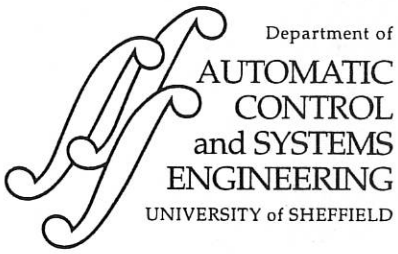
Monograph:
Swain, Anjan and Zalzala, A. (1999) An Object Oriented Approach to Genetic Programming in C ++. Research Report. ACSE Research Report 740 . Department of Automatic Control and Systems Engineering



AN OBJECT ORIENTED APPROACH TO GENETIC PROGRAMMING IN C++

Anjan Swain and Ali Zalzala

Robotics Research Group,
Department of Automatic Control and Systems Engineering
The University of Sheffield
Mappin Street, Sheffield S1 3JD, United Kingdom

Research Report #740
January 1999

RRG Tel : +44 (0)114 2225250
Fax : +44 (0)114 2731729
EMail : rrg@sheffield.ac.uk
Robotics Research Group

An Object Oriented Approach to Genetic Programming in C++

A.K. Swain

Department of Automatic Control and Systems Engineering, University of Sheffield, Mappin Street, Sheffield S1 3JD, UK

Email: cop97aks@sheffield.ac.uk

A.M.S. Zalzala[1]

Department of Computing & Electrical Engineering, Heriot-Watt University, Edinburgh, EH14 4AS, UK

Email: a.zalzala@hw.ac.uk

ABSTRACT

This paper discusses an object-oriented approach for the implementation of genetic programming (GP) in C++ programming language. Basic GP data structures have been formulated and their implementation aspects have been discussed.

Keywords - Genetic programming, automatic programming, object oriented programming, C++.

1. INTRODUCTION

Genetic programming (GP) is a major variation of GA to automatic generation of computer programs, which are evolved to solve or approximately solve problems [1-4]. The evolving individuals are themselves computer programs represented usually by tree structures. In tree representation for GP, the individual programs are represented as rooted trees with ordered branches. Each tree is composed of functions as internal nodes and terminals as leaf nodes of the problem. To solve a particular problem, it is required to define a priori a *terminal set* **T** and a *function set* F, such that the selected functions and terminals will be useful to get the solution for the problem. Each function in the function set must satisfy the *closure* property by accepting gracefully as arguments the return value of any other function and any data type in the terminal set.

GP starts with an initial population of randomly generated tree structured computer programs. A fitness is assigned to each individual program that evaluates the performance of the individual on a suitable set of test cases. Individuals are selected based on their fitness and the selected individuals are allowed to survive. Then, crossover creates offspring by exchanging subtrees between two parent trees at selected random crossover points. An example of this crossover mechanism has been illustrated in Fig. 1. Other genetic operators such as mutation, permutation, editing and a define-building block operation are rarely used [1].

A steady state GP algorithm used in this implementation is shown in Fig. 2.

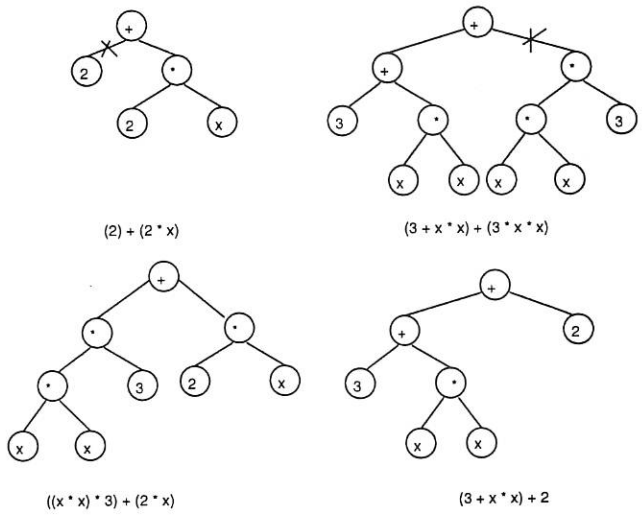


Fig.1. Sub-tree crossover in GP. Crossover sites are shown with cross marks on parent trees

Steady State GP Algorithm

Step - 1. Initialise the population

Step - 2. Select the winner or winners from a randomly sampled subset of population (competitors) using some selection algorithm.

