GENE EXPRESSION PROGRAMMING FOR LOGIC
CIRCUIT DESIGN

by

STEVEN MANDLA MASIMULA

submitted in accordance with the requirements for
the degree of

MASTER OF SCIENCE

In the subject

APPLIED MATHEMATICS

at the

University of South Africa

Supervisor: Professor Yorick Hardy

3 October 2017
I declare that **GENE EXPRESSION PROGRAMMING FOR LOGIC CIRCUIT DESIGN** is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

__________________________  ____________________________
SIGNATURE                      DATE

(Mr S M Masimula)
Acknowledgements

I wish to extend my sincere appreciation and gratitude to the following people for their role and contribution towards my studies:

- Professor Yorick Hardy my supervisor for his tremendous patience, guidance and supervision.

- My family, friends and colleagues for their support.
Abstract

Finding an optimal solution for the logic circuit design problem is challenging and time-consuming especially for complex logic circuits. As the number of logic gates increases the task of designing optimal logic circuits extends beyond human capability. A number of evolutionary algorithms have been invented to tackle a range of optimisation problems, including logic circuit design. This dissertation explores two of these evolutionary algorithms i.e. Gene Expression Programming (GEP) and Multi Expression Programming (MEP) with the aim of integrating their strengths into a new Genetic Programming (GP) algorithm. GEP was invented by Candida Ferreira in 1999 and published in 2001 [8]. The GEP algorithm inherits the advantages of the Genetic Algorithm (GA) and GP, and it uses a simple encoding method to solve complex problems [6, 32]. While GEP emerged as powerful due to its simplicity in implementation and flexibility in genetic operations, it is not without weaknesses. Some of these inherent weaknesses are discussed in [1, 6, 21]. Like GEP, MEP is a GP-variant that uses linear chromosomes of fixed length [23]. A unique feature of MEP is its ability to store multiple solutions of a problem in a single chromosome. MEP also has an ability to implement code-reuse which is achieved through its representation which allow multiple references to a single sub-structure.

This dissertation proposes a new GP algorithm, Improved Gene Expression Programming (IGEP) which improves the performance of the traditional GEP by combining the code-reuse capability and simplicity of gene encoding method from MEP and GEP, respectively. The results obtained using the IGEP and the traditional GEP show that the two algorithms are comparable in terms of the success rate when applied on simple problems such as basic logic functions. However, for complex problems such as one-bit Full Adder (FA) and AND-OR Arithmetic Logic Unit (ALU) the IGEP performs better than the traditional GEP due to the code-reuse in IGEP.

Key terms:
Logic circuit design; Genetic algorithms; Genetic programming; Gene expression programming; Multi expression programming; Improved gene expression programming; Improved multi expression programming; Cartesian genetic programming; Automatically defined function; Multi-expression based Gene expression programming.
List of Figures

2.1 An example of GEP expression tree (ET) representing a logic expression: \( \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \). 4
2.2 Flowchart of GEP algorithm ................................................................. 5
2.3 An example of the roulette wheel representation .................................... 7
2.4 An example of prefix MEP representation of the chromosome: \( \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \&a \&b \&c \). 11
2.5 Graphical representation of MEP ......................................................... 12
2.6 Graphical representation of the MEP logic expression shown in Figure 2.4 12
3.1 A K-expression translates into three unique sub-expressions. (a) \( E_1 = (b \& c) \mid [a \& (b \& c)] \), (b) \( E_2 = b \& c \), (c) \( E_3 = a \& (b \& c) \), (d) \( E_4 = b \& c \) ........................................ 18
3.2 An example of Cartesian Genetic Programming (CGP) representation as a list of integers .... 19
3.3 An example of CGP representation in a form of an acyclic directed graph .......... 20
3.4 Prefix Improved Multi Expression Programming (IMEP) representation of the chromosome in Figure 2.1 ................................................................. 21
List of Tables

2.1 An example of roulette wheel ......................................................... 6

4.1 GEP and IGEP parameters ................................................................. 24
4.2 Truth table for AND, OR, NAND, NOR, XOR and NOT ............................... 24
4.3 Results for basic logic gates obtained using IGEP ................................. 25
4.4 Results for basic logic gates obtained using GEP ................................. 26
4.5 Truth table for an AND-OR ALU ...................................................... 27
4.6 Truth table for one-bit Half Adder (HA) ............................................. 27
4.7 Results for an AND-OR ALU and one-bit HA obtained using IGEP ............... 28
4.8 Results for an AND-OR ALU and one-bit HA obtained using GEP ............... 28
4.9 Truth table for one-bit FA ................................................................. 29
4.10 Results for one-bit FA obtained using IGEP ...................................... 30
4.11 Results for one-bit FA obtained using GEP ...................................... 30

