Thus, to the extent that point mutation may be useful, the crossover operation already provides it.

Angeline[1] defines four distinct forms of mutation for parse trees. The grow mutation operator randomly selects a leaf from the tree and replaces it with a randomly generated new subtree. The shrink mutation operator selects an internal node from the tree and replaces the subtree below it with a randomly generated leaf node. The switch mutation operator selects an internal node from the parse tree and reorders its argument subtrees. The cycle mutation operator selects a random node and replaces it with a new node of the same type. If a leaf node is selected, then it is replaced by a leaf node. If an internal node is

selected , then it is replaced by a function primitive that takes an equivalent number of arguments. The four mutation forms are illustrated in figures 5.3.1 – 5.3.4.

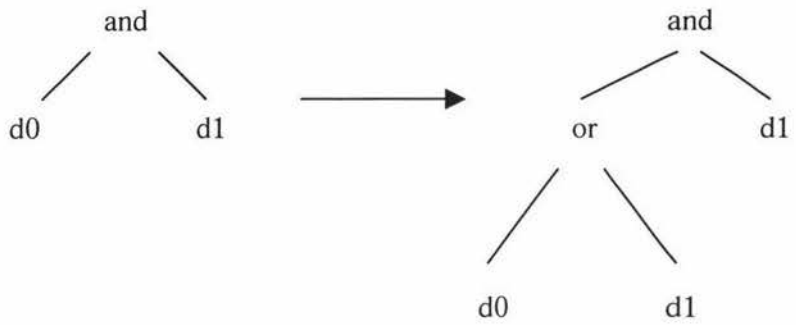

Figure 5.3.1 The grow mutation operator

Figure 5.3.2 The shrink mutation operator

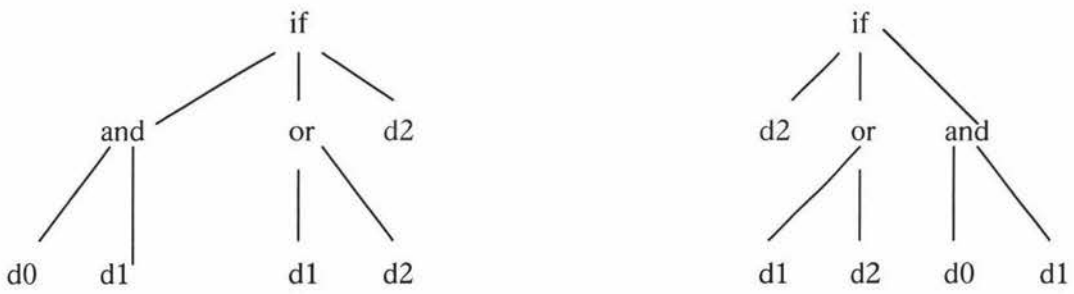

Figure 5.3.3 The switch mutation operator

Figure 5.3.4 The cycle mutation operator

5.3.2 Crossover

Crossover is the most commonly used operator in genetic programming, where genetic material is swapped between two parent individuals to produce offspring. Holland [16] introduces crossover in three steps. First, two individuals are chosen at random from the population of 'parent' strings designated by the selection operator. Second, one or more

string locations are chosen as breakpoints (crossover points) delineating the string segments to exchange. Finally, parent string segments are exchanged and then combined to produce two resultant 'offspring' individuals. The method used in Koza [25] is one point crossover. As the name suggests, only one crossover point is selected in parent individuals and the subtree below the crossover point is swapped between two parents. Koza [24] suggested that a selection bias should be applied to the parent's selection so that in most cases, a node is selected as a crossover point instead of leaf. This can balance the depth of nodes in the tree.

Takuya et al [20] introduced a depth – dependent crossover. In standard crossover, a node is selected randomly regardless of its depth. It has an influence on the resultant structure in two ways. First, the operator swaps larger parts of subtrees for shallower nodes. This leads to the propagation of larger parts of useful subtrees to the entire population. Second, the operator encapsulates a larger part of a tree, so that substructures for shallower nodes are protected from the destructive crossover. This random selection may not work well for the effective recombination or for the accumulation of building blocks. For depth–dependent crossover, a node is selected depending on its depth. The algorithm for depth – dependent crossover is:

Step 1: Given a tree, determine the depth d for applying the depth – dependent crossover.

Step 2: Select randomly a node of which depth is equal to d in Step 1.

Step 3: Apply the canonical (standard) crossover for the nodes chosen in Step 2.

A bias is applied to the selection of depth d so that the node is closer to the root. Takuya claimed that this improves the crossover performance and accumulates building blocks. As the generated programs become large, depth–dependent crossover does not work efficiently. This can be overcome by non–destructive depth–dependent crossover, in which each offspring is kept only if its fitness is better than that of its parent. In some experiments, non–destructive crossover generated smaller programs in Takuya's work.

Poli and Langdon[37] introduced a variation of crossover called one point crossover, which works by selecting a common crossover point in the parent programs and then swapping the corresponding subtrees like standard crossover. The crossover locations are chosen only from those parts where both parents have the same shape. A study [38] reports that one point crossover is valid and improves the performance of standard crossover significantly .

Uniform crossover [35] takes the same name as in genetic algorithms but has a different operation. In GA uniform crossover, the positions of parent strings are denoted as 0 or 1, offspring 1 copies the elements directly from parent 1 in those bit position marked by a 1. Offspring 2 copies the elements from parent 2 in those bit position marked by 0. Both offspring then copy the remaining elements from the other parent in relative order. Thus, each parent donates 50% of its genetic material to offspring. GP uniform crossover begins with the observation that many parse trees are similar in structure. The area in one tree that has a corresponding node at the same location in the other is called a common region. Those pairs of nodes within the common region that have the same arity are called interior. The GP uniform algorithm is: Once the interior nodes have been identified, the parent trees

are both copied. Interior nodes are selected for crossover with some probability. Non interior nodes within the common region can also be crossed, but the nodes and their subtrees are swapped. Nodes outside the common region are not considered. As reported, using this representation, performance on the even–6–parity problem is improved by three orders of magnitude compared with standard genetic programming.

D'haeseleer [10] describes two variations on crossover. The first variation is called strong context preserving crossover. In this method, subtree swapping is limited to those subtrees which are in exactly the same position in both parent trees. This is accomplished by first uniquely naming each node in both trees as a tuple of n coordinates $T = (b_1, b_2, \cdots b_n)$, which specify the path from the root of the tree to the subtree root. Where n is the depth from the tree root to the subtree root and each b_i represents its branch, in left to right order. If a node is identified in parent 1 with a matching node in parent 2, the nodes and their subtree are swapped. This is similar to GP uniform crossover except for the interior region.

The second variation presented is called weak context preserving crossover. In this variation, the same node naming scheme is used. The subtree selected from the first parent is taken from among the nodes in which there is a matching node in the second tree. The subtree selected from the second parent is a random subtree of the node that matches the name of the first parent's subtree. This modification maintains a certain sense of lexical locality and overcoming the problem with good genetic material being unable to migrate out of its position in the tree.

5.3.3 Hoist

Hoist was first introduced by Kinnear [22] [23] where a new individual is created by selecting a point inside a copy of an existing individual chosen by fitness proportional selection and elevating (hoisting) it up to be an entire new individual. Hoist results in a smaller structure in size than its parent and is useful for avoiding the individual structure expanding. It is a special case of crossover or mutation. This operator acts to protect useful genetic material in the population.

5.3.4 Editing

Editing was proposed by Koza [25] and is a simplification operator. The operation of editing is described as: If any function has only terminals as its argument, then editing will evaluate that function and replace it with the value obtained from the evaluation. For example, the expression (+ 1 2) can be simplified as 3, and a Boolean expression (NOT (NOT (X))) can be simplified as X. In an example of IF – THEN – ELSE function, if the test argument is always true, the flow will go to the THEN branch and the test is not necessary. The editing operator decreases the size of the tree and improves the evaluation efficiency. As a side effect, the editing operator removes some genetic material from the population that may be useful in the later time.

5.3.5 Encapsulation

In artificial intelligence and machine learning a key issue is how to scale a large problem into several smaller problems in an automatic manner. The encapsulation operation is a

means for automatically identifying a potentially useful subtree and giving it a name so that it can be referenced and used later. The encapsulation operation begins by selecting a highly fit individual from the population. A non – leaf node is selected from a copy of the selected tree. This operator removes the subtree located at the selected point and defines a new function to permit references to the deleted tree. The new encapsulated function has no arguments. The body of the new encapsulated function is the deleted subtree. For example, consider the LISP expression (+ A (- B C)); if (- B C) is selected as the subtree, a function name E0 may be given to refer to this encapsulated function. For efficient execution, E0 may be compiled. After replacing (- B C) with E0, the original expression becomes (+ A (E0)).

The effect of the encapsulation operation is that the selected subtree in the newly created individual is no longer subject to disruption through crossover, because it is now a single point in the individual.

The encapsulation operation is carried out on a copy of the parent individual, the parent tree is not affected by this operation. Since the parent is selected proportional to its fitness, the parent individual may participate in additional genetic operations during the current generation.

5.3.6 Permutation

Permutation operates on one parent individual. The parent tree is selected in proportion to its fitness. Permutation begins by selecting a function point of the tree at random. If the function at the selected point has k arguments, a permutation is selected at random from the

set of $k!$ possible permutations. Then the arguments of the function at the selected point are permuted in accordance with the random permutation. For example, consider the expression (+ A (% B C)), if the % is selected as the permutation point, the position of B and C is exchanged. This results in a new offspring (+ A (% C B)). Consider this expression: (+ A (+ B C)), if '+' in (+ B C) is selected as the permutation point, in this example it has no effect on the return value.

5.4 Fitness Functions

The problem solving method used by genetic programming is through a wide spread search within the problem solution domain. Heuristic search uses error information and general information about the problem domain in order to try alternatives that have more chance of succeeding. Genetic programming adopts heuristic search methods to solve the problem. The fitness value generated by the genetic programming system is evaluated and used to determine how close a solution is to the desired solution. Before the 'desired' solution has been found, a fitness function must be able to determine how 'better' or 'worse' an individual is. That means the individual's fitness can be compared by the fitness value. As a weak research method, in genetic programming, an explicit knowledge is normally not necessary. Instead, empirical credit assignment is usually adopted. This is especially useful in some complex applications such as gaming strategy.

At a high level, genetic programming performs a select – copy – modify loop during the evolution. In terms of an empirical approach, the information necessary for the genetic system to make a selection is the ranking of the population. Other information such as the

structure of individuals and their shape is not needed for a genetic system. The possible application of this approach creates new points in the problem space from the current population members. The new individuals may be considered as a new experiment being tested against the old one. And furthermore, it is used as the basis of future experimentation.

5.4.1 Supervised Learning

There many different methods used to assign a fitness value to an individual. The common method in genetic programming is supervised learning where a suite of training data is designed to train the possible solutions proposed during evolution. Genetic programming uses parse trees to represent the possible solutions in its search space. A training pair includes a set of training data and a target. Applying the training data set to the parse tree will result in a value. Comparing this value with the corresponding target will result in an error. Applying the whole training data suite to the parse tree will generate a set of errors. In order to accumulate the error, it must be converted to a positive value. The accumulated errors represent the distance between the resulting solution and the desired solution. The fitness function translates the distance to a fitness value, which is fed back to the genetic system for further genetic operation.

5.4.2 Objective Fitness

Typically in evolutionary computations, fitness functions return an exact, objective measure of the absolute worth of the parse tree. A fitness function represents a complete order of all

individuals in the population[9]. In some environments, this 'absolute objective' knowledge is easily obtained, but in some environment such as a gaming strategy, it is not possible to access such information. For example, in a game simulation, A, B, C are individuals in the population, A beats B, B beats C, and C beats A, an objective fitness function is not accurate enough to represent such an environment.

5.4.3 Relative And Competitive Fitness

Relative fitness determines an individual's worth by direct comparison to some other solution either evolved or provided as a component of the environment. Competitive fitness is one type of relative fitness measure that is sensitive to the contents of the population [9]. In this fitness measure, two individuals in the population compete with each other and the winner is rewarded. With such a measure, an absolute worth for each individual is not necessary because comparing two competitors gives a relative worth. The main advantage of competitive fitness is that it is self scaling. Competitive fitness measures do not care about the absolute fitness value of each individual but the relation. During evolution, only the 'better' individuals can survive and reproduce. One problem in using a competitive fitness measure is when the population size is large, the number of competitions is quite significant. Suppose the population size is N, then there will be

$$\frac{N(N-1)}{2}$$

competitions in each generation.

5.4.4 Multiobjective Fitness Function

When solving a real world problem, the user may desire that the solution must satisfy more than one criterion. For example, the user may desire a program which matches the input-output behavior of the training cases and also meets certain physical constraints on its size and on the amount of time it takes to run. This example requires a multiobjective fitness function.

There are three main classes of objective in fitness function measure. The most commonly used fitness measure in genetic programming is the error measurement. It determines how well a proposed solution is comparing to the desired target.

The second class of fitness measure is the length of the parse tree. In many modelling problems, solutions which not only fit the data but which are short are required. The parse tree evolved by a genetic programming system may vary in length while they have close fitness. One reason is they may contain redundant subtrees, or they may have evolved a different way of solving the problem. A shorter program usually indicates a more efficient solution to the problem while a longer one usually indicates waste of resources or redundancy. To limit program size, it is simply necessary to place a parameter of program size into the system. Editing is a method to combine several nodes into one and makes the tree shorter. Another method used to reduce tree length is through the mutation operation. This can be done by adding a bias to the mutation operation to produce small offspring. One implementation scheme, 'Minimum Description Length', may be applied to determine

the fitness of an individual in which the fitness value is scored not just upon how well the program performance but also on its size.