Step - 3. Procreate offspring by modifying the selected winners with genetic operators.

Step - 4. Replace the worst individual with the offspring.

Step - 5. If the termination criterion not satisfied Then repeat Steps 2-5. else output the single best individual of the entire run.

Fig. 2. The structure of a steady state GP algorithm.

[1] Corresponding author.

The implementation of tree structured individuals and their subsequent manipulation involves the complexities of their dynamic creation and evaluation. To generate syntactically correct programs, programming languages like LISP, C, or C++ are used. All the programs are represented internally as parse trees. Koza selected LISP programming language which has the property that the functions can easily be visualised as trees with syntax in prefix form, and the syntax is preserved by restricting the language to suit the problem with appropriate number of constants, variables, statements, functions, and operators. Recently, GP software developers are using mostly compiled languages such as C and C++, because of their faster speed to evaluate the individual fitness functions, where GP spends most of their time. To date very few publications [5-8] discussed the implementation aspects of GP in C++. In this paper, an object oriented approach using C++ programming language has been described. Here, the variable length tree structures are implemented using pointers.

2. GENETIC PROGRAMMING DEVELOPMENT

Essentially, GP manipulates a population of variable length tree structured computer programs using genetic operators. Where each computer program, known as an individual, consists of functions and terminals. These functions, terminals and their attributes are imbued inside tree nodes. This can be interpreted as that each individual computer program tree consists of nodes, and a collection of different types of such trees constitutes a population. Then genetic operators are applied on this population to evolve better individuals with the progress of the generation. This hierarchy is shown in Fig.3. This makes it clear that nodes serve as the key component of a GP system, around which the entire GP system is developed. So our GP system development starts at this point treating nodes as its stepping stones. As already discussed, nodes may either be a function node or a terminal node. This feature is illustrated in Fig. 4. The classification of nodes in this manner sprouts our object oriented design philosophy. As a first step towards object oriented design, the base class node is formed, and from this both function and terminal nodes are inherited. In the next step, this base node class has been converted to an abstract class with the inclusion of pure virtual node accessing functions. In Fig. 4, the dashed box represents an abstract base class and the lines indicate that the classes are publicly derived from its base class. The detailed data structure for these are described in the next section.

3. GENETIC PROGRAMMING DATA STRUCTURES AND IMPLEMENTATION

The implementation of GP in C++ accommodates the following basic data structures:

- System input for both function nodes and terminal nodes
- Node features of each function node and terminal node
- Individual program tree
- Population of individuals

All these data structures are described in the following subsections.

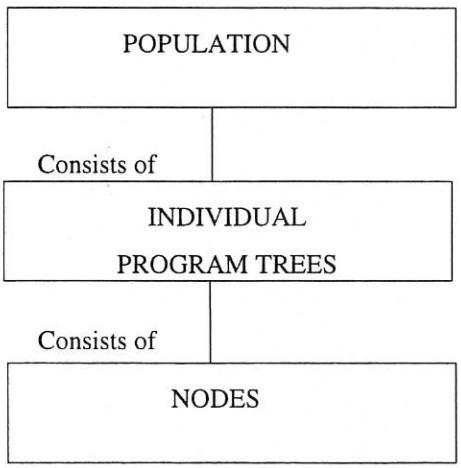


Fig. 3. Genetic programming structure

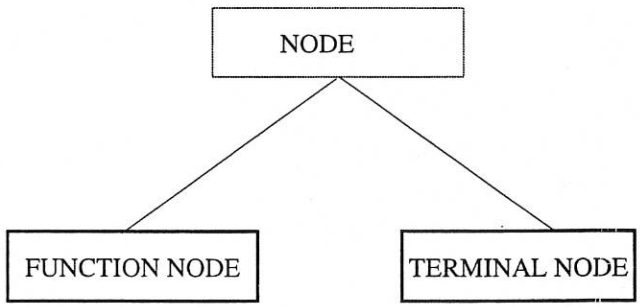


Fig. 4. Node hierarchy

3.1 SYSTEM INPUT DATA

In general a node can completely be defined with the following four primary attributes:

- Types of nodes, i.e., function node or terminal node
- Arity of the node
- Node objective
- Value of the node

These primary attributes can not completely define a particular node in a linked list of tree nodes. Hence, for the identification of a specific node in a tree, it needs to add some extra attributes, categorised as secondary attributes, those are:

- Node number, and

- Level at which the node is placed in a tree, i.e., depth information.