A.1 Summary of the input file ................................................................. 32
A.2 List of operations ............................................................................. 33
List of Acronyms

GEP Gene Expression Programming
MEP Multi Expression Programming
IGEP Improved Gene Expression Programming
GP Genetic Programming
ET expression tree
IS Insertion Sequence
RIS Root Insertion Sequence
CGP Cartesian Genetic Programming
ADF Automatically Defined Function
IMEP Improved Multi Expression Programming
ALU Arithmetic Logic Unit
HA Half Adder
FA Full Adder
GA Genetic Algorithm
MGEP Multi-expression based Gene Expression Programming
Contents

1 Introduction and background 1
   1.1 Introduction ................................................................. 1
   1.2 Statement of the problem ................................................. 1
   1.3 Background of study ..................................................... 1
   1.4 Objectives of study ..................................................... 2

2 Gene expression programming and multi expression programming 3
   2.1 Gene expression programming ........................................... 3
      2.1.1 Gene representation ................................................ 4
      2.1.2 Genetic operators ................................................... 4
         2.1.2.1 Selection, replication and elitism ............................. 5
      2.1.2.2 Mutation .......................................................... 8
      2.1.2.3 Transposition of Insertion Sequence (IS) elements ........ 8
      2.1.2.4 Root transposition .............................................. 8
      2.1.2.5 Gene transposition .............................................. 9
      2.1.2.6 Recombination ................................................... 9
   2.2 Multi expression programming ......................................... 11
      2.2.1 Gene representation ................................................ 11
      2.2.2 Genetic operators ................................................... 13
         2.2.2.1 Crossover ....................................................... 13
      2.2.2.2 Mutation .......................................................... 13
   2.3 Improved gene expression programming ................................ 14
   2.4 Fitness function .......................................................... 16
   2.5 Termination criterion .................................................... 17

3 Review of literature 18

4 Results 23
   4.1 Example 1: Designing logic circuits for basic logic gates .......... 24
   4.2 Example 2: Logic circuits for an AND-OR ALU and one-bit HA .... 27
   4.3 Example 3: Logic circuits for one-bit FA ............................... 29

5 Conclusion 31
A Implementation of the IGEP algorithm

A.1 Input and output description ........................................ 32
A.2 The IGEP algorithm ......................................................... 33
Chapter 1

Introduction and background

1.1 Introduction

Savage [27, p. 35] defines a logic circuit, the basic building block of real-world computers, as a circuit in which the operations are boolean. The design of a logic circuit is the process of determining, from input/output behaviour specification, a structure (a combination of logic gates) that is functional such that, for given inputs, the structure implements a given truth table. Above all, this design must be as optimal as possible in terms of specified constraints (e.g. the number of gates) [5]. The design of functional logic circuits (especially complex ones) can be cumbersome for human designers, let alone designing optimum logic circuits. On this note, various genetic algorithms for automation of the logic circuit design process have been developed. However, the focus is and has always been on the performance of these algorithms. This research explores the application of one type of genetic algorithms, GEP, in the area of logic circuit design. Most importantly, in this study the efficiency of the GEP algorithm is enhanced by borrowing strength from another genetic algorithm called MEP [23, 31]. The enhanced GEP algorithm is referred to as IGEP algorithm.

Section 2.1 of this dissertation outlines the GEP while the MEP and IGEP algorithms are discussed in Sections 2.2 and 2.3, respectively. Literature review is conducted in Chapter 3 and the results of nine case studies are presented in Chapter 4. Chapter 5 concludes the dissertation.

1.2 Statement of the problem

More formally, the focus of this research is to design a functional optimum logic circuit that performs a desired function (specified by a truth table), given a certain specified set of logic gates using IGEP, the modified GEP algorithm. The complexity of a logic circuit is a function of the number of gates in the circuit. The complexity of a logic gate is generally the number of inputs to it.