The third class of fitness measure is the run time of a program proposed by the genetic programming system. In most genetic programming systems, the solutions are restricted to non-iterative programs. But if the program is iterative or contains loops, the program may go into an infinite loop. In this case one program may halt the whole genetic programming system. Teller [42] introduces two solutions to this problem. The "popcorn" algorithm allows fitness testing to progress whilst it continues to do something else (which may increase its fitness) but imposes a maximum waiting time between events, once the waiting time elapses, fitness testing stops. The "anytime" algorithm requires the program to have its best estimate available on demand once a fixed time limit has expired. After this time the program is stopped, even if it is in the middle of a loop, and its fitness is based upon this answer (which is stored in an indexed memory).

Maxwell [31] suggested a co-routine execution model in which an external time limit is set for the system. If this time expires, the program is interrupted if it is still running, thus allowing other members of the population to be run. The interrupted program is given a partial fitness that can be used for selection scheme. The fitness value is determined not only by its distance to target but on the time used. The looping individuals are given a low fitness and are removed from the population eventually.

5.5 Multiobjective Optimisation

When an individual has more than one fitness measure, it would be best to optimise all performance measures simultaneously. In a real world problem, this is not always possible

because different fitness measure represent different aspects of the environment. Multiobjective problems generally do not have a single best solution. Instead, there are usually several acceptable solutions where no one solution is optimal in all of the objectives. Attempts to combine multiple objectives into a single numeric fitness value sometimes cause a genetic algorithm to suffer from premature convergence. In the multiobjective problem many trade-offs can be taken into consideration by a wide spread exploration of the problem space. One solution to this problem is to define fitness not as a single value but rather as a set of values. Each member of the set represents a measure of fitness on a single problem objective. The optimisation problem for such a measure is to find an individual whose set of measures is based on the trade off between the conflicting objectives of the problem.

There are many different methods that have been proposed to optimise multiobjective problems. The most obvious one is to combine all the objectives into one.

5.5.1 The Weighted Sum Approach

The weighted sum approach combines the independent fitness measurements associated with each objective into a single numerical fitness measure. The objectives are given weights defined by the user. The weights are a set of positive coefficients. The weighted objectives are then added together to form a single measurement for the individuals. This is then used for the selection scheme. This approach has been widely used in genetic programming system to solve multiobjective problems.

5.5.2 The Median – Rank Approach

In this approach, the population is first ranked according to each single objective separately. Then the median of the ranks is computed and used for fitness assignment. One type of median-rank method does not scale the rank, i.e. they all have the same importance. This can be modified by introduce a weight to ranks according to their importance.

5.5.3 Pareto Ranking

The Pareto criterion for one solution to be superior to another is for it to be at least as good in all attributes, and superior in at least one. A problem solution is defined to be Pareto-optimal if for each problem objective, the process of improving the value of that objective causes at least one other objective to degrade in order to retain a valid problem solution[9]. A problem solution A is Pareto-dominant over a problem solution B if for all of the problem objectives, solution A is closer to the optimal value than solution B. In the approach originally proposed by Goldberg, each of the members of the population are compared to determine domination relationships. All members in the population, which are not dominated by any other members are assigned cost one and placed in the first rank. They are then removed from consideration. The remaining members of the population which are not dominated by any other member which is still in consideration are then assigned a cost two and placed in rank two. This process repeats until all of the population has been sorted and ranked. At this point, the ranked population are used for standard selection scheme[9].

5.6 Avoiding Local Premature Convergence

In any problem solving algorithm, what we desire is a possible best solution to the problem domain. But in the practical application of genetic programming and genetic algorithms, there is a tendency that highly fit individuals will dominate the population as the evolution progresses. If a selection scheme with high selection pressure is employed, the probability that the solution converges to a local optimum instead the global optimum will increase. This happens because the less fit individuals with useful genetic material do not have enough chance of staying in the population. As the poor individuals disappear, so do the useful genes in the population. This observation results in a balancing between the efficiency and quality. Less diversity will lead a fast convergence with high probability of a local optimum, while more diversity will result in slow convergence but with high probability of global optimum.

Potts et al [36] described the three reasons that cause premature convergence.

(a) Loss of critical alleles due to selection.

If individuals that carry chromosomes are not selected for reproduction, critical alleles will be lost. Improving the selection technique may reduce the loss of critical alleles.

(b) Schemata disruption due to crossover.

When a crossover point between two alleles that are critical members of a certain solution hyper-plane is chosen, crossover disruption happens. Many variations of crossover and techniques have been studied to reduce the crossover disruption.

(c) Parameter settings such as mutation rate, crossover rate and population size.

Different parameter settings have different effects on the evolutionary system. The trade off is how to find a set of suitable parameters for the particular system with a high efficient search.

In the development of evolutionary computing, several schemes have been proposed to maintain diversity.

One possible method for maintaining diversity in a population is through the use of the mutation operator. The mutation operation involves random generation of genetic material, which may be a partial solution to the problem. Changing the mutation rate can vary the degree of diversity to the population. A high mutation rate corresponds to a strong degree of diversity. But Koza [25] considered that if the population converges to a local optimum, when the mutation operation is applied to an individual, the population often quickly re-converges. So, in practice, mutation does not help much in avoiding premature convergence. Because of the various operators proposed, some powerful mutation introduced in previous section and the use of an adaptive selection mechanism can be used to maintain diversity.

Potts et al [36] introduced an improved algorithm called GAMAS to cope with premature convergence. The GAMAS algorithm adds three new population based operators, migration, artificial selection, and recycling, to the simple genetic algorithm. The new architecture maintains four separate populations of individuals. Population I is used to maintain the set of most fit individuals encountered during the run in the other populations.

Crossover and mutation is not performed on this population. Population II is used for exploration of the search space. It runs using the simple genetic algorithm using a high mutation rate. Population III is used for exploitation of the search space. It has a low mutation rate, which allows recombination to dominate. Population IV is used for both exploration and exploitation with balanced probability settings for crossover and mutation.

Artificial selection occurs between the populations by maintaining population I as an incipient dominant species. At the end of each generation's run for the other populations, those individuals from population II, III and IV with better fitness than those in population I are selected to replace those in population I. Migration of genetic material between populations II, III and IV is forced after each generation to maintain diversity in the populations. Similar to simulated annealing, migration occurs at a higher frequency during the early generations and slows during subsequent generations. Population recycling occurs in populations II, III and IV by re-initialising the populations to random individuals after a fixed number of generations.

In their experiments, the GAMAS algorithm gives an optimal solution to problems significantly faster than the use of simple genetic algorithm.

Ryan [21] presents a method called the Pygmy Algorithm for maintaining diversity in genetic algorithms through the use of disassortative mating. The Pygmy Algorithm requires the maintenance of two lists of individuals, each with its own fitness function. The first population keeps the best individuals, in which the fitness function is just the normal fitness function i.e. the better the solution to the problem, the higher the reward it gets. Ryan calls

these individuals 'Civil Servants'. If an individual does not qualify to be a 'Civil Servant', it is placed in a second population. Ryan calls these individuals 'Pygmys'. The fitness function in this population is slightly different with a weighting for length. That means, the shorter the tree length, the higher reward it gets. These two population groups will take part in genetic operations later.

When selecting parents for breeding, each population has 50% chance of donating gene material. One parent is selected from the 'Civil Servant' population and one parent is selected form the 'Pygmy' population. The purpose of this selection scheme is so that children receive a short, efficient program from their parents. The 'Civil Servant' list ensures that no useful genetic material will disappear in the population. The 'Pygmy' list ensures that a compact solution is adopted by the children.

One method for maintaining population diversity is through the use of niching [9]. In this method the population is divided into several subpopulations. When individuals are selected for the application of a genetic operator, a subpopulation is selected and the parents are chosen from within that subpopulation. The operator is applied and the resulting offspring are replaced within the same subpopulation. Each subpopulation can be viewed as a simulated biological niche. Within the boundaries of a subpopulation the pressure of a strong selection mechanism can force convergence. These same boundaries prevent a fast convergence of the global population due to the fact that genetic operations can only occur within a subpopulation. In a pure application of this technique, several parallel searches are occurring within the global population.

In order to reach a global population convergence, it is necessary to relax the strong boundaries between the subpopulations. There are two ways in which genetic material may cross subpopulation boundaries. Husband [19] suggests a method that uses a multi-species co-evolutionary algorithm to divide a problem to many species. Each species is evolved under different evaluation functions and more than one species interact with each other. Husband claimed that this model is most coherent and effective when implemented as a distributed genetic algorithm with local selection operating.

Local mating is a selection scheme in which the probability of two individuals being chosen for crossover is inversely proportional to the spatial distance between the two individuals. Local evaluation is a competitive fitness scheme in which fitness is based on spatial locality. Collins and Jefferson stated that

> *"local mating is more appropriate for artificial evolution than the panmictic mating schemes that are usually used in genetic algorithms. In addition, local mating appears to be superior to panmictic mating even when considering traditional applications. Local mating (a) finds optimal solutions faster; (b) typically finds multiple optimal solutions in the same run; and (c) is much more robust than tradition genetic algorithm."*

D'haeseleer and Bluming [21] analysed the issue of locality of selection and fitness evaluation in genetic programming. Experiments involving local mating and localized competitive fitness were performed in the domain of conducting a battle between simulated robot tanks. Their experiment starts from an initially random population, from which it was possible to evolve passive strategies. Achieving a population with viable active and passive strategies is much more complex and difficult. This requires search, navigation, and attack behaviours to be generated. Two measurements are used to test the degree of diversity.

Calculating the correlation coefficients for every two individuals in the population, the average of these coefficients is called the phenotypical diversity. Taking the overall average of the correlation between all frequency signatures gives the genotypical diversity. The results of these experiments show that the use of locality results in a more diverse population without any loss in average global population fitness.

5.7 Seeding The Population

In the beginning of a genetic programming system run, it is generally required to perform an initialisation process, this seeds the population with genetic material for the search. Different seeding approaches have been studied in the recent years. In this section, we will introduce several seeding methods that are reported in the literature.

Koza [25] has introduced three different methods that randomly seed the population. The first one is a full method in which a tree is randomly generated so that all leaf nodes are at the same arbitrary maximum depth, i.e. the nodes over the maximum depth are all function nodes, and the nodes below the maximum depth line are all terminals. If the current depth is more than the maximum depth line, the function nodes have to be randomly determined and the numbers of the arguments of the function are also determined. The terminal slots have to be increased to match the arguments of the function. If the current depth is below the maximum depth line, it is easy to select a terminal from the terminal set and decrease the terminal slot until all the terminal slots are filled. This approach is illustrated in the figure below:

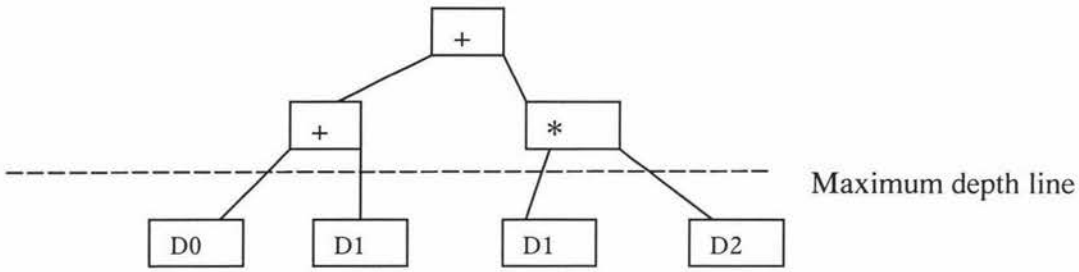

Figure 5.7.1 The full method

The second algorithm introduced by Koza is the grow method, this is slightly modified from the full method. In this method, the depth of leaf nodes in the tree may vary, but the maximum depth of the tree is still limited by a randomly generated maximum depth. When the current depth is above the maximum depth line, the node is not restricted to a function but a terminal. This choice can be weighted based on the proportion of terminals to functions at each depth. In practice this proportion increases as the depth increases. If a terminal occupies a slot, the branch is completed. If a slot is filled by a function, the branch just keeps growing. Once the maximum depth line is reached, all slots are filled by terminals. This idea can be illustrated by the figure below:

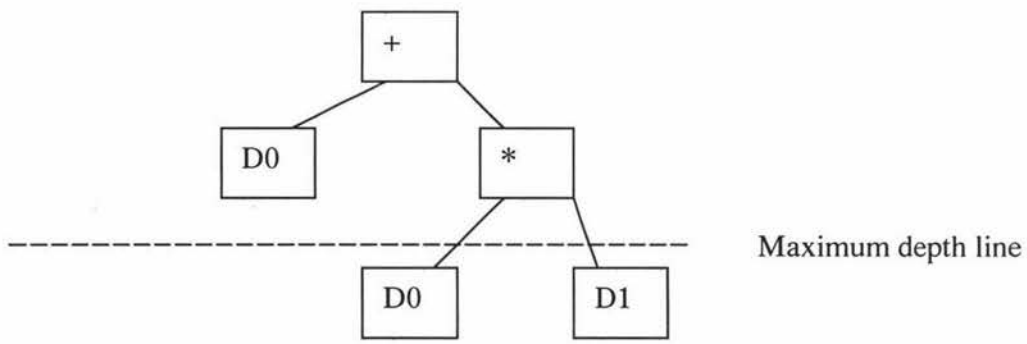

Figure 5.7.2 The grow method

The third method is a combination of the full method and the grow method. It is called the ramped half-and-half method. When initialising a population, the full method and the grow method are both applied to the initialisation process and each has fifty percent chance of being used. Ramping increases the variation of the tree shape and furthermore helps to improve the diversity in the initialisation process.