These secondary attributes are useful for implementing different genetic operators such as crossover and mutation, and evaluating a tree. Fig. 5 illustrates the implementation of these node attributes. The character variable Name in the structure implementation of Data assigns a particular single character symbol to a node. Whereas, the depth defines the position of a node in a tree.

A database has been created to store node functionality, i.e., application specific function set and terminal set. This has been shown in Fig. 6. Here, function and terminal sets are stored in arrays of size FUNCSIZE and TERMSIZE, respectively.

```
structure Data
{
// Primary Attributes
int nodeType;
 int Arity;
char Name;
float Value;
// Secondary Attributes
unsigned int nodeNumber;
unsigned int depth;
}; // end of Data
```

Fig. 5. Representation of Node Attributes

```
class FTValues

{
public:
void DefineFunctionSet();
void DefineTerminalSet();
void SetNumberofFunctions();
void SetNumberofTerminals();
void SetTotal();
private:
FunctionNode FArray[FUNCSIZE];
TerminalNode TArray[TERMSIZE];
int numberofFunctions;
int numberofInputs, total;
}; // end of class FTValues
```

Fig. 6. A Class Structure Representing Function and Terminal Sets

3.2 NODE DATA STRUCTURE

In this implementation, each node has three fields, LeftmostChild, Label (data field) and RightSibling. This is shown in Fig.7. LeftmostChild pointer points to the oldest son of a given parent node and RightSibling pointer points to the brothers of the oldest son, i.e., LeftmostChild and RightSibling pointers keep track of all the sons of a given node. Label holds all the attributes of a given node, so it is of type structure Data, as illustrated in Fig. 5. Two pure virtual functions have been added to acquire the node information. The implementation of node class is shown in Fig. 8.

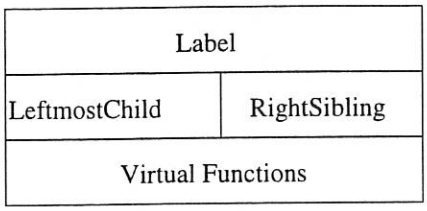


Fig. 7. Node Data Structure

```
typedef Node*  PtrType;
Class Node
{
public:
virtual void SetData() = 0;
virtual Data GetData() = 0;
void SetDepth(unsigned int);
void SetNodeNumber(unsigned int);
unsigned int GetDepth();
unsigned int GetNodeNumber();
Data Label;
PtrType  RightSibling;
PtrType  LeftmostChild;
} // end of Class Node
```

Fig. 8. Node Class Representation

Function nodes and terminal nodes are publicly derived from class node. Function nodes have child nodes, whose number depends on the type of the function. This makes the program to accept any type of user defined functions. On the other hand, the terminal nodes consist of inputs and constants. When a particular terminal node is selected to serve as an input node, its value changes during run time training phase to accept different input test samples. Whereas, if a terminal node is chosen to be a constant then a random number is substituted in its value field during its construction, and its value usually remains constant throughout the test phase. In some cases these constants may change with application of special type of mutation operator [9]. Terminal nodes do not accept any input. The class structures for function and terminal node are shown in Fig. 9 and Fig. 10, respectively. The member functions SetData() and GetData() are defined to set and retrieve all the input values.

```
class FunctionNode : public Node
        {
        public:
        FunctionNode(); // Constructor
        ~FunctionNode(); // Destructor
        void SetData();
        Data GetData();
        }; // end class FunctionNode
```

Fig. 9. FunctionNode Class Representation

```
class TerminalNode : public Node
        {
        public:
        TerminalNode(); // Constructor
        ~TerminalNode(); // Destructor
        void SetData();
        Data GetData();
        }; // end class Terminal Node
```