1.3 Background of study

As the scale of logic circuit design increases (i.e. increasing number of logic gates), the problem of logic circuit design, let alone determining the optimality of the resulting circuit, extends beyond human capability. It is expected that the results of this research will assist in terms of simplifying the design of logic circuits and thus improve on the time it takes to produce optimal logic circuits. This research focuses on the implementation
using an improved GEP, IGEP, which integrates reuse of common sub-structures (genes) into the traditional GEP. IGEP borrows strength from MEP’s sub-expression reuse feature. With this feature, the MEP is able to repeatedly use the same sub-expression (common sub-structure) in an expression without repeating it. This feature improves the efficiency of the MEP algorithm as it takes shorter time to evaluate chromosomes since repeated sub-structures are evaluated only once and reused in the subsequent stages of evaluation.

1.4 Objectives of study

The main objectives of this research are:

- To explore the GEP and MEP techniques.
- To demonstrate how GEP can be applied in the area of logic circuit design.
- To improve the efficiency of GEP by reusing sub-structures (genes).
Chapter 2

Gene expression programming and multi expression programming

2.1 Gene expression programming

GEP was invented in 1999 and appeared in a publication for the first time in 2001 by Candida Ferreira [8]. GEP uses the same kind of diagram representation of GP, but the entities produced by GEP are the expression of genome (ETs). GEP led to the invention of chromosomes capable of representing any ET. Like genetic algorithms and GP, GEP is a genetic algorithm as it uses populations of individuals, selects them according to fitness, and introduces genetic variation using one or more genetic operators [8]. The main building blocks of GEP are chromosomes and the ETs. For that purpose a language called Karva [8] was created to read and express the information of GEP chromosome. Furthermore, the structure of chromosomes was designed to allow the creation of multiple genes, each encoding a sub-ET. The genes are structurally composed of a head and tail and it is this structural and functional organisation that always guarantees the production of valid chromosome, no matter how much or how profoundly the chromosomes are modified.

The ETs express the genetic information encoded in the chromosomes. Translating chromosomes to ETs requires some kind of code and rules. The genetic code is very simple: a one-to-one relationship between symbols and functions or the terminals they represent. The rules are also simple: they determine the spatial organisation of the functions and terminals in the ETs and the type of interaction between the sub-ETs in multi-genic systems [8].

In the context of the Karva language, given the chromosome (genotype), the phenotype can easily be represented by an ET for example as shown in Figure 2.1. For this example, a set of functions $F = \{ & , | \}$ and the set of terminals $T = \{a, b, c\}$ are used, where symbols $|$ and $\&$ represent logical functions OR and AND, respectively. The example shows one-genic chromosome in the form of a logic expression, GEP prefix representation and an ET. The bold part is the head and remaining part is the tail.
Logic expression: \((b \& c) \land (a \& (b \& c))\)

Prefix representation of GEP chromosome: \(\& bc \& a \& b cacbccccabbc\)

Expression tree:

![Expression tree](image_url)

Figure 2.1: An example of GEP ET representing a logic expression: \(\& bc \& a \& b cacbccccabbc\).

In this example, the bold part of the GEP chromosome is the head and it also coincides with what is called K-expression in Karva language. K-expression is the valid segment of the chromosome obtained by reading the ET from left to right and from top to bottom; and it can be represented as an ET shown in Figure 2.1.

### 2.1.1 Gene representation

GEP genes are composed of a head and a tail. The head contains symbols that represent both functions and terminals, whereas the tail contains only terminals. For each problem, the length of the head \(h\) is chosen, whereas the length of the tail \(t\) is a function of \(h\) and the number of arguments of the function with most arguments \(n\) (arity), and is given by \([8, 9]\):

\[
\begin{align*}
  t &= h(n - 1) + 1 \\
\end{align*}
\]  

(2.1)

GEP chromosomes are usually composed of more than one gene of equal length. For each problem or run, the number of genes, as well as the length of the head, are \textit{a priori} chosen. Each gene codes for sub-ET and the sub-ETs interact with one another forming a more complex multi-subunit ET \([8, 9]\).

Figure 2.1 shows a one-genic chromosome of length 19. The head has length \(h=9\) and the arity is \(n=2\) which gives a tail of length \(t=10\).

### 2.1.2 Genetic operators

Genetic operators are the core of all genetic algorithms and two of them are common to all evolutionary systems i.e. selection and replication. The following operators (discussed in detail in \([8]\) and summarised in \([30, p. 455]\))
are used in GEP to evolve or introduce genetic variation into the population:

a. selection, replication and elitism

b. mutation

c. transposition of IS elements

d. root transposition

e. gene transposition

f. recombination - this is a crossover operation. It can take any of the three forms:
   - one-point recombination
   - two-point recombination
   - gene recombination.