Ramsey and Grefenstette[39] introduced a case-based method. The basic idea of this method is to apply an agent system which contains a genetic algorithm based learning component and a separate execution component. Every time the execution component detects a radical change in the environment through feedback on its problem solving performance, the learning component restarts the genetic algorithm. The initial population is seeded with past examples of successful problem solving behaviours. While it is possible that no particular previous problem solving behaviour covers the current learning situation, it is hypothesized that elements of previous problem solving behaviours may be useful in the new environment. When this is true, seeding the initial population should give improved performance over random generation.

Langdon and Nordin [28] have tried to seed the population in a different way. In their experiments, the genetic programming system starts with a solution rather than a randomly generated population. This is done by seeding the initial population with perfect individuals that can already solve the fitness cases. The 'perfect individuals' are determined by a simple algorithm from the fitness cases used for training. Once the population has been

seeded, the system goes to next step. This method has been successfully applied to image compression and can be surely applied to other applications.

5.8 Strongly Typed Genetic Programming

In Koza's standard genetic programming system, one of the requirements is the satisfaction of the property of closure. That is, all functions have to accept arguments and return values of the same type. In practice, only one data type is defined in a genetic programming system. This is easy to implement and has proved an efficient method. But what if it is necessary to have more than one data type defined in the system [41]. Montana [32] introduced Strongly Typed Genetic Programming(STGP). STGP allows functions to accept arguments and return values of different data types. In STGP, each function argument, function return type and terminal is given a tag, which maintains the pre-defined data types for the elements.

Generation, crossover and mutation are constrained so syntactically correct S-expressions are produced. A randomly generated subtree must conform to the data type of the node in which the subtree will be located. After mutation, the mutation point must keep the same data type. Similarly, crossover is carried out only between two subtrees, which have the same data type at the root node.

5.9 Compiling Genetic Programming System

Nordin [33] proposed a Compiling Genetic Programming System (CGPS), which uses binary machine code to represent the parse tree in the population rather than a high level language expression. Machine code in the individuals is called by a standard C function call

and no intermediate language and interpreter are needed in such a system. This can improve the computation efficiency significantly.

A function in machine code consists of four parts. The first part is the header, which deals with the administration, necessary when entering a function. This section is often constant and can not be changed by the mutation or crossover operator. The second part is the footer, which is the opposite of header. It does the clean up operation after the function call. Again, this is also constant. The third part is a return instruction, which forces the system to leave the function and return control to the calling procedure. The length of a program is controlled by this part. The fourth part is the function body, which consists of the actual program to evaluate the function.

As reported, the CGPS is 1000 times faster than the standard genetic programming system in Prolog.

5.10 Parallel Algorithm Discovery And Orchestration (PADO)

The PADO [43] architecture takes signals or images as input and outputs the correct class of a signal or image. In the PADO architecture, programs are made up of several groups, each group being used to recognise a particular class of image. The group with the highest output is used as the overall output of PADO. The output of a group is a linear combination of the output of the programs included in the group. Programs are represented as directed graphs of nodes. As an arbitrary directed graph of N nodes, each node can have as many as N arcs. The arcs represent possible flows of control and nodes perform actions and take decisions on which of the possible control flows should be followed. Each node includes an action, a branching condition and a stack. Programs have access to shared indexed memory.

In general, loops may form in the directed graph so causing the programs to execute indefinitely. To allow for this, programs are executed for a fixed time after which they are aborted and their answer is extracted from a designated cell in the indexed memory. The algorithm of PADO is described as follows.

```
Function PADO(population, signals, C, S) returns C groups of S algorithms
      Inputs: population: a set of P randomly generated algorithms
            signals: a set of training signals
            C: the number of classes
            S: the number of algorithms from each class to return
      Repeat
            Loop over signals
                  EvaluateFitness(population, signal i )
            Split population into C distinct subpools of size P/C based
            on fitness
            Loop i from 1 to C
                  MatingPool i ← Reproduction (SubPool i )
                  NewSubPop i ← Recombination (MatingPool i )
            Population ← Σ NewSubPop i
      Until return requested
Return the most fit S algorithms in each of the C subpools
```

6. Analysis And Design

In order to design a system that for a genetic programming system, it is necessary to consider some issues that involve the working environment, input and output. They are stated as follows.

6.1 Requirements

Requirements are given bellow:

1. The system must be able to read a data set, which is stored in a text format file. The data file contains five columns in which that first four are the training set, the last

one is the target. The file is organised in rows where each row represents a training – target pair.

2. The parameters requirement such as population size, program size must be predefined. Once running, the system cannot change these parameters.
3. The system must be able to implement genetic operations such as crossover, reproduction and initialisation, in order to search the solution space of the problem domain.
4. At any time, the best solution so far must be recorded so that the best solution can be retrieved even before termination.

6.2 Four Point DCT

Our target is to evolve a 4×4 block DCT. Based on the analysis of chapter 1, it is more efficient to implement a 2 dimensional DCT in terms of 1 dimensional DCTs using the row – column approach. We follow this approach and try to evolve a 4 point one dimensional DCT. It is possible theoretically to evolve a parse tree, in which the entire 4 point DCT can be translated. In practice, however, it was found to be almost impossible to evolve such a program. In fact, it is not necessary to perform the entire 4 point DCT using one translation formula. An easier way is to transform the 4 point DCT one point at a time. The computational complexity can be reduced by such a way and implementation becomes simpler.

6.3 Representation

Because we are performing the DCT one point in a time, it is obvious that one parse tree is a good way of expressing a one point translation. Four parse trees are required to

accomplish a 4 point DCT. The nodes are functions that are selected from a function set and the leaves are terminals that are from a terminal set. The terminals may be input data or random numbers provided by a text file or randomly generated by the system.

6.4 Working Environment

We have chosen Microsoft Visual C++ as our working environment because of its convenience and many features. When working with Visual C++, a number of classes are automatically created upon setting up a project. Visual C++ also provides a context device that is very convenient for displaying the status of the system while it is running. Threads are easy to create and maintain, so the genetic programming system can be run in parallel with other applications. The system structure is clear and easy to maintain by using Visual C++. Programs written in C++ are generally more efficient than those written in LISP. These reasons lead us to work with Visual C++.

The hardware environment was as follows:

Operating system:	Windows 95
CPU:	Cyrus 333
RAM:	160M

6.5 Input And Output

The system reads training data from a text file in which the 32 set training – target data pairs are located. Each row represents a set of training – target data pairs, which contains five numbers. The first four numbers are the training set while the last one is the target.

Training data is selected randomly from an 8–bit bitmap image with a range of 0 and 255. This data is converted to a range of between –128 to + 127. Since we are creating a training data set, the translation speed is not an important issue here. Any transform method can be applied to accomplish the DCT. The standard DCT is used for translation in our implementation. That is:

$$T(k) = c(k)\sum_{n=0}^{3} S(n)\cos\frac{\pi(2n+1)k}{8}$$

where $c(0) = 1/2$, and $c(k) = \sqrt{1/2}$, $k \neq 0$.

As the four points are translated separately, we have four files after translation. Each file records one point of the data set.

During run time, improving solutions are found continuously. Each time a better individual has been evolved, the system displays its fitness value and the distance between the current value and the target value for each fitness case, and saves the 'best so far'.

6.6 The Complexity Of DCT

The reason why we chose a 4 point DCT instead of a general 8 point is because of the computational complexity of the genetic programming system. For an 8 point DCT, the translation difficulty varies for each point. The first and the fourth point are more easier than the others. Genetic programming solves problems by very widespread search in the solution domain to find a best solution. The sufficiency requirement must be satisfied when

we supply the system with a function set and terminal set. For an 8 point DCT, we tried to supply the terminal set with 8 input data points and 8 random numbers. The terminal set contains 16 elements in total. Because of hardware limitations we found that this problem was just too complex to be solved in a reasonable length of time.

6.7 Classes

In addition to the initially–generated classes from Visual C++, we write two more classes. The first class is CPopulation, which defines the behaviour of individuals in the population. The integer argument in the constructor is the index of an individual that is used for addressing. Note that the crossover operator is not defined in this class, since crossover takes two individuals as arguments. It is defined outside of this class. The class definition is listed below:

```
class CPopulation  {
public:
     CPopulation(int i);
     virtual ~CPopulation();
     void Mutate();
     CTree *GetTree() {return tree;}
     float GetFitness(){return fitness;}
     float TestFitness();
     void Initialise();
     void CopyPopulation(CPopulation *pop);

private:
     CTree *tree;
     float fitness;
     int position;
};
```

The second class is CTree, this defines the lower level operations on an individual. The aString is where the parse tree is located and frAux stores randomly generated numbers.

The CTree class definition is as follows:

```
class CTree  {
public:
     CTree();
     CTree(const CTree* aTree);
     virtual ~CTree();
     void RandomTree();
     float ProgramValue(int i);
     char *GetString(void) {return aString;}
     void SetString(char *s){strcpy(aString,s);}
     float GetAux(int i) {return frAux[i];}
     void SetAux(int i,float f) {frAux[i]=f;}
protected:
     void PushOperator (char op);
     void PushValue(float value);
     int RandomFunction(void);
     int RandomTerminal(void);
     void RandomAux();
private:
     char aString[MAX_PROG_SIZE+1];
     float frAux[AUXSIZE];
};
```

6.8 High Level Algorithm Description

After running the system, a thread is created, which enters a while loop after initialisation. The genetic operations used in the system include create, reproduction, crossover and mutate. An overview of the algorithm is described as follows:

```
t : = 0 ;
initialise population P(t);
evaluate population P(t);
while not terminate do
     P'(t) : = create P(t);
     Evaluate P'(t);
     Q : = tournament selection P'(t);
     P"(t) : = reproduce Q;
     P"(t) : = crossover Q;
```

```
    P"(t) : = mutate Q;
    evaluate P"(t)
    t : = t + 1;
end while
```

P(t) is a population of candidate solutions, Q is a special set of individuals that is considered for selection. t is the generation.

6.9 Implementation

In a genetic programming system, as stated in the previous chapter, many parameters have to be considered carefully. Function set, terminal set, structure, fitness measurement, selection process, genetic operators and parameter setting are all key. Incorrect parameter settings will lead the system to converge prematurely or not at all. In this section, we will describe the implementation of our system.

6.10 Tournament Selection

The selection operator selects the more fit individuals from one generation, causing the population to become more adaptive to the environment. The goal of the selection process is to increase the average of the population while maintaining some diversity. A selection operation with very low selection pressure will fail to improve the population as a whole. A selection with very high selection pressure will quickly converge to a local optimum, at which a possible better solution is behind the scene.

In this experiment, Tournament selection is chosen for the selection process in which a small random sample of individuals of the population is taken. Tournament size is 4. The best two individuals are selected as parents. To these parents are applied the genetic

operators such as reproduction and crossover. The worst two individuals may be filled with newly generated individuals, which are the product of genetic operators. In tournament selection, the parents remain in the population after the genetic operation. Only the worst individuals are replaced by newcomers. This scheme is called a Steady State system. Differing from Roulette Wheel Selection, the worst individual has no chance to breed while applying tournament selection. The advantage of tournament selection is that less global variables have to be maintained when considering the coding process. Compared to ranking selection, a sort of the entire population is not required in tournament selection.

6.11 Fitness Measurement

The genetic programming system needs to use a measurement to determine the fitness level of a specific individual. Fitness measurement is the pressure for driving system evolution. The fitness function should reflect every potential error between the target and the current case. The fitness function should give an improved solution a high reward and give a penalty to a poor one. The target value is a simple floating-point number in our experiment. Our fitness function is based on the square of the difference between target and actual value

.

$$Error = (C - D)^2$$

C is the current evaluated value. D is the target value.

The algorithm to calculate the total errors for all fitness cases can be described as follows:

```
total error : = 0;
for each fitness case i
     Ci  : = evaluate fitness i;
     Error = (C_i − D_i)^2 ;
     total error : = total error + error;
end for
```

Use of the square of the difference not only gives a positive error but also enlarges the error. This will give higher reward to the improved population. We add up the error for all fitness cases and have a global error for a particular individual. It is then convenient to convert the total error to a percentage value. Thus, we set the fitness as:

fitness=total error / (total error + constant)

This will result in a fitness value with the range between 0 and 1. The constant value is not a vital factor for the solution but gives a convenient measure.

Limited error fitness (LEF) [13] is adopted in our experiment, which is a modification to the standard supervised learning in a genetic programming system. With LEF, an error limit is introduced to stop the evaluation process when the total error exceeds the error limit. When the limit is reached no further test cases are evaluated. The fitness of the individual is given a very low value. The LEF technique saves CPU time but gives a similar result to standard GP. In this experiment, the error limit is set to a fixed value for simplicity.

6.12 Genetic Operators

The genetic operators used in this experiment include create, reproduction, crossover, and mutation.

6.12.1 Create

The create operator is performed in the same way as the process of initialisation where a complete new individual is introduced into the population. Create is a special case of mutation and provides new genetic material and increase the search diversity.

6.12.2 Reproduction

The reproduction operator encourages good performing individuals to dominate the population. Tournament selection with a tournament size of four is the selection mechanism used in this experiment. To undergo reproduction, four individuals are randomly selected and sorted according to their fitness value. The worst two individuals are copied from the best two individuals. To save CPU time, fitness value is not evaluated but copied from parent to child. The best two individuals remain in the population after reproduction.