Fig. 10. TerminalNode Class Representation

3.3 THE TREE CLASS

The individual computer program trees implemented here are rooted general trees [10-12]. The LeftmostChild pointer of each node points to the oldest or the first child of that node and the RightSibling pointer points to the nodes next sibling. Each tree keeps note of its root node pointer only. Once the root pointer is defined, the entire tree can be constructed recursively by linking through each nodes leftmost child and right sibling pointer fields. This has been described in Fig. 11 in the routine CreateTree(..), which is a member function of class tree. Other important membership functions of class tree used to create a complete tree are MakeList(.), SetChildren(.) and AssignDepth(). The MakeList(.) member function creates a linked list of children of a node and SetChildren(.) member function sets this linked list of children to their parent node. AssignDepth() member function helps to create a tree of prespecified depth. The tree class incorporates all tree manipulation functions, such as evaluating a tree, assigning node numbers and depths to each node, and displaying trees and their parameters. The tree class has been illustrated in Fig. 12.

```
Void Tree :: CreateTree(PtrType CNode, FTValues& Ft))
{
        if(CNode)
        {
        int dummyVar = CNode->Label.Arity;
        if(dummyVar != 0)
        {
        depthCount = CNode->Label.GetDepth();
        MakeList(dummyVar, Ft);
        SetChildren(CNode);
        AssignDepth();
        } // end inner if
        CreateTree(CNode->LeftmostChild, Ft);
        CreateTree(CNode->RightSibling, Ft);
        } // end outer if
} // end CreateTree
```

Fig. 11. Creation of Tree

```
class Tree
        {

        friend class Population;
        public:
        Tree(); // Constructor
        Tree(Tree* ); // Copy Constructor
        ~Tree(); // Destructor
        // Create Tree
        void SetRoot(FTValues&); // sets Root Node
        void CreateTree(PtrType, FTValues&);
        void MakeList(int, FTValues&);
        void SetChildren(PtrType);
        // Evaluation
        void Evaluation();
        void FitnessEvaluation(PtrType);
        // Displaying and Acquiring Tree Parameters
        private:
        PtrType Root;
        PtrType StartPtr, CurrentPtr, EndPtr; // Creates
Linked List
        FunctionNode *FList; // Holds Functions
        TerminalNode *TList; // Holds Terminals
        }; // end Class Tree
```

Fig. 12. Tree Class Representation

```
class Population
        {
        public:
        // Population Initialisation
        Population(); // Constructor
        ~Population(); // Destructor
        void Initialise(FTValues&);
        // Genetic Operators
        // Selection
        void Selection();
        Tree* FindBest();
        int FindWorst();
        // Crossover
        void Crossover();
```

```
// Mutation
void Mutation();
// Book Keeping Functions
private:
Tree Pool[PopulationSize];
Tree* Parent[2];
unsigned int crossNode;
// end Population Class
```

Fig. 13. Population Class

3.4 POPULATION CLASS

The final class in the discussion of GP is population class. Until now, we have seen how to develop a tree. In all evolutionary computation methods, a population of candidate solutions is maintained, which then gets manipulated by the use of genetic operators. Being the size of the population is prespecified, so here a fixed array of tree objects are created. Then these trees are manipulated using crossover and mutation operators to procreate offspring to progress the generation. The population class holds all the tree manipulation functions and their book keeping jobs, which is shown in Fig. 13. The crossover operation swaps two randomly selected subtrees of two parents to produce two offspring. The crossover membership function implementation is shown in Fig. 14. All the classes are defined to be hierarchically friends (not shown in the respective class representations).

```
Void Population :: Crossover()
{
unsigned int dummyDepth1, dummyDepth2,
        crossPoint1, crossPoint2;
PtrType crossNode1, crossNode2, parentNode1,
        parentNode2, dummyPtr;
Boolean FLAG1, FLAG2;
Tree dad, mum;
do
{
dad.Root = dad.CopyTree(Parent[0]);
mum.Root = mum.CopyTree(Parent[1]);
crossNode1 = SelectNode(Parent[0]); // Select a Random
Node
crossPoint1 = crossNode1->GetNodeNumber();
// Get the Node to which crossNode1 is connected
parentNode1 = GetCrossNode(Parent[0]->Root,
crossPoint1);
crossNode2 = SelectNode(Parent[1]); // Select a Random
Node
crossPoint2 = crossNode2->GetNodeNumber();
parentNode2 = GetCrossNode(Parent[1]->Root,
crossPoint2);
if((parentNode1->LeftmostChild)&&((parentNode1-
>GetNodeNumber()+1) == crossPoint1))
FLAG1 = TRUE;
else FLAG1 = FALSE;
if((parentNode2->LeftmostChild)&&((parentNode2-
>GetNodeNumber()+1) == crossPoint2))
FLAG2 = TRUE;
else FLAG2 = FALSE;
if((FLAG1 == TRUE)&&(FLAG2==TRUE))
{
dummyPtr = crossNode1->RightSibling;
crossNode1->RightSibling = crossNode2->RightSibling;
crossNode2->RightSibling = dummyPtr;
dummyPtr = parentNode1->LeftmostChild;
parentNode1->LeftmostChild = parentNode2-
>LeftmostChild;
parentNode2->LeftmostChild = dummyPtr;
} // end if
if((FLAG1 == TRUE)&&(FLAG2==FALSE))
{
crossNode1->RightSibling = crossNode2->RightSibling;
crossNode2->RightSibling = NULL;
dummyPtr = parentNode1->LeftmostChild;
parentNode1->LeftmostChild = parentNode2-
>RightSibling;
parentNode2->RightSibling = dummyPtr;
} // end if
if((FLAG1 == FALSE)&&(FLAG2==TRUE))
{
crossNode1->RightSibling = crossNode2->RightSibling;
crossNode2->RightSibling = NULL;
dummyPtr = parentNode1->RightSibling;
parentNode1->RightSibling = parentNode2-
>LeftmostChild;
parentNode2->LeftmostChild = dummyPtr;
} // end if
if((FLAG1 == FALSE)&&(FLAG2==FALSE))
{
dummyPtr = crossNode1->RightSibling;
parentNode1->RightSibling = parentNode2-
>RightSibling;
parentNode2->RightSibling = dummyPtr;
} // end if
// Renumber the Offspring
// Evaluate the Offspring
// Calculate the New Depth
dummyDepth1 = Parent[0]->GetMaxDepth();
dummyDepth2 = Parent[1]->GetMaxDepth();
} while((dummyDepth1 > maxCrossDepth) ||
(dummyDepth2 > maxCrossDepth));
// Replace two worst offspring by dad and mum
} // end Crossover
```