In [1], the GEP algorithm is summarised as shown in Figure 2.2, below:

![Flowchart of GEP algorithm](image)

Figure 2.2: Flowchart of GEP algorithm

Each of these operators will be discussed in detail in the next sections.

### 2.1.2.1 Selection, replication and elitism

Selection is the process of determining the number of times a particular individual is chosen for reproduction and, thus, the number of offspring that an individual will produce. The principle behind genetic algorithms is essentially Darwinian natural selection. There is a number of selection methods that one can choose from e.g. ranking selection, tournament selection and roulette wheel (fitness-proportional) selection. However, the roulette wheel selection is a popular selection method. In *rank selection*, selection is based on the rank (not
the numerical value) of the fitness values of the individuals in the population while in tournament selection, a specified group of individuals (typically two) are chosen at random from the population and the one with the better fitness (i.e., the lower standardised fitness) is then selected [15, p. 100]. The fitness value for each individual is calculated based on the fitness function. The detailed discussion on the fitness function can be found in Section 2.4. The roulette wheel selection probabilistically selects individuals based on their fitness values $f_i$ such that $f_i > 0$. The probability that an individual (chromosome) $i$ is selected, $p_i$, is computed as:

$$p_i = \frac{f_i}{\sum_{k=1}^{n} f_k}.$$  \hspace{1cm} (2.2)

Since now $\sum_{i=1}^{n} p_i = 1$, this method allows the fitness values to be used as probabilities. Individuals are then mapped one-to-one into contiguous intervals proportionally to their fitness i.e. the size of each individual interval corresponds to the fitness value of the associated individual. The circumference of the roulette wheel is the sum of all fitness values of the individuals. The fittest individual occupies the largest interval or segment, whereas the least fit have correspondingly smaller intervals within the roulette wheel, see Figure 2.3.

Using the roulette wheel selection, parents are selected according to their fitness. The greater the fitness of a chromosome, the greater the chance of that chromosome being selected. To select two individuals for reproduction, two random numbers, $r_1$ and $r_2$ are generated such that $0 \leq r_1 \leq 1$ and $0 \leq r_2 \leq 1$. Two chromosomes $g_i$ and $g_j$ (where $i$ and $j$ are positions in the population) are chosen by means of the following criteria:

$$\sum_{u=1}^{i} p_u \geq r_1 \geq \sum_{u=1}^{i-1} p_u,$$ \hspace{1cm} (2.3)

$$\sum_{v=1}^{j} p_v \geq r_2 \geq \sum_{v=1}^{j-1} p_v.$$ \hspace{1cm} (2.4)

That is, a random number, $r$, is generated in the interval $[0, 1]$ and the first population member whose probability of selection added to the preceding population members (cumulative probability of selection) is greater than or equal to $r$ is selected. This process is repeated until the desired number of individuals has been selected.

Table 2.1 below shows an example of a roulette wheel. Consider a population of seven individuals with associated fitness and probability of selection proportional to fitness.

<table>
<thead>
<tr>
<th>Population ($i$)</th>
<th>Fitness ($f_i$)</th>
<th>$p_i$</th>
<th>Cumulative $p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual 1</td>
<td>4</td>
<td>0.098</td>
<td>0.098</td>
</tr>
<tr>
<td>Individual 2</td>
<td>8</td>
<td>0.195</td>
<td>0.293</td>
</tr>
<tr>
<td>Individual 3</td>
<td>5</td>
<td>0.122</td>
<td>0.415</td>
</tr>
<tr>
<td>Individual 4</td>
<td>6</td>
<td>0.146</td>
<td>0.561</td>
</tr>
<tr>
<td>Individual 5</td>
<td>7</td>
<td>0.171</td>
<td>0.732</td>
</tr>
<tr>
<td>Individual 6</td>
<td>8</td>
<td>0.195</td>
<td>0.927</td>
</tr>
<tr>
<td>Individual 7</td>
<td>3</td>
<td>0.073</td>
<td>1.000</td>
</tr>
</tbody>
</table>

| SUM              | 41             | 1.000|
Graphically, the roulette wheel can be represented as a wheel in which segments are of possibly different sizes, based on each individual’s relative fitness, see Figure 2.3 below.