6.12.3 Crossover

Crossover happens between four selected individuals with the best two individuals used as parents. A crossover point is selected in each parent separately before undergoing crossover. If the selected point is a leaf, it is a complete subtree itself. If the selected point is a node, the complete subtree under that point has to be found and identified. These two subtrees are exchanged and spliced into other individuals such that two new individuals are generated which take over the worst two individuals. The length of the newly generated individuals may exceed the maximum program length after crossover. In this experiment, if

the length of the new individual exceeds the maximum program length, it just remains the original parse tree. In this case, only one new offspring is bred. In case both lengths of two offspring exceed the maximum program size, a new crossover point has to be selected again.

6.12.4 Mutation

In terms of standard mutation, any point inside the parse tree may be selected as a mutation point. The subtree under the mutation point is removed and is replaced with a newly generated subtree. The method of generating a new subtree is just like individual initialisation. Mutation provides new genetic materials for the population which extends the search space. In [27] (2.4.6), several variations of mutation are introduced. Mutating Constants at Random changes the value of a constant randomly so that noise is introduced. The Mutating Constants Systematically variation can behave in different ways. In this variation, mutation can replace the input variables by constants or vice versa. Mutation can use simulated annealing to update numerical values. In this experiment, we follow this variation in which "a numerical partial gradient ascent is achieved to reach the nearest local optimum". The constant that is selected as the mutation point randomly increases or decreases by 5% when the mutation operation is performed. This mutation operation variation helps a relatively stable individual move toward a possible optimum.

6.13 Function Set

One of the important problems in genetic programming is to select the function set. Koza[25] suggests " always pick the most powerful and useful seeming functions from the problem domain that you can think of". The arithmetic operations are chosen in our experiment. For the consideration of efficiency and sufficiency, we take the function set F={ADD, SUB, MULT, DIV} for all the point transformation. The functions ADD, SUB, MULT, DIV in the function set respond to +, -, *, / respectively. Here, ADD, SUB, MULT are just the normal arithmetic operations. DIV is a protected division. That means 1 will be returned if division is by 0 and no error is produced. Operators in the function set take two arguments. To achieve the closure property, arguments have the same data type – float.

6.14 Terminal Set

Another important problem in GP is the selection of a terminal set. The main requirement of the terminal set selection is that of sufficiency[25]. How to make sure the terminal set satisfies the sufficiency property is a problem. There are many different rule based DCT algorithms such as the original DCT and the fast DCT. Different DCT algorithm use a different constant matrix in the translation procession. In terms of genetic programming, we never assume any previous knowledge of the rule based DCT. To choose how many constants to use, our balance is between sufficiency and efficiency. More constants may potentially converge to a better point but take longer time. Conversely, less constants may break the rule of sufficiency but converge easily at which the possibility of local converging to a optimum increases.

In this paper, the task is to evolve a 4 point cosine transform, with one tree for each point. We may of course give the 4 transforms the same terminal set.

$$T=\{I0, I1, I2, I3, AUX0, AUX1, AUX2, AUX3\}$$

Where, I0 to I3 represent the source input data. AUX0 to AUX3 are random numbers. $0<AUXn<1$. We know the different target points have different association with source data. In our experiment, we simplify the terminal set for target point 0 and target point 2.

$$T=\{I0, I1, I2, I3, AUX0\}$$

This reduction gives a good result, and is more efficient.

7. Testing And Results

7.1 Parameter Setting

The results shown below were obtained using the environment listed in the chapter 6. All the runs are based on tournament selection with tournament size of four and the following parameters.

- Creation probability: 0.02
- Crossover probability: 0.7
- Mutation probability: 0.02
- Population size: 500
- Maximum program size: 50 (for point 0 and point 2)
- Maximum program size: 80 (for point 1 and point 3)

7.2 Fitness Function

The fitness function applied to the system for all of the four points is the same:

$$fitness = \frac{50.0}{50.0 + total\_error}$$

total_error is defined as the square of the difference described in the previous chapter.

7.3 Running Convergence

The results are summarised in figure 7.1 to figure 7.4. The figures show 15 runs for each point in which point 0 and point 2 evolved over one million generations

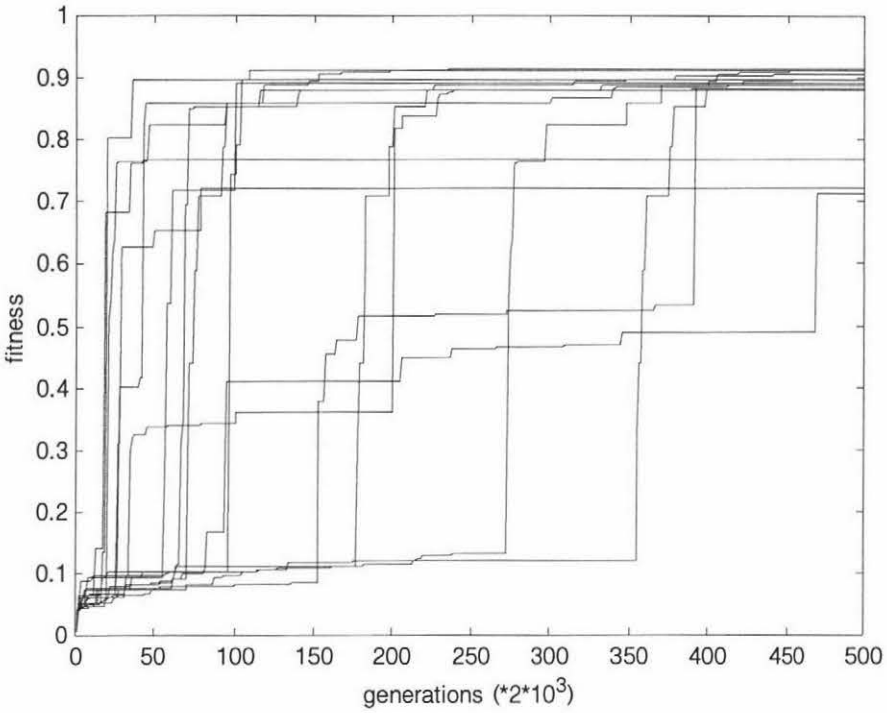

Figure 7.1 Result for point 0 over 15 trials

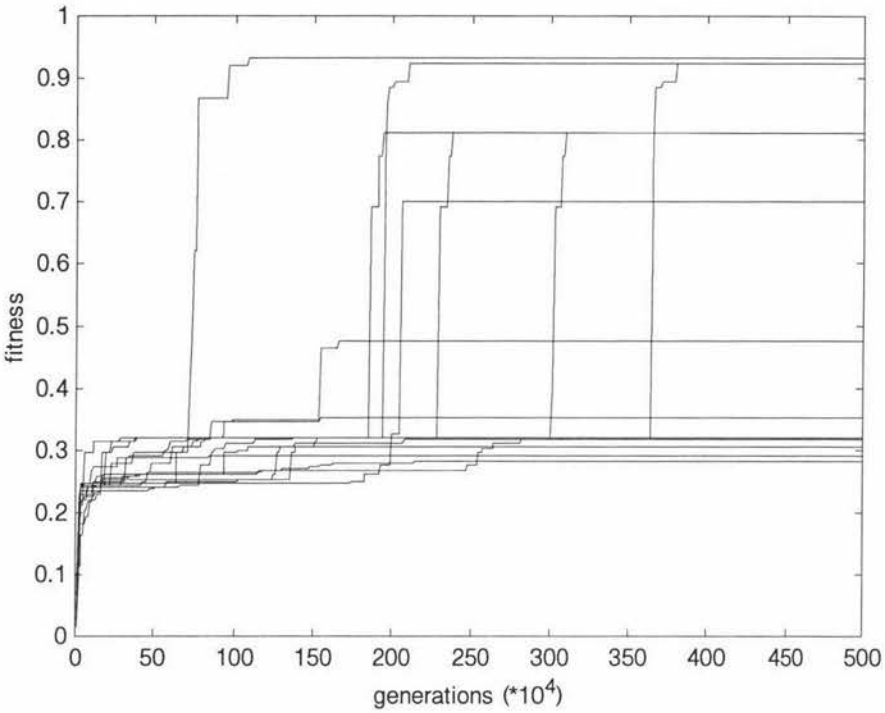

Figure 7.2 Result for point 1 over 15 trials

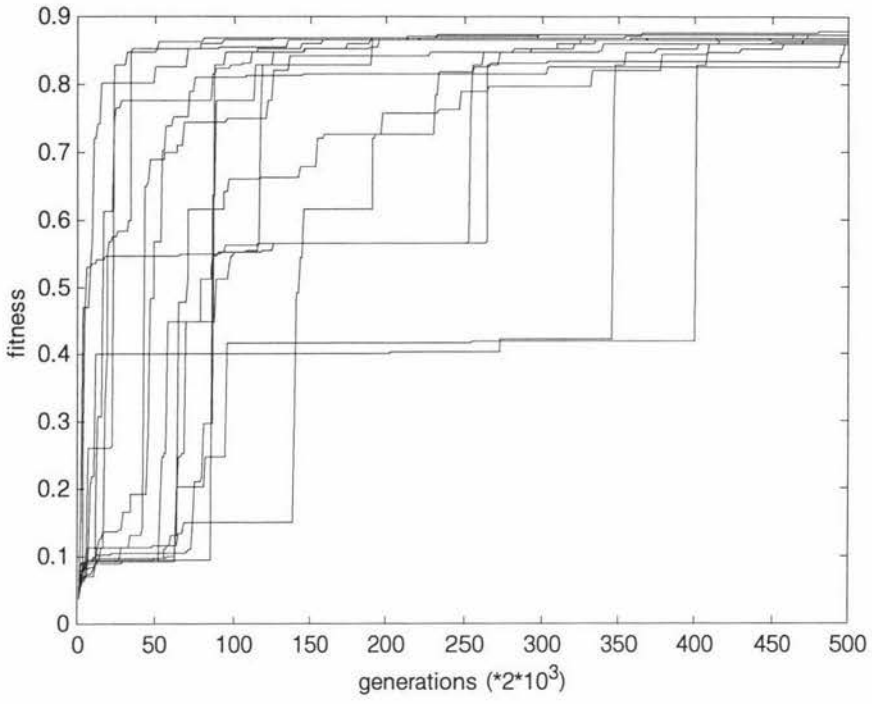

Figure 7.3 Result for point 2 over 15 trials

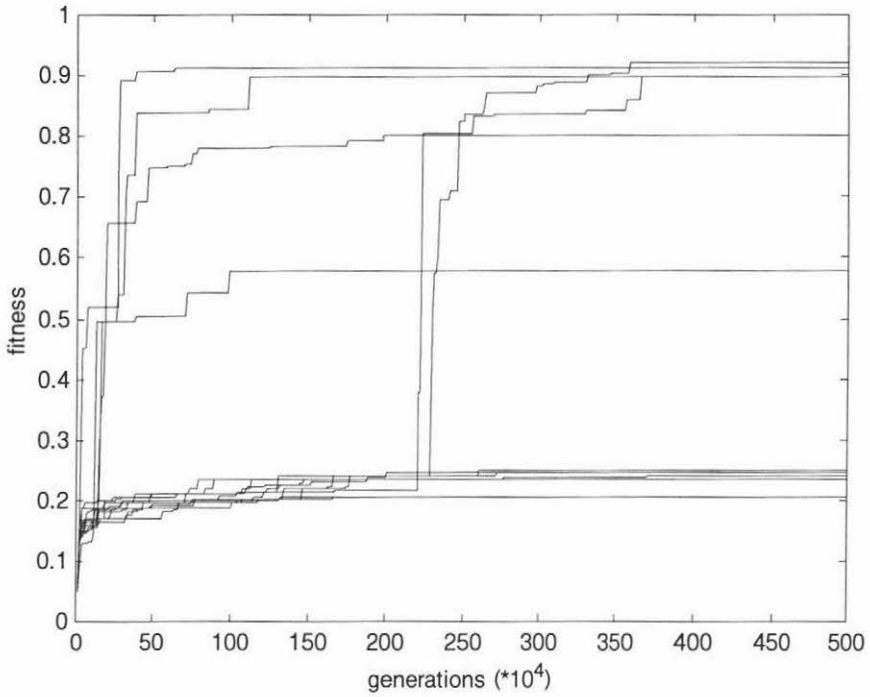

Figure 7.4 Result for point 3 over 15 trials

Point 1 and point 3 are for five million generations. The run times are between 1 and 6 hours for each run. Point 0 and point 2 take less time than point 1 and point 3 to converge to optimum solutions. This is because of the difference in complexity of the problems and the total number of the elements in the function and terminal set.