Fig. 14. Crossover Operation

4. AN EXAMPLE OF SYMBOLIC REGRESSION

As an example of GP, consider the problem of symbolic regression. For this task a set of data points have been

provided, and the job is to find the underlying functional relationship in symbolic terms.

Suppose given a set of 10 data points in terms of x and y co-ordinates of a simple function $y = x^2/2$ [2].

The preliminary preparatory steps decided are:

- Terminal set - **T** = {x, R}

where variable x is set a particular value during training and R is an integer random number between -5 and 5, i.e., whenever the terminal element R is selected as a node then at that point a random integer number between -5 and 5 is inserted into the tree as the terminal node value.

- Function set - F = {+, -, *, %}
- Fitness function: $1/(1+0.5 \sum_{i=1}^{20} e_i^2)$

where e_i is the difference between the actual and calculated value of output y_i for a given input x_i. A total of 10 fitness cases have been chosen between the interval [0, 1].

- Control parameters: The control parameters used for this example are shown in Table 1. In addition to this, there is the provision of assigning probabilities to selection of function and terminal nodes. For this regression example, the probability of selecting a function node is taken as 0.75.
- Termination criteria: When, maximum number of function evaluations reached. Here, the maximum number of function evaluations is set to a value of 600, excluding the function evaluations for the initial population.

After defining above five preliminary steps, the actual GP process will be progressed as per the steps outlined in Fig.2. Fig. 15 illustrates the variation of the fitness of the best individual in the population, modifying at a time, two of the individuals in the entire population pool. After 284 ($2 \times 127 + 30$) function evaluations the program produces the exact parsimonious output of ($x \times (x/2)$). After this point, the same individual tree remains in the pool through out the rest of the period, which is contrast to the result reported by Banzaf et al. [2]. Fig. 16 illustrates the variation of the average fitness of all the individuals in the pool. The control parameters reported in Table 1 is a rough comparison between both the methods. To make a rigorous comparison it is required to know the implementation details of the program used by Banzaf et al.

5. DISCUSSIONS AND CONCLUSIONS

In this implementation all the basic C++ class structures have been described, which can easily be modified for incorporation of new features. Pointer based approach have been followed keeping in view the large memory requirements of GP. Of course, the additional pointer indirection requires more time to run the program, in addition to other program complexity overhead. This program has been implemented on a Pentium II based PC using Microsoft Visual C++ Compiler. The developed program can very easily be used to solve various applications with only few minor changes in the program construct such as fitness functions and new node function definitions.