![Roulette Wheel Diagram](image)

Figure 2.3: An example of the roulette wheel representation

To select two individuals for reproduction from the population shown in Table 2.1, suppose two random numbers, $r_1 = 0.814$ and $r_2 = 0.635$ are generated. Using $r_1$ the first parent chromosome to be selected for reproduction is ‘Individual 6’ as $\sum_{v=1}^{6} p_v = 0.927 \geq r_1 = 0.814$. The second parent chromosome to be selected is ‘Individual 5’ as $\sum_{v=1}^{5} p_v = 0.732 \geq r_2 = 0.635$. It should be noted that in this study the lower the fitness value the fitter the individual, hence the roulette wheel is applied on the inverse of the fitness value.

It is worth noting that probabilistic operations enter the genetic algorithms in three different phases. First, the initial population must be selected. This choice can be made randomly (or if some prior knowledge of good starting points exist, these can be chosen). Next, members of the population have to be selected for reproduction. One way to do this is to select individuals probabilistically based on their fitness. The third way probabilities enter into consideration is in the selection of the genetic operation to be used.

As described in [26, 18], elitism is essentially a mechanism that protects the best chromosomes in subsequent generations. In classical genetic algorithms the best individuals are not always transferred to the next generation. It does not always happen that the next generation contains the fittest chromosomes from the current population. Elitism is applied in order to protect populations against the loss of fittest individuals as a result of genetic operators. The fittest individuals are always carried forward to the next generation unaltered and thus the minimum fitness of the population can never reduce from one generation to the next. Above all, elitism usually brings about a more rapid convergence of the population.
2.1.2.2 Mutation

This operation involves changing symbols in the chromosome but this is done such that the structure of the chromosome remains intact i.e. symbols in the tail of a gene may not operate on any arguments. For example, consider the following one-genic chromosome:

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\text{Parent 1: } & a & b & c & a & c & b & a & a & c & b & b & b & b & b & b
\end{array}
\]

Suppose a mutation changed the 'c' in position 6 to 'long' and the 'c' in position 6 to 'long', obtaining:

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\text{Child 1: } & a & b & c & a & c & b & a & a & c & b & b & b & b & b & b
\end{array}
\]

It should be noted that if a function is mutated into a terminal or vice versa, or a function of one argument is mutated into a function of two arguments or vice versa, the ET is modified drastically [8]. Sometimes, mutation takes place in the non-coding region of the chromosome, leading to what is called neutral mutation.

2.1.2.3 Transposition of IS elements

A portion of a chromosome is chosen to be inserted in the head of a gene. The tail of the gene remains unaffected. Thus symbols are removed from the end of the head to make room for the inserted string [30]. The transposition operator randomly chooses the chromosome, the start of the IS element, the site, and the length of the transposon. A transposition rate of 0.1 is typically used. Consider the following one-genic parent chromosome:

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\text{Parent 1: } & a & b & c & a & c & b & a & a & c & b & b & b & b & b & b
\end{array}
\]

Suppose that a sequence 'bbaa' in position 6 to 9 is chosen to be an IS element and inserted in the head starting from position 1, obtaining:

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\text{Child 1: } & a & b & a & a & b & b & a & a & c & b & b & b & b & b & b
\end{array}
\]

2.1.2.4 Root transposition

All Root Insertion Sequence (RIS) elements start with functions, and thus are chosen among the sequences of the heads. For that, a point is randomly chosen in the head and the gene is scanned downstream until a function is found. This function becomes the start position of the RIS element. If no functions are found, the operator does nothing. Typically, a root transposition rate of 0.1 is used. As an example, consider the following one-genic parent chromosome:
Suppose the sequence ‘ja’ in position 3 to 4 in the parent chromosome above is copied into the root of the chromosome, resulting in:

Child 1: | a | & & | a c a b a c b b b

2.1.2.5 Gene transposition

In gene transposition, an entire gene functions as a transposon and transposes itself to the beginning of the chromosome. In contrast with other forms of transposition, in gene transposition the transposon (the gene) is deleted in the place of the origin [1]. One gene (except the first) in a chromosome is randomly chosen to be the first gene. All other genes in the chromosome are shifted downwards in the chromosome to make place for the first gene. Consider the following two-genic chromosome:

<table>
<thead>
<tr>
<th>Gene 1</th>
<th>Gene 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>Parent 1:</td>
<td>&amp; a b a a b b a</td>
</tr>
<tr>
<td>Child 1:</td>
<td>&amp; &amp;</td>
</tr>
</tbody>
</table>

In this example, gene 2 is selected for gene transposition by default since there are only two genes in the chromosome, giving:

<table>
<thead>
<tr>
<th>Gene 1</th>
<th>Gene 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>Child 1:</td>
<td>&amp; &amp;</td>
</tr>
</tbody>
</table>

In case of a two-genic chromosome the operator essentially swaps the positions of the genes.