7.4 The Best Results

Parts of the test data are randomly selected from a 8-bit black and white bitmap picture while some of them are the result of the DCT. The test data is listed in Appendix A. The raw evolved parse trees are as follows:

Point 0:

((0.504865)*(((b)-(a))-(c)))-((a)-((d)*(0.504865))+(0.504865)-

(div(a,a)))+div(div(div((0.504865)-(d),c),(b)-(a)),(d)-(0.504865))+c+a+a

Point 1:

(0.638957)-((0.638957)*(((0.372204)-((((b)-(c))*((0.638957)*(0.638957)))-(d)))-(a)))

Point 2:

((a)-(b)+(d+0.449478)-(div(b,div(a,0.449478))+c)+div(d,div((b)-((c)*((d)-(c))+(div(a,c))-

((b)*(d))),(b)-(0.449478))))*(0.449478)

Point 3:

((div(div(((0.389072)*((0.134196)*((div(0.427715,b))*((0.427715)-(0.389072)))))-(((d+c)-

(0.134196))*(d)),d),((b)-((a)-(a)))-(b)))*((0.604979)*(((0.134196+(a)-(d))*(0.427715))-

(((b)-(0.604979+(c)-((0.389072)-(0.134196))))-(0.134196)))))-

((0.134196)*((0.604979)*(0.427715)))

We simplify the raw programs by hand since they contain redundant code. The simplified programs are as follows:

Point 0:

0.504865*(b-a-c+d)+0.495135+div(div(div(0.504865-d,c),b-a),d-0.504865)+c+a

Point 1:

0.638957-(0.638957*((0.372204-((b-c)*0.408266-d))-a))

Point 2:

(a-b+d+0.449478-(div(b,div(a,0.449478))+c)+div(d,div(b-(c*(d-c)+

(div(a,c))-b*d),b-0.449478)))*0.449478

Point 3:

0.604979*(((0.134196+a-d)*0.427715)-(b-0.484299-c))-0.034724

The fitness of the four programs computed by using the 32 set test data is listed in the following table:

Point	Fitness
Point 0	0.9141
Point 1	0.9329
Point 2	0.8778
Point 3	0.9204

Note that the total error is the sum of the error squared, which magnifies the actual error value. For the point 1, the fitness of the evolved program is 0.9329, this resulted in a total difference of 3 between target data and computed data for all 32 set test data. This result is very close to the ideal target. For detailed reference, the differences between the target data and the computed data are listed in Appendix B.

For point 0, in one run, an interesting result is evolved. It has the form:

(c+b+d+a) * 0.50086

This solution has the fitness value of 0.9022.

For point 2, in one run, an easy to understand solution is also evolved:

$$0.456939*(0.456939-c-b+d+a)$$

This solution has the fitness value of 0.8635.

For point 1, although we supply 4 random numbers in the terminal set, the best result contains three random numbers only. This result shows that genetic programming converges to a solution that is problem oriented instead of a combination of the elements in the function and terminal set.

7.5 Random Search Comparison

In order to compare the evolutionary algorithm and random search algorithm, we have evaluated the randomly created individuals for each point. The results are listed as follows. Five million individuals are evaluated for each point. Fitness range is from 0 to 1 which is divided into 20 ranges. The first range contains individuals that have the fitness value:

$$0 <= \text{fitness} < 0.05$$

The following ranges have a similar meaning.

Range	Point 0	Point 1	Point 2	Point3

0	4,999,947	4,998,790	4,999,137	4,935,424
0.05	52	1122	858	64512
0.1	0	44	3	60
0.15	0	28	1	4
0.2	0	16	1	0
0.25	0	0	0	0
0.3	0	0	0	0
0.35	0	0	0	0
0.4	0	0	0	0
0.45	0	0	0	0
0.5	0	0	0	0
0.55	0	0	0	0
0.6	0	0	0	0
0.65	0	0	0	0
0.7	1	0	0	0
0.75	0	0	0	0
0.8	0	0	0	0
0.85	0	0	0	0
0.9	0	0	0	0
0.95	0	0	0	0

The results of the random search shows that most of the individuals have a very low fitness value. Surprisingly, one individual for point 0 has a high fitness value of 0.7, which may reflect the fact that point 0 evolution is the easiest problem of all. From the distribution, we can see that a random search algorithm is so weak that it is almost impossible to find a good solution.

7.6 Eight Point DCT

Parameter setting is the same as for the four point DCT except the maximum program size is set to 200, the terminal set contains 8 random numbers. Four trials have been tested in our experiment in which each trial runs for five million generations. For point 1, the evolved best individual has the fitness value of 0.225. The time spent on each run is about 10 hours. The result is illustrated in figure 7.5.

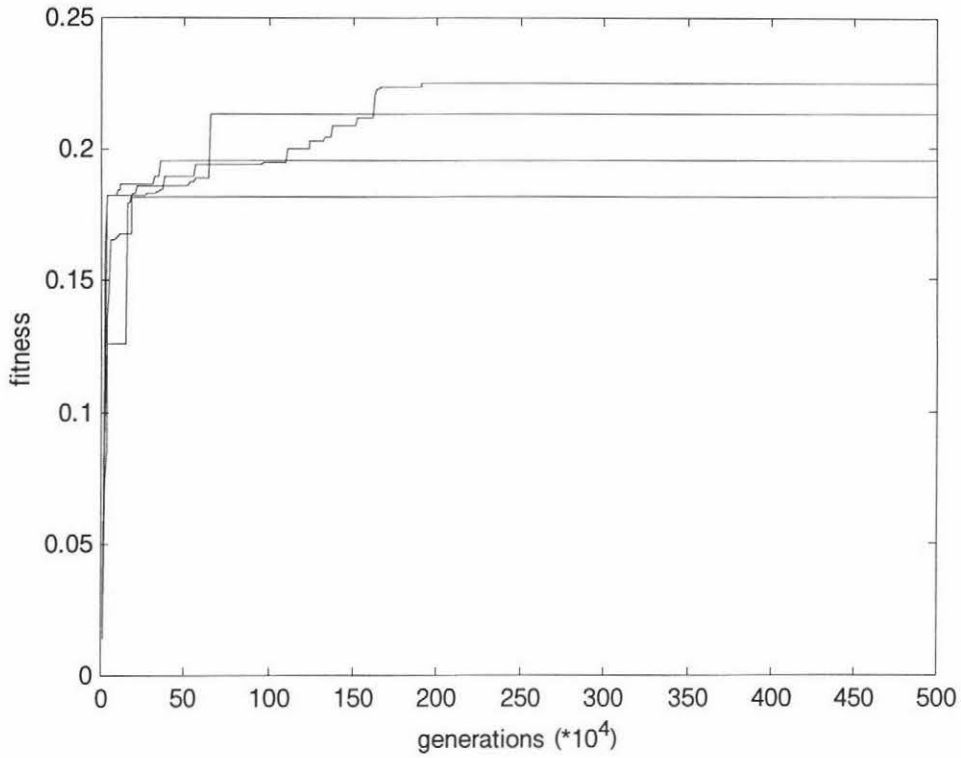

Figure 7.5 Running result for point 1 over 4 trials

7.7 Summary

The results have shown that the DCT can be evolved by means of the genetic programming paradigm. This result uses 15 multiplications and 32 additions to carry out a 4 point DCT. The purpose of this investigation was not to evolve a fast DCT but to explore the possibility of evolving a DCT by means of genetic programming. Evolution of a fast DCT and a 8–point DCT are suggested as future work.

8. Conclusion

Research into the DCT continues to be an important topic in transform coding and signal processing. Different algorithms for calculating the DCT were presented in chapter 1. All these algorithms are based on an exact ruled based approach. In this paper, we have successfully evolved an approximate evolutionary algorithm based DCT. Koza first showed the power of the genetic programming paradigm in widespread application in [25]. Here, we have shown one more the power of genetic programming by evolving a symbolic regression.

Our result includes four separate parse trees for four points. By examining the results, the evolved parse tree for point 1 and point 3 have higher fitness than point 0 and 2. This may reflect the different parameter setting between point 0, point 2 and point 1, point 3. There are four random numbers in the terminal set of point 1 and point 3 while only one random number in the terminal set of point 0 and point 2. More random numbers in the terminal set may make the solution easier to fine-tune. To improve the results, we might add some more random numbers for point 0 and point2, and reduce the mutation step (say 1%).

We have successfully evolved a four point DCT in this experiment. The successful evolution of the four point DCT implies that an eight point DCT is also possible. This is the most useful size because it is used in many standards such as JPEG.

Employing an object oriented language as our implementation language makes the programming task much simpler. The time spent evaluating a parse tree written in C++ is

much less than in LISP. Visual C++ provided a flexible and simple method for designing the user interface.

9. Suggestions Of Future Work

The run time of the genetic programming system for the four point DCT varies. Point 0 and point 2 take one to two hours while point 1 and point 3 take about 6 hours. The result all showed that it is possible to improve fitness. Improvement to the result may be carried out by more sophisticated genetic operators or by increased run time, which extends the search space in the solution domain.

The four point DCT used in this project is not widely used in practice. Many international standards such as JPEG use 8 point DCTs. Future work may include an evolutionary algorithm for a fast 8 point DCT that is more useful.

The most important issue in the implementation of the DCT is its efficiency. This is the reason why research is still being carried out on the DCT. Genetic programming provides a new research approach for this field – not limited by rule based methods, but using an evolutionary algorithm. Theoretically, the most efficient 8 point DCT can be evolved with an evolutionary approach. To evolve a fast 8 point DCT, computation speed will be a key issue since the terminal set contains many more elements. As stated in chapter 5, CGPS can improve the evaluation speed significantly and increase the probability of success. Distributed Genetic Programming is another choice for improving search speed. It would

be possible to use a large parallel computer to evolve many niches in parallel with some form of limited interchange of genetic material.

References:

[1]. Angeline, P. J. (1996). Genetic Programming's continued evolution. In *Advances in genetic programming* Vol. 2. Angeline, P. J. & Kinnear, K. (Ed). (pp89-110). MIT Press.

[2]. Angeline, P. J. & Pollack, J. B. (1993). Evolutionary model acquisition. In *Proceedings of the second annual conference on evolutionary programming.* La Jolla, California.

[3]. Bäck, T. (1994). Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In Michaelewicz, Z. (Ed). Proceedings of the first IEEE world congress on evolutionary computation. IEEE world congress on computational intelligence. Vol. 1. (pp57-62). New York, NY. IEEE Press.

[4]. Bäck, T., Fogel, D. B., Michalewicz, Z. (Ed). (1997). *Handbook of evolutionary computation.* IOP publishing Ltd and Oxford University Press.

[5]. Bhaskaran, V. & Konstantinides, K. (1997). *Image and Video compression Standards. Algorithms and Architectures. Second Edition*. Kluwer Academic Publishers

[6]. Bot, M. C. J. & Langdon, W. B. (2000). Application of genetic programming to induction of linear classification trees. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., & Fogarty, T. C. (Ed). *Genetic Programming.* European conference. EuroGP'2000. Proceedings. (pp247-258). Springer.

[7]. Chen, W. H., Smith, C. H. & Fralick, S. C. (1977). A fast computational algorithm for the discrete cosine transform. IEEE. Trans On Comm. (pp1004-1009).

[8]. Clarke, R. J. (1985). *Transform Coding of Images*. Academic Press.

[9]. Coley, D. A. (1999). *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific Publishing Co.

[10]. D'haeseleer, P. (1994). Context preserving crossover in genetic programming. In Michaelewicz, Z. (Ed). *Proceedings of the first IEEE World congress on evolutionary computation.* IEEE World congress on computation intelligence. Vol. 1. (pp256-261). New York, NY. IEEE Press.

[11]. D'haeseleer, P. & Bluming, J. (1994). Effects of locality in individual and population evolution. In Kinnear, Jr. K. E. (Ed). *Advances in genetic programming.* Chapter 8. MIT Press.

[12]. Fukunaga, A. Stechert, A. & Mutz, D. (1998). A genome compiler for high performance genetic programming. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Gogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H.& Riolo, R. (Ed). , *Genetic Programming 1998: Proceedings of the Third Annual Conference* , pages 86-94, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[13]. Gathercole, C. & Ross, P. (1997). Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In *Genetic programming.* Proceedings of the second annual conferences. (pp119-127). Stanford University. CA. USA. Morgan Kaufmann.

[14]. Golubski, W. & Feuring, T. (1999). Evolving neural network structures by means of genetic programming. In Poli, R., Nordin, P., Langdon, W. B., Fogarty, T. C.(Ed). *Genetic Programming.* Second European Workshop, EuroGP'99. Proceedings. (pp211-220). Springer.

[15]. Hancook, P. J. B. (1994). An emparison of selection methods in evolutionary algorithms. In fogarty, T. C. (Ed). *Evolutionary computing*. AISB Workshop. Leeds. U.K. April 11-13 1994. (pp80-94). Springer.

[16]. Holland, J. H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* Ann Arbor: University of Michigan Press.

[17]. Hou, H. S. (1987). A fast recursive algorithm for computing the discrete cosine transform. IEEE. Tran. ASSP. (pp1455-1461).

[18]. Howard, D., Robert, S. C. & Brankin, R. (1999). Evolution of ship Detectors for Satellite SAR Imagery. In Poli, R., Nordin, P., Langdon, W. B., Fogarty, T. C.(Ed). *Genetic Programming.* Second European Workshop, EuroGP'99. Proceedings. (pp135-148). Springer.

[19]. Husband, P. (1994). Distributed co-evolutionary genetic algorithms for multi-criteria and multi-constraint optimisation. In Fogarty, T. C. (Ed). *Evolutionary Computing.* AISB Workshop. Springer.

[20]. ITO, T., IBA, H. & SATO, S. (1998). Non-destructive depth-dependent crossover for genetic programming. In Banzhaf, W., Poli, R., Schoenauer, M. & fogarty, T. C. (Ed). *genetic programming*. First European Workshop, EuroGP'98. Paris France April 1998. Proceedings. Springer.

[21]. Kinnear, jr. K. E. (Ed). (1994). *Advances in genetic programming.* MIT Press.

[22]. Kinnear, Jr. K. E. (1993). Evolving a sort: Lessons in genetic programming. In *1993 IEEE International conference on Neural networks.* San Francisco. Vol. 2 (881-888). Piscataway. NJ. IEEE Press.