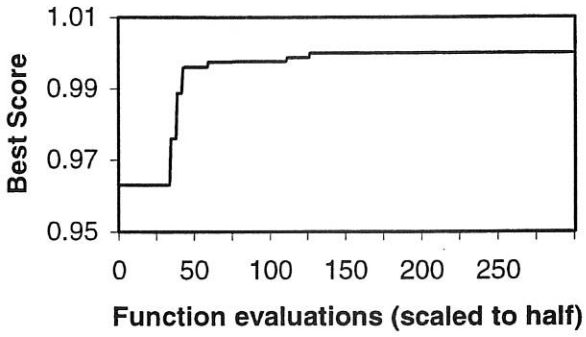


Fig. 15 Best Score of the Regression Example

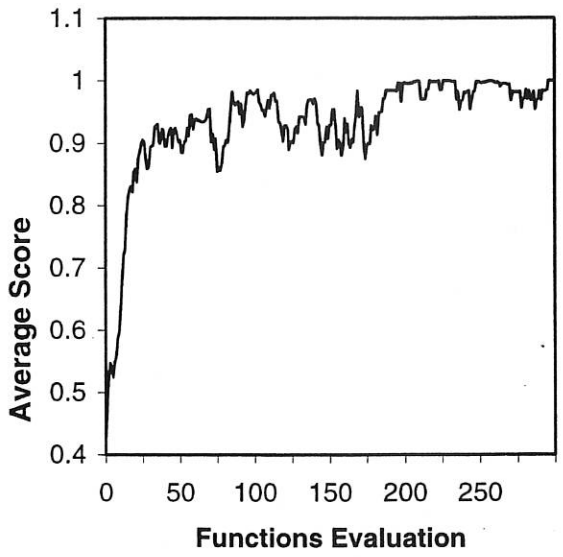


Fig. 16 Average Score of the Regression Example

This implementation stresses on the number of function evaluations rather than number of generations, where the former makes it easy to compare the performance of different available GP algorithms. The reported results on regression analysis indicates the efficacy of the developed software and to strengthen this point it is required to consider more examples. In the future work this software

with certain modification will be used for more complex task of robot control.

Acknowledgement

The first author would like to thank the commonwealth scholarship commission in the United Kingdom for providing the support to pursue this work.

Table 1: Control Parameters

Parameters	Values used in reference [2]	Values used in this paper
Population size	600	30
Selection method	Tournament Selection	Tournament Selection
Tournament Size	4	4
Maximum tree depth after crossover	200	10
Initialisation method	Grow	Grow

References

[1] J.R. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Cambridge, MA, 1992.

[2] W. Banzaf, P. Nordin, R.E. Keller and F.D. Francone, *Genetic Programming: An Introduction*. Morgan Kaufmann Publishers, Inc., USA, 1998.

[3] A.K. Swain and A.M.S. Zalzala, "An overview of genetic programming: Current trends and applications," Research Report Number 732, Robotics Research Group, Dept. of ACSE, Univ. of Sheffield, 1998.

[4] W.B. Langdon and A. Qureshi, "Genetic programming-computers using "natural selection" to generate programs," Research Note RN/95/76, University College London, UK, 1995.

[5] M.J. Keith and M.C. Martin, "Genetic programming in C++: Implementation issues," *Advances in Genetic Programming* (K.E. Kinnear, Jr., ed.), MIT Press, Cambridge, MA, pp.285-310, 1994.

[6] J. Cona, "Developing a genetic programming system," *AI Expert*, pp.20-29, Feb. 1994.

[7] A. Singleton, "Genetic programming with C++," *Byte*, Feb 1994.

[8] C. Harris and B. Buxton, "GP-COM: A distributed, component-based genetic programming system in C++," *Genetic Programming 1996: Proceedings of the First Annual Conference* (J.R. Koza, D.E. Goldberg, D.B. Fogel and R.L. Riolo, eds.), Stanford University, CA. MIT Press, Cambridge, MA, 1996.

[9] K. Chellapilla, "Evolving computer programs without subtree crossover," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 3, pp.209-216, Sep. 1997.

[10] E. Horowitz, S. Sahani and D. Meheta. *Fundamentals of Data Structure in C++*. Computer Science Press, W.H. Freeman and Co., NY, 1995.

[11] L. Ammeraal. *Algorithms and Data Structures in C++*. John Wiley & Sons, 1996.

[12] A.M. Tenenbaum, Y. Langsam and M.J. Augenstein. *Data Structures Using C*. Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1990.