2.1.2.6 Recombination

In GEP, there are three kinds of recombination: one-point, two-point and gene recombination. In all cases, two parent chromosomes are randomly chosen and paired to exchange some genetic material between them.

a. One-point recombination

In one-point recombination the chromosomes are paired and split in the same point and corresponding sections are swapped. Consider the following two one-genic parent chromosomes for one-point recombination:

| Parent 1: | a | & & | a c a b a c b b b |
| Parent 2: | & & | b a a | b c a a a c b b b |
Suppose that position 3 is chosen as recombination point, resulting in the following children:

- **Child 1:**
  - 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
  - a a a b c a a c b b b

- **Child 2:**
  - 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
  - & b & & a c a a c b b b

**b. Two-point recombination**

In this operator, two parent chromosomes are split into three and the middle portion is swapped forming new children. The recombination points are chosen randomly, for example:

- **Parent 1:**
  - 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
  - a & & a c a b a c b b b

- **Parent 2:**
  - 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
  - & b & a a & b c a a a c b b b

Suppose positions 3 and 10 are selected randomly such that the genetic material between positions 3 and 10 is swapped between the parent chromosomes to produce two children:

- **Child 1:**
  - 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
  - a a b c a a a c b b b

- **Child 2:**
  - 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
  - & & b & & a c a a a c b b b

**c. Gene recombination**

In gene recombination, genes are exchanged between two parent chromosomes, forming two children containing genes from both parents. The exchanged genes are randomly chosen and occupy the same position in the parent chromosomes. Below is an example of two two-genic parent chromosomes:

- **Gene 1**
  - 0 1 2 3 4 5 6 7 8
  - Parent 1: & & c a b b a b
  - Parent 2: & & b & & a c c b c

- **Gene 2**
  - 0 1 2 3 4 5 6 7 8
  - Parent 1: & & a b c a b c c
  - Parent 2: & & a b a a c b a

Suppose the second gene in parent chromosome 1 is chosen at random and swapped with the second gene in parent chromosome to obtain:

- **Gene 1**
  - 0 1 2 3 4 5 6 7 8
  - Parent 1: & & c a b b a b
  - Parent 2: & & b & & a c c b c

- **Gene 2**
  - 0 1 2 3 4 5 6 7 8
  - Parent 1: & & a b c a b c c
  - Parent 2: & & a b a a c b a

- **Child 1:**
  - 0 1 2 3 4 5 6 7 8
  - & & c a b b a b

- **Child 2:**
  - 0 1 2 3 4 5 6 7 8
  - & & b & & a c c b c
The children contain entire genes from both parents. In this kind of recombination, similar genes can be exchanged but, most of the times, the exchanged genes are very different and new material is introduced in the population. However, with this operator, no new genes are created [8].

2.2 Multi expression programming

Like GEP, MEP is a GP variant that uses linear chromosomes of fixed length [23]. A unique feature of MEP is its ability to store multiple solutions of a problem in a single chromosome. Note that this feature does not increase the complexity of the MEP decoding process when compared to other techniques storing a single solution in a chromosome. Studies show that MEP performs better than other competitor techniques such as GEP and CGP [19] for some well-known problems such as symbolic regression [23]. As already indicated, GEP has been implemented successfully in the area of logic circuit design. However, the performance and efficiency of these algorithms has always been an issue despite the proposed improvements by various researchers. On this note, this research integrates the features of the MEP into GEP in order to improve the performance and efficiency of GEP, notably the ability of MEP to represent a common sub-structure in a chromosome without repeating it i.e. code-reuse. The code-reuse ability in MEP is achieved through its representation which allow multiple references to a single sub-structure.