[23]. Kinnear, Jr. K. E. (1993). Generality and difficulty in genetic programming: Evolving a sort. In Forrest, S. (Ed). *Proceedings of the fifth International conference on genetic algorithms.* (pp287-294). San Matco. CA. Morgan Kaufmann.
[24]. Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In *Eleventh international joint conference on artificial intelligence.* Proceedings Vol. 1. (pp768-774). San Matco CA. Morgan Kaufmann.

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

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

[27]. Langdon. W. B. (1998). *Genetic programming and data structures : genetic programming data structures = automatic programming.* Boston : Kluwer Academic Publishers.

[28]. Langdon, W. B. & Nordin, J. P. (2000). Seeding genetic programming populations. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., & Fogarty, T. C. (Ed). *Genetic Programming.* European conference. EuroGP'2000. Proceedings. Springer.

[29]. Lee, B. G. (1984). A new algorithm to compute the discrete cosine transform. IEEE. Tran. ASSP. (pp1243-1245).

[30]. Loeffler, C., Lightenberg, A. & Moschytz, G. (1989). Practical fast 1-D DCT algorithms with 11 multiplications. Proc. IEEE ICASSP Vol. 2, (pp988-991).

[31]. Maxwell, III. S. R. (1994). Experiments with a coroutine model for genetic programming. In Proceedings of the 1994 IEEE world congress on computational intelligence. Orlando, Florida. USA. IEEE Press.

[32]. Montana, D. J. (1994). Strongly typed genetic programming. BBN Technical Report #7866. Bolt Beranek and Newman, Inc.

[33]. Nordin, P. (1994). A compiling genetic programing system that directly manipulates the machine code. In Kinnear, Jr. E. J. (Ed). *Advances in genetic programming.* MIT Press.

[34]. Nordin, P. & Banzhaf, W. (1996). Programmatic compression of images and sound. In Koza, J. R., Goldberg, D. E., Fogel, D. B. & Riolo, R. L. (Ed). , *Genetic Programming 1996: Proceedings of the First Annual Conference* , pages 345-350, Stanford University, CA, USA, 28-31 July 1996. MIT Press.

[35]. Page, J., Poli, R. & Langdon, W. B. (1999). Smooth uniform crossover with smooth point mutation. In Poli, R., Nordin, P., Langdon, W. B. & Fogarty, T. C. (Ed). *Genetic*

Programming. Second European Workshop, EuroGP'99, Göteborg, Sweden, May 1999, Proceedings. Springer.

[36]. Potts, J. C., Giddens, T. D. & Yadav, S. B. The development and evaluation of an improved genetic algorithm base on migration and artificial selection. In *IEEE Transactions on systems. Man and Cybernetics.* Vol. 24. (pp73-86).

[37]. Poli, R. & Langdon, W. B. (1997). A new schema theory for genetic programming with one-point crossover and point mutation. In *Genetic programming.* Proceedings of the second annual conferences. (pp278-285). Stanford University. CA. USA. Morgan Kaufmann.

[38]. Poli, R. & Langdon, W. B. (1998). A Review of theoretical and experimental result on schemata in genetic programming. In Banzhaf, W., Poli, R., Schoenauer, M. & fogarty, T. C. (Ed). *Genetic Programming*. First European Workshop, EuroGP'98. Paris France April 1998. Proceedings. Springer.

[39]. Ramsey, C. & Grefenstette , J. J. (1993). Case-based initialization of genetic algorithms. In Forrest, S. (Ed). *Proceedings of the fifth international conference on genetic algorithms*. (pp84-91). San Matco. CA. Morgan Kaufmann.

[40]. Rao, K. R. & Yip, P. (1990). *Discrete Cosine Transform: Algorithms, Advantages, and Applications*. Academic Press

[41]. Sarafopoulos, A. (1999). Automatic generation of affine IFS & strongly typed genetic programming. In Poli, R., Nordin, P., Langdon, W. B. & Fogarty, T. C. (Ed). *Genetic Programming.* Second European Workshop, EuroGP'99, Göteborg, Sweden, May 1999, Proceedings. Springer.

[42]. Teller, A. (1994). Genetic programming, Index memory, the halting problem and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium.* (pp270-274). Pensacola, Florida, USA, IEEE Press.

[43]. Teller, A. (1996). Evolving programs: The co-evolution of intelligent recombination operator. In Angeline, P. J. & Kinnear, Jr. K. E. (Ed). *Advances in genetic programming 2.* MIT Press.

[44]. Vázquez, R. K. & Fleming, P. J. (2000). Use of genetic programming in the identification of rational model structures. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., & Fogarty, T. C. (Ed). *Genetic Programming.* European conference. EuroGP'2000. Proceedings. (pp181-192). Springer.

Appendix A: Test Data

The test data are listed below:

Source Data				Target Data			
Point 0	Point1	Point2	Point3	Point0	Point1	Point2	Point3
40	33	33	22	64	12	-1	5
26	40	36	26	64	1	-11	-2
43	26	33	22	62	12	3	10
29	43	22	36	65	1	0	-15
43	40	19	36	69	10	10	-11
36	33	15	26	55	11	7	-8
36	43	26	33	69	7	0	-9
29	29	19	4	41	19	-6	0
33	33	29	26	61	6	0	0
15	33	26	4	39	9	-19	-1
36	33	33	26	64	7	-1	3
22	29	26	12	45	7	-9	1
33	40	29	26	64	8	-4	-4
33	12	12	4	31	19	6	8
26	33	29	22	55	4	-6	-1
12	4	8	0	12	7	0	6
90	89	87	77	172	9	-3	2
70	76	66	47	130	18	-11	0
3	1	14	25	22	-17	6	3
17	16	29	31	47	-12	1	5
7	7	0	-8	3	12	-3	0
-2	0	-8	-2	-5	2	2	-4
10	9	11	11	21	0	0	1
10	6	2	-6	6	12	-1	2
-9	2	12	-5	0	-4	-13	5
3	-3	-8	-1	-4	4	6	-1
8	5	-6	1	4	8	5	-4
5	18	-1	-7	8	13	-8	-8
0	1	-9	-5	-6	6	1	-4
1	-1	-11	-6	-8	7	3	-4
-6	0	5	1	0	-5	-4	1
-85	-90	-110	-115	-199	25	0	-4

Table apx 1: The test data.

Appendix B: Comparison Of Target Data And Computed Data

The table below lists the computed data using the best solution and the target data given in table apx 1.

Computed Point 0	Target Point 0	Computed Point 1	Target Point 1	Computed Point 2	Target Point 2	Computed Point 3	Target Point 3
64	64	12	12	-1	-1	5	5
65	64	1	1	-11	-11	-2	-2
62	62	12	12	3	3	10	10
66	65	1	1	0	0	-14	-15
70	69	10	10	10	10	-11	-11
56	55	11	11	7	7	-8	-8
70	69	7	7	0	0	-9	-9
41	41	19	19	-7	-6	1	0
61	61	6	6	-1	0	0	0
39	39	9	9	-18	-19	-1	-1
64	64	7	7	-1	-1	3	3
45	45	8	7	-9	-9	1	1
65	64	8	8	-4	-4	-5	-4
31	31	19	19	6	6	8	8
55	55	4	4	-6	-6	-1	-1
12	12	7	7	0	0	6	6
172	172	9	9	-4	-3	2	2
130	130	18	18	-11	-11	0	0
22	22	-17	-17	6	6	2	3
47	47	-12	-12	2	1	5	5
3	3	12	12	-3	-3	0	0
-5	-5	2	2	2	2	-5	-4
21	21	-1	0	1	0	1	1
7	6	12	12	-1	-1	2	2
0	0	-5	-4	-12	-13	5	5
-4	-4	4	4	6	6	-2	-1
4	4	8	8	5	5	-5	-4
8	8	13	13	-9	-8	-8	-8
-6	-6	6	6	1	1	-4	-4
-8	-8	7	7	4	3	-4	-4
0	0	-5	-5	-4	-4	2	1
-200	-199	25	25	0	0	-4	-4

Table apx 2: Comparison of target data and computed data.

Appendix C: The Implementation Code

```
//macrodef.h : the macro definition
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <float.h>
#include <assert.h>
#include <memory.h>


#define FNAME_MAX 80
#define TEST_GROUP 32
#define DCTSIZE 4
#define AUXSIZE 1

#define POPULATION_SIZE  500
#define PCROSS            0.7
#define PMUT              0.02
#define MAX_PROG_SIZE    50
#define STACK_SIZE       50
#define LINE_WIDTH      300
#define THRESHOLD          0.01
#define TOURN_SIZE 4
#define MAX_TEST_POINTS   10
#define KILL_CHANCE   0.02
#define COPY_BEST_CHANCE  0.10
#define COFIT 50.0
#define PX_CNST 0.5
#define N_FUNCTIONS 4
#define N_TERMINALS 5

#define ADD 43
#define SUB 45
#define MULT 42
#define DIV 47

#define I0 97
#define I1 98
#define I2 99
#define I3 100

#define AUX0 48
#define AUX1 49
#define AUX2 50
#define AUX3 51


#define BIGFLOAT (1.0e15)
#define SMALLFLOAT (1.0e-15)
#define BOUNDF(f) (f==0? f: (f>0 ?((f)>BIGFLOAT ? BIGFLOAT :
((f)<SMALLFLOAT ? \
      SMALLFLOAT : (f))):((f)<-BIGFLOAT ? -BIGFLOAT : ((f)>-SMALLFLOAT ?
- SMALLFLOAT: (f))) ))
```

```
// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////////
////

#if
!defined(AFX_MAINFRM_H__AADEF38B_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
#define AFX_MAINFRM_H__AADEF38B_2C0E_11D5_810B_0C210FC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
      CMainFrame();
      DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CMainFrame)
      virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CMainFrame();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // control bar embedded members
      CStatusBar  m_wndStatusBar;
      CToolBar    m_wndToolBar;

// Generated message map functions
protected:
      //{{AFX_MSG(CMainFrame)
      afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
            // NOTE - the ClassWizard will add and remove member
functions here.
            //    DO NOT EDIT what you see in these blocks of generated
code!
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};
```

```
////////////////////////////////////////////////////////////////////////////
////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif //
!defined(AFX_MAINFRM_H__AADEF38B_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
```

```
// gpp.h : main header file for the GPP application
//

#if !defined(AFX_GPP_H__AADEF387_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
#define AFX_GPP_H__AADEF387_2C0E_11D5_810B_0C210FC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
      #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
////
// CGppApp:
// See gpp.cpp for the implementation of this class
//

class CGppApp : public CWinApp
{
public:
      CGppApp();

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CGppApp)
      public:
      virtual BOOL InitInstance();
      //}}AFX_VIRTUAL

// Implementation
      //{{AFX_MSG(CGppApp)
      afx_msg void OnAppAbout();
            // NOTE - the ClassWizard will add and remove member
functions here.
            //    DO NOT EDIT what you see in these blocks of generated
code !
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

float random_number();
/////////////////////////////////////////////////////////////////////////////
////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif //
!defined(AFX_GPP_H__AADEF387_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
```

```
// gppDoc.h : interface of the CGppDoc class
//
/////////////////////////////////////////////////////////////////////////////
////

#if
!defined(AFX_GPPDOC_H__AADEF38D_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
#define AFX_GPPDOC_H__AADEF38D_2C0E_11D5_810B_0C210FC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "Population.h"

class CGppDoc : public CDocument
{
protected: // create from serialization only
      CGppDoc();
      DECLARE_DYNCREATE(CGppDoc)

// Attributes
public:

// Operations
public:
      void ReadInputs();
      bool GetDrawControll(){return drawControll;}
      void SetDrawControll(bool dr){drawControll=dr;}

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CGppDoc)
      public:
      virtual BOOL OnNewDocument();
      virtual void Serialize(CArchive& ar);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CGppDoc();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

protected:
private:
      bool drawControll;

// Generated message map functions
protected:
      //{{AFX_MSG(CGppDoc)
            // NOTE - the ClassWizard will add and remove member
functions here.
            //    DO NOT EDIT what you see in these blocks of generated
code !
      //}}AFX_MSG
```

```
	DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif //
!defined(AFX_GPPDOC_H__AADEF38D_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
```

```
// gppView.h : interface of the CGppView class
//
/////////////////////////////////////////////////////////////////////////////
////

#if
!defined(AFX_GPPVIEW_H__AADEF38F_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
#define AFX_GPPVIEW_H__AADEF38F_2C0E_11D5_810B_0C210FC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CGppView : public CScrollView
{
protected: // create from serialization only
      CGppView();
      DECLARE_DYNCREATE(CGppView)

      void PushOperator(char op);
      void PushString(char str[]);
      void OutputProg(void);

// Attributes
public:
      CGppDoc* GetDocument();

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CGppView)
      public:
      virtual void OnDraw(CDC* pDC);  // overridden to draw this view
      virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
      protected:
      virtual void OnInitialUpdate(); // called first time after
construct
      virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
      virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
      virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CGppView();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
```

```
	//{{AFX_MSG(CGppView)
	afx_msg void OnRunStartrun();
	afx_msg void OnRunShowresult();
	afx_msg void OnRunStoprun();
	afx_msg void OnRunOutputprogram();
	afx_msg void OnRunReset();
	afx_msg void OnUpdateRunStartrun(CCmdUI* pCmdUI);
	afx_msg void OnUpdateRunShowresult(CCmdUI* pCmdUI);
	afx_msg void OnUpdateRunStoprun(CCmdUI* pCmdUI);
	afx_msg void OnUpdateRunOutputprogram(CCmdUI* pCmdUI);
	afx_msg void OnUpdateRunReset(CCmdUI* pCmdUI);
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
private:
	bool showResult;
	bool outputProgram;
	bool startRun;
	bool stopRun;
	bool reset;
};

#ifndef _DEBUG  // debug version in gppView.cpp
inline CGppDoc* CGppView::GetDocument()
   { return (CGppDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif //
!defined(AFX_GPPVIEW_H__AADEF38F_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
```

```
// Population.h: interface for the CPopulation class.
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_POPULATION_H__AADEF3B2_2C0E_11D5_810B_0C210FC10000__INCLUDED
_)
#define AFX_POPULATION_H__AADEF3B2_2C0E_11D5_810B_0C210FC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "Tree.h"

class CPopulation
{
public:
	CPopulation(int i);
	virtual ~CPopulation();
	void Mutate();
	CTree *GetTree() {return tree;}
	float GetFitness(){return fitness;}
	float TestFitness();
	void Initialise();
	void CopyPopulation(CPopulation *pop);

private:
	CTree *tree;
	float fitness;
	int position;

};

#endif //
!defined(AFX_POPULATION_H__AADEF3B2_2C0E_11D5_810B_0C210FC10000__INCLUDED
_)
```

```
// Tree.h: interface for the CTree class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_TREE_H__AADEF3B3_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
#define AFX_TREE_H__AADEF3B3_2C0E_11D5_810B_0C210FC10000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "macrodef.h"

class CTree
{
public:
	CTree();
	CTree(const CTree* aTree);
	virtual ~CTree();
	void RandomTree();
	float ProgramValue(int i);
	char *GetString(void) {return aString;}
	void SetString(char *s){strcpy(aString,s);}
	float GetAux(int i) {return frAux[i];}
	void SetAux(int i,float f) {frAux[i]=f;}

protected:
	void PushOperator (char op);
	void PushValue(float value);
	int RandomFunction(void);
	int RandomTerminal(void);
	void RandomAux();

private:
	char aString[MAX_PROG_SIZE+1];
	float frAux[AUXSIZE];

};

#endif //
!defined(AFX_TREE_H__AADEF3B3_2C0E_11D5_810B_0C210FC10000__INCLUDED_)
```

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "gpp.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
      //{{AFX_MSG_MAP(CMainFrame)
            // NOTE - the ClassWizard will add and remove mapping macros
here.
            //    DO NOT EDIT what you see in these blocks of generated
code !
      ON_WM_CREATE()
      //}}AFX_MSG_MAP
      // Global help commands
      ON_COMMAND(ID_HELP_FINDER, CFrameWnd::OnHelpFinder)
      ON_COMMAND(ID_HELP, CFrameWnd::OnHelp)
      ON_COMMAND(ID_CONTEXT_HELP, CFrameWnd::OnContextHelp)
      ON_COMMAND(ID_DEFAULT_HELP, CFrameWnd::OnHelpFinder)
END_MESSAGE_MAP()

static UINT indicators[] =
{
      ID_SEPARATOR,           // status line indicator
      ID_INDICATOR_CAPS,
      ID_INDICATOR_NUM,
      ID_INDICATOR_SCRL,
};

/////////////////////////////////////////////////////////////////////////////
////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
      // TODO: add member initialization code here

}

CMainFrame::~CMainFrame()
{
}
```

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
		return -1;

	if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_TOP
		| CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
		!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
	{
		TRACE0("Failed to create toolbar\n");
		return -1;	// fail to create
	}

	if (!m_wndStatusBar.Create(this) ||
		!m_wndStatusBar.SetIndicators(indicators,
		  sizeof(indicators)/sizeof(UINT)))
	{
		TRACE0("Failed to create status bar\n");
		return -1;	// fail to create
	}

	// TODO: Delete these three lines if you don't want the toolbar to
	//  be dockable
	m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
	EnableDocking(CBRS_ALIGN_ANY);
	DockControlBar(&m_wndToolBar);

	return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	if( !CFrameWnd::PreCreateWindow(cs) )
		return FALSE;
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs

	return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
	CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
	CFrameWnd::Dump(dc);
}
```

```
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
////
// CMainFrame message handlers
```

```
// gpp.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "gpp.h"

#include "MainFrm.h"
#include "gppDoc.h"
#include "gppView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
////
// CGppApp

BEGIN_MESSAGE_MAP(CGppApp, CWinApp)
      //{{AFX_MSG_MAP(CGppApp)
      ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
            // NOTE - the ClassWizard will add and remove mapping macros
here.
            //    DO NOT EDIT what you see in these blocks of generated
code!
      //}}AFX_MSG_MAP
      // Standard file based document commands
      ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
      ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
      // Standard print setup command
      ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
////
// CGppApp construction

CGppApp::CGppApp()
{
      // TODO: add construction code here,
      // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
////
// The one and only CGppApp object

CGppApp theApp;

/////////////////////////////////////////////////////////////////////////////
////
// CGppApp initialization

BOOL CGppApp::InitInstance()
{
```

```
	AfxEnableControlContainer();

	// Standard initialization
	// If you are not using these features and wish to reduce the size
	//  of your final executable, you should remove from the following
	//  the specific initialization routines you do not need.

#ifdef _AFXDLL
	Enable3dControls();			// Call this when using MFC in
a shared DLL
#else
	Enable3dControlsStatic();	// Call this when linking to MFC
statically
#endif

	// Change the registry key under which our settings are stored.
	// TODO: You should modify this string to be something appropriate
	// such as the name of your company or organization.
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));

	LoadStdProfileSettings();  // Load standard INI file options
(including MRU)

	// Register the application's document templates.  Document
templates
	//  serve as the connection between documents, frame windows and
views.

	CSingleDocTemplate* pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
		IDR_MAINFRAME,
		RUNTIME_CLASS(CGppDoc),
		RUNTIME_CLASS(CMainFrame),	// main SDI frame window
		RUNTIME_CLASS(CGppView));
	AddDocTemplate(pDocTemplate);

	// Parse command line for standard shell commands, DDE, file open
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;

	// The one and only window has been initialized, so show and update
it.
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}


////////////////////////////////////////////////////////////////////////
////
// CAboutDlg dialog used for App About
```

```
class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
	//}}AFX_DATA

	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CAboutDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
	//}}AFX_VIRTUAL

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
		// No message handlers
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CGppApp::OnAppAbout()
{
	CAboutDlg aboutDlg;
	aboutDlg.DoModal();
}


extern time_t now;

int intrnd (){
	static int seed=now;
	double const a    = 16807;      /* ie 7**5 */
```

```
        double const m     = 2147483647; /* ie 2**31-1 */
        double temp = seed * a;
        seed = (int) (temp - m * floor ( temp / m ));
        return seed;
}

float rand_0to1() { return (float)(intrnd()) / 2147483647.0; }

float random_number(){
        float r;
        for (; (r = rand_0to1() ) >= 1.0; );
        return ( r );
}
////////////////////////////////////////////////////////////////////////////
////
// CGppApp message handlers
```

```
// gppDoc.cpp : implementation of the CGppDoc class
//

#include "stdafx.h"
#include "gpp.h"

#include "gppDoc.h"

#include "Tree.h"
#include "macrodef.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
////
// CGppDoc

time_t now;
float input[TEST_GROUP][DCTSIZE];
float shouldbe[TEST_GROUP];

IMPLEMENT_DYNCREATE(CGppDoc, CDocument)

BEGIN_MESSAGE_MAP(CGppDoc, CDocument)
      //{{AFX_MSG_MAP(CGppDoc)
            // NOTE - the ClassWizard will add and remove mapping macros
here.
            //    DO NOT EDIT what you see in these blocks of generated
code!
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
////
// CGppDoc construction/destruction

CGppDoc::CGppDoc()
{
      drawControll=false;

}

CGppDoc::~CGppDoc()
{
}

BOOL CGppDoc::OnNewDocument()
{
      if (!CDocument::OnNewDocument())
            return FALSE;

      time(&now);
      ReadInputs();
```

```
	return TRUE;
}


/////////////////////////////////////////////////////////////////////////////
////
// CGppDoc serialization

void CGppDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
}

/////////////////////////////////////////////////////////////////////////////
////
// CGppDoc diagnostics

#ifdef _DEBUG
void CGppDoc::AssertValid() const
{
	CDocument::AssertValid();
}

void CGppDoc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
////
// CGppDoc commands

void CGppDoc::ReadInputs() {
	int i,j;
	FILE * fin;
	if((fin=fopen("problem.dat","r"))==NULL) {
		AfxMessageBox("can not open file");
	}
	for (i=0;i<TEST_GROUP;i++){
		for(j=0;j<DCTSIZE;j++){
			fscanf(fin,"%f",&input[i][j]);
		}
		fscanf(fin,"%f",&shouldbe[i]);
	}
	fclose(fin);
	return;
}
```

```
// gppView.cpp : implementation of the CGppView class
//

#include "stdafx.h"
#include "gpp.h"

#include "gppDoc.h"
#include "gppView.h"

#include "macrodef.h"
#include "Population.h"
#include "Tree.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
////
// CGppView

float bestResult[TEST_GROUP];
char bestString[MAX_PROG_SIZE+1];
float bestAux[AUXSIZE];
extern char opStack[STACK_SIZE];
char outputStack[STACK_SIZE][LINE_WIDTH];
extern int stack;

extern float tempResult[TEST_GROUP];
extern float shouldbe[TEST_GROUP];
float bestFitness;
CPopulation *bestProg=NULL;

volatile bool threadControll=true;

IMPLEMENT_DYNCREATE(CGppView, CScrollView)

BEGIN_MESSAGE_MAP(CGppView, CScrollView)
      //{{AFX_MSG_MAP(CGppView)
      ON_COMMAND(ID_RUN_STARTRUN, OnRunStartrun)
      ON_COMMAND(ID_RUN_SHOWRESULT, OnRunShowresult)
      ON_COMMAND(ID_RUN_STOPRUN, OnRunStoprun)
      ON_COMMAND(ID_RUN_OUTPUTPROGRAM, OnRunOutputprogram)
      ON_COMMAND(ID_RUN_RESET, OnRunReset)
      ON_UPDATE_COMMAND_UI(ID_RUN_STARTRUN, OnUpdateRunStartrun)
      ON_UPDATE_COMMAND_UI(ID_RUN_SHOWRESULT, OnUpdateRunShowresult)
      ON_UPDATE_COMMAND_UI(ID_RUN_STOPRUN, OnUpdateRunStoprun)
      ON_UPDATE_COMMAND_UI(ID_RUN_OUTPUTPROGRAM,
OnUpdateRunOutputprogram)
      ON_UPDATE_COMMAND_UI(ID_RUN_RESET, OnUpdateRunReset)
      //}}AFX_MSG_MAP
      // Standard printing commands
      ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
      ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
      ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
```

```
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
////
// CGppView construction/destruction

CGppView::CGppView()
{
	showResult=false;
	outputProgram=false;
	startRun=true;
	stopRun=false;
	reset=false;
}

CGppView::~CGppView()
{
}

BOOL CGppView::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs

	return CScrollView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
////
// CGppView drawing

void CGppView::OnDraw(CDC* pDC)
{
	CGppDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	char s[256];
	if (pDoc->GetDrawControll()) {
		sprintf(s,"Fitness so far: %f",bestFitness);
		pDC->TextOut(20,20,s);
		pDC->TextOut(20,50,"Shoutbe:   Result:");
		for (int i=0;i<TEST_GROUP;i++) {
			sprintf(s,"%8.2f
%8.2f",shouldbe[i],bestResult[i]);
			pDC->TextOut(20,70+i*18,s);
		}
		CSize docSize(100,(85+TEST_GROUP*18));
		CRect rect;
		GetClientRect(&rect);
		CSize pageSize(rect.right,rect.bottom);
		CSize lineSize(0,16);
		SetScrollSizes(MM_TEXT,docSize,pageSize,lineSize);
	}


}

void CGppView::OnInitialUpdate()
```

```
{
	CScrollView::OnInitialUpdate();

	CSize sizeTotal;
	// TODO: calculate the total size of this view
	sizeTotal.cx = sizeTotal.cy = 100;
	SetScrollSizes(MM_TEXT, sizeTotal);
}

/////////////////////////////////////////////////////////////////////////////
////
// CGppView printing

BOOL CGppView::OnPreparePrinting(CPrintInfo* pInfo)
{
	// default preparation
	return DoPreparePrinting(pInfo);
}

void CGppView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
	// TODO: add extra initialization before printing
}

void CGppView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
	// TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////////
////
// CGppView diagnostics

#ifdef _DEBUG
void CGppView::AssertValid() const
{
	CScrollView::AssertValid();
}

void CGppView::Dump(CDumpContext& dc) const
{
	CScrollView::Dump(dc);
}

CGppDoc* CGppView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CGppDoc)));
	return (CGppDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
////
// CGppView message handlers

void UpdateBest(float fit,CPopulation *pop) {
	CTree *tr;
```

```
	int i;
	if(fit > bestFitness) {
		bestFitness=fit;
		bestProg=pop;
		tr=bestProg->GetTree();
		strcpy(bestString,tr->GetString());
		for(i=0;i<TEST_GROUP;i++)
			bestResult[i]=tempResult[i];
		for(i=0;i<AUXSIZE;i++)
			bestAux[i]=tr->GetAux(i);
	}
	return;
}

void InitialisePop(CPopulation *pop[]){
	int i;
	float fit;

	pop[0]=new CPopulation(0);
	pop[0]->Initialise();
	bestProg=pop[0];
	for (i=1;i<POPULATION_SIZE;i++) {
		pop[i]=new CPopulation(i);
		pop[i]->Initialise();
		fit=pop[i]->TestFitness();
		UpdateBest(fit,pop[i]);
	}

	return;
}
int random_point ( char string[] ) {
  return  ((int) (random_number() * (float) (strlen(string) )  ));
}

int matching_point ( char string[], int subtree ) {


	int i;
	int varCount;

	if ( string[subtree] == '\0' )
		varCount = 0;
	else
		varCount = 1;
	for (i = subtree; varCount > 0; i++ ){
		switch (string [i]){
			case ADD:
			case SUB:
			case MULT:
			case DIV:
				varCount++;
				break;

			case '\0':

				assert (1 == 0);
				break;
```

```
                default :
                        varCount--;
                        break;
            }
        }
        return ( i );
}

int Connect(char output[],int outsize,char buff1[],int end1,char
buff2[],int end2,
                char buff3[], int end3){
        if (( end1 + end2 + end3 ) >= outsize ) return ( 1 );
        memcpy ( output,              buff1, end1 );
        memcpy ( &output[end1],       buff2, end2 );
        memcpy ( &output[end1+end2], buff3, end3 );
        output [end1+end2+end3] = '\0';
        return ( 0 );
}

void Crossover(CPopulation *pop0, CPopulation *pop1, CPopulation *pop2,
CPopulation *pop3){
        int mum_end1,mum_start2, mum_end2;
        int dad_end1,dad_start2, dad_end2;
        char *mum, *dad, *child1, *child2;
        CTree *tr;
        tr=pop0->GetTree();
        mum=tr->GetString();
        tr=pop1->GetTree();
        dad=tr->GetString();
        tr=pop2->GetTree();
        child1=tr->GetString();
        tr=pop3->GetTree();
        child2=tr->GetString();
        do {
                mum_end1   = random_point (mum);
                mum_start2 = matching_point (mum, mum_end1 );
                mum_end2   = strlen (mum);
                dad_end1   = random_point (dad);
                dad_start2 = matching_point (dad, dad_end1 );
                dad_end2   = strlen (dad);
        } while (Connect (
           child1, MAX_PROG_SIZE+1,
           mum, mum_end1,
           &dad[dad_end1], dad_start2 - dad_end1,
           &mum[mum_start2], mum_end2 - mum_start2 ) &&
           Connect (
           child2, MAX_PROG_SIZE+1,
           dad, dad_end1,
           &mum[mum_end1], mum_start2 - mum_end1,
           &dad[dad_start2], dad_end2 - dad_start2 ));
    return;
}

UINT ThreadProc(LPVOID param) {
```

```
	int p[TOURN_SIZE];
	int i,j,k,first;
	int generation=1;
	float fit;
	CPopulation *arPop[POPULATION_SIZE];

	InitialisePop(arPop);

	while(threadControll) {
		for(i=0;i<POPULATION_SIZE;i++) {
			if((arPop[i]!=bestProg) &&
(random_number()<KILL_CHANCE)) {
				arPop[i]->Initialise();
				fit=arPop[i]->TestFitness();
				UpdateBest(fit,arPop[i]);
			}
		}

		first=(int)(random_number()*POPULATION_SIZE);

		for(i=0;i<TOURN_SIZE;i++){
			p[i]=(first+i)%POPULATION_SIZE;
		}
		for (i=TOURN_SIZE-1;i>=0;i--){
			for(j=0;j<i;j++){
				if ((arPop[p[j]]->GetFitness()) < (arPop[p[j+1]]-
>GetFitness())) {
					k=p[j];
					p[j]=p[j+1];
					p[j+1]=k;
				}
			}
		}

		if((arPop[p[2]]!=bestProg) && (arPop[p[3]]!=bestProg)){
			if(random_number()<COPY_BEST_CHANCE) {
				arPop[p[2]]->CopyPopulation(arPop[p[0]]);
				arPop[p[3]]->CopyPopulation(arPop[p[0]]);
			} else {

	Crossover(arPop[p[0]],arPop[p[1]],arPop[p[2]],arPop[p[3]]);
				fit=arPop[p[2]]->TestFitness();
				UpdateBest(fit,arPop[p[2]]);
				fit=arPop[p[3]]->TestFitness();
				UpdateBest(fit,arPop[p[3]]);
			}
			if (!(generation%50)) {
				arPop[p[2]]->Mutate();
				arPop[p[3]]->Mutate();
				fit=arPop[p[2]]->TestFitness();
				UpdateBest(fit,arPop[p[2]]);
				fit=arPop[p[3]]->TestFitness();
				UpdateBest(fit,arPop[p[3]]);
			}
		}
		generation++;
	}
```

```
	return 0;
}


void CGppView::PushOperator(char op) {
	opStack [stack++] = op;
	return;
}


void CGppView::PushString(char str[]) {
	char first[LINE_WIDTH];
	char result[LINE_WIDTH];

	if ( (stack <= 0) || (opStack[stack-1] != 0) ) {
		opStack [ stack ]  = 0;
		strcpy (outputStack[stack], str);
		stack++;
	} else {
		--stack;
		strcpy (first, outputStack[stack]);
		switch (opStack[--stack]) {
	  case ADD:
			sprintf (result, "%s+%s", first, str);
		break;

	  case SUB:
			sprintf (result, "(%s)-(%s)", first, str);
		break;

	  case MULT:
			sprintf (result, "(%s)*(%s)", first, str);
		break;

	  case DIV:
			sprintf (result, "div(%s,%s)", first, str);
		break;

	  default:
			assert (1 == 0 );
		break;
		}

		PushString ( result );
	}

}


void CGppView::OnRunStartrun()
{
	HWND hWnd=GetSafeHwnd();
	AfxBeginThread(ThreadProc,hWnd,THREAD_PRIORITY_NORMAL);
	startRun=false;
	showResult=true;
```

```
	stopRun=true;
	return;
}

void CGppView::OnRunShowresult()
{
	CGppDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	pDoc->SetDrawControll(true);
	Invalidate();
}

void CGppView::OnRunStoprun()
{
	threadControll=false;
	outputProgram=true;
	stopRun=false;
}

void CGppView::OnRunOutputprogram()
{
	char buff [] = " ";
	int i;
	FILE *fout;

	stack = 0;

	for ( i = 0; bestString[i] != 0; i++ ){
		switch (bestString [i]){
		case ADD:
		case SUB:
		case MULT:
		case DIV:
			PushOperator(bestString[i]);
		break;

		default:
			buff [ 0 ] = bestString [i];
			PushString( buff );
			break;
		}
	}

	assert (stack == 1);

	if ((fout=fopen("dct.cpp","w"))==NULL){
		AfxMessageBox("Can not open file \"dct.cpp\" ");
		_ASSERT(1==0);
	}

	fprintf (fout, "\nfloat div ( float top, float bot ) {\n");
	fprintf (fout, "  if ((bot>-0.001)&&(bot<0.001))\n");
	fprintf (fout, "		return (1.0);\n");
	fprintf (fout, "  else\n");
	fprintf (fout, "		return ( top/bot );\n");
	fprintf (fout, "}\n");
```

```
	fprintf (fout, "\nfloat dct( int a,int b,int c,int d ) {\n");
	fprintf (fout, "  return ");
	i=0;
	stack--;
	while (outputStack[stack][i]!='\0'){
		switch (outputStack[stack][i]) {
		case AUX0:
			fprintf(fout,"%f",bestAux[0]);
			break;
		case AUX1:
			fprintf(fout,"%f",bestAux[1]);
			break;
		case AUX2:
			fprintf(fout,"%f",bestAux[2]);
			break;
		case AUX3:
			fprintf(fout,"%f",bestAux[3]);
			break;
		default:
			fprintf(fout,"%c",outputStack[stack][i]);
			break;
		}
		i++;
	}
	fprintf(fout,";\n}\n");
	fclose (fout);
	showResult=false;
	outputProgram=false;
	reset=true;
	return;
}

void CGppView::OnRunReset()
{
	bestFitness=0.0;
	reset=false;
	startRun=true;
	showResult=false;
	outputProgram=false;
	stopRun=false;
	return;
}

void CGppView::OnUpdateRunStartrun(CCmdUI* pCmdUI)
{
	pCmdUI->Enable(startRun);
}

void CGppView::OnUpdateRunShowresult(CCmdUI* pCmdUI)
{
	pCmdUI->Enable(showResult);
}

void CGppView::OnUpdateRunStoprun(CCmdUI* pCmdUI)
{
	pCmdUI->Enable(stopRun);
}
```

```
void CGppView::OnUpdateRunOutputprogram(CCmdUI* pCmdUI)
{
	pCmdUI->Enable(outputProgram);
}

void CGppView::OnUpdateRunReset(CCmdUI* pCmdUI)
{
	pCmdUI->Enable(reset);
}
```

```
// Population.cpp: implementation of the CPopulation class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "gpp.h"
#include "Population.h"

#include "macrodef.h"
#include "Tree.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
float tempResult[TEST_GROUP];
extern float shouldbe[TEST_GROUP];

CPopulation::CPopulation(int i)
{
      tree=new CTree();
      fitness=0.0;
      position=i;
}

CPopulation::~CPopulation()
{
      delete tree;
}

void CPopulation::Initialise() {

      tree->RandomTree();

      return;
}

float CPopulation::TestFitness() {
      int i;
      float fTotalError = 0.0;

      for(i=0;i<TEST_GROUP;i++) {
            tempResult[i]=tree->ProgramValue(i);
            fTotalError += ((float)shouldbe[i]-
tempResult[i])*((float)shouldbe[i]-tempResult[i]);
            if (fTotalError>1000) {
                  fitness=0.0;
                  return 0.0;
            }
      }
      fitness=(float)COFIT/(fTotalError+(float)COFIT);
      return fitness;
```

```
}


void CPopulation::Mutate() {
      int i=(int)(random_number()*AUXSIZE);

      if (random_number()>0.5)
            tree->SetAux(i,(float)(1.05*tree->GetAux(i)));
      else
            tree->SetAux(i,(float)(0.95*tree->GetAux(i)));
      return;
}



void CPopulation::CopyPopulation(CPopulation* pop){
      char *s;
      CTree *tr;
      float f;
      tr=pop->GetTree();
      s=tr->GetString();
      tree->SetString(s);
      for(int i=0;i<AUXSIZE;i++) {
            f=tr->GetAux(i);
            tree->SetAux(i,f);
      }
      fitness=pop->fitness;
}
```

```
// Tree.cpp: implementation of the CTree class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "gpp.h"
#include "Tree.h"

#include "macrodef.h"
#include "Population.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

int stack;
float valueStack[STACK_SIZE];
char opStack[STACK_SIZE];  //unsigned !

extern float input[TEST_GROUP][DCTSIZE];
extern float shouldbe[TEST_GROUP];

CTree::CTree()
{

}

CTree::CTree(const CTree* aTree) {
      int i;
      strcpy(aString,aTree->aString);
      for (i=0;i<AUXSIZE;i++) {
            frAux[i]=aTree->frAux[i];
      }
}

CTree::~CTree()
{

}

int CTree::RandomFunction(void)      {
      int function [ N_FUNCTIONS ] = { ADD,SUB,MULT,DIV};
      return ( function [ (int) (random_number() * N_FUNCTIONS)] );
}


int CTree::RandomTerminal(void) {
      int term_buf[N_TERMINALS]={I0,I1,I2,I3,AUX0};
      return ( term_buf[(int)(random_number()*N_TERMINALS)] );
}
```

```
void CTree::RandomAux(){
      int i;

      for(i=0;i<AUXSIZE;i++){
            frAux[i]=random_number();
      }
      return;
}

void CTree::RandomTree() {
      int arguments = 1;
      int i;

      RandomAux();
      while(1){
            for ( i = 0; (arguments > 0) && (i < MAX_PROG_SIZE) ; i++ ){
                  if( random_number() >
(float)(arguments*arguments+1)/(float)(MAX_PROG_SIZE-i) ) {
                        aString[i] =(char) RandomFunction();
                        arguments++;
                  } else {
                        aString[i] =(char) RandomTerminal();
                        arguments--;
                  }
            }
            if(!arguments) break;
      }
      aString[i]='\0';
      assert ( arguments == 0 ); /* some thing wrong? */
      return;
}



void CTree::PushOperator(char op) {
      opStack [stack++]       = op;
      valueStack [stack]      = 0.0;
      return;
}


void CTree::PushValue ( float value ) {
      float a;
      float result;


      if((stack<=0)||(opStack[stack-1]!=0)) {
            opStack[stack]=0;
            valueStack[stack++]=value;
      } else {
            a = valueStack[--stack];
            switch (opStack[--stack]) {
            case ADD:
                  result = a + value;
```

```
                break;

            case SUB:
                result = a - value;
                break;

            case MULT:
                result =(float) BOUNDF(a * value);
                break;

            case DIV:
                if ((value > -0.001)&&(value<0.001))
                    result = 1.0;
                else
                    result = a / value;
                break;

            default:
                assert (1 == 0 );
                break;
            }
            PushValue ( result );
        }
        return;
}

float CTree::ProgramValue(int i) {
        int k;

        stack = 0;
        for ( k = 0; aString[k] !='\0'; k++ ) {
            switch (aString[k]){
            case ADD:
            case SUB:
            case MULT:
            case DIV:
                PushOperator(aString[k]);
                break;

            case I0:
                PushValue(input[i][0]);
                break;
            case I1:
                PushValue(input[i][1]);
                break;
            case I2:
                PushValue(input[i][2]);
                break;
            case I3:
                PushValue(input[i][3]);
                break;

            case AUX0:
                PushValue(frAux[0]);
                break;
```

```
            default:
                  assert(1==0);  /* something wrong? */
                  break;
            }
      }
      assert (stack == 1);
      return (valueStack[--stack]);
}
```