2.2.1 Gene representation

The MEP representation ensures that no cycle arises while the chromosome is decoded. Figure 2.4 shows an example of MEP representation. According to this representation scheme the first symbol of the chromosome must be a terminal symbol. Each gene encodes a terminal or function symbol. A gene encoding a function includes pointers towards the function arguments which always point to expressions in earlier positions in the chromosome. In this way only syntactically correct MEP individuals are obtained since the translation of a MEP chromosome is done by reading the chromosome top-down with later expressions referencing earlier expressions via pointers [23]. The prefix MEP representation of the chromosome in Figure 2.1 is given below in Figure 2.4:

0: b
1: c
2: & 0, 1
3: a
4: & 2, 3
5: | 2, 4

Figure 2.4: An example of prefix MEP representation of the chromosome: |&bc&ca&bc|abc|ccc|abc|bc.

Note that the sub-structure ‘&bc’ is repeated in Figure 2.1. As can be seen, in the MEP representation this common sub-structure is not repeated. It is exactly this ability of MEP to repeatedly use the same sub-expression (common sub-structure) in an expression without repeating it that this research aims to exploit to the benefit of GEP as it takes shorter time to evaluate chromosomes; since repeated sub-structures are evaluated only once and the results are reused in the subsequent stages of evaluation.
MEP can be expressed by a directed graph of indexed nodes. The graph has a set of \( n_i \) inputs that are indexed as nodes 0 to \( n_i - 1 \), a set of \( n_n \) nodes and a set of \( n_o \) outputs. Each \( n_n \) node has a number of inputs and a function which computes an output based on the inputs as shown in Figure 2.5 [10]. The genotype is a list of integers that determine the connectivity and functionality of the nodes. As shown in [10], these can be mutated and crossed over to create new directed graphs.

Figure 2.5: Graphical representation of MEP

As an example, the logic expression shown in Figure 2.4 translates to the directed graph shown in Figure 2.6.

Figure 2.6: Graphical representation of the MEP logic expression shown in Figure 2.4
According to [23], using the above representation, a terminal symbol specifies a simple expression and a function symbol specifies a complex expression obtained by connecting the operands specified by the argument positions with the current symbol. The chromosome fitness is defined as the number of non-repeated evaluations in an expression e.g. there are six function evaluations in the prefix MEP representation of the chromosome shown above.

Notable differences between GEP and MEP [21, p. 12] are:

- The pointers toward function arguments are encoded explicitly in MEP chromosomes whereas in GEP these pointers are computed when chromosomes are parsed. MEP also needs to evolve the pointers toward function arguments. This leads to a more compact representation of GEP when compared to MEP.
- MEP representation is suitable for code-reuse, while the GEP is not.
- The non-coding regions of GEP are always at the end of the chromosome, whereas in MEP these regions can appear anywhere in the chromosome.

2.2.2 Genetic operators

Unlike GEP, two genetic operators are used in MEP: crossover and mutation. These operators preserve the chromosome structure. All offsprings obtained by crossover and mutation are always syntactically correct [20, p. 4]. Thus, no extra processing for repairing newly obtained individuals is needed [11, p. 105]. In [10], the author notes that applying genetic operators (i.e. mutation and crossover) to a graph structure is not simple since the integrity of the chromosome must not be compromised. In contrast, the mutation operator in GEP is simple as there is only one restriction i.e. symbols in the tail of a gene may not operate on any arguments.

2.2.2.1 Crossover

Crossover points are picked at random i.e. any point along the parent chromosome can be chosen and everything before this point goes to one child while everything after goes to the other child [10]. The crossover process involves the random selection of two parent chromosomes for recombination. For instance, within the uniform recombination the offsprings are taken randomly from one parent or another. As an example, consider two parent chromosomes where positions 1, 2 and 5 are crossed:

<table>
<thead>
<tr>
<th>Parent 1</th>
<th>Parent 2</th>
<th>Child 1</th>
<th>Child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: b</td>
<td>0: a</td>
<td>0: b</td>
<td>0: a</td>
</tr>
<tr>
<td>1: c</td>
<td>1: b</td>
<td>1: b</td>
<td>1: c</td>
</tr>
<tr>
<td>2: &amp; 0, 1</td>
<td>2:</td>
<td>0, 1</td>
<td>2:</td>
</tr>
<tr>
<td>3: a</td>
<td>3: &amp; 0, 1</td>
<td>3: a</td>
<td>3: &amp; 0, 1</td>
</tr>
<tr>
<td>4: &amp; 3, 2</td>
<td>4:</td>
<td>2, 3</td>
<td>4: &amp; 3, 2</td>
</tr>
<tr>
<td>5:</td>
<td>2, 4</td>
<td>5:</td>
<td>2, 4</td>
</tr>
</tbody>
</table>

2.2.2.2 Mutation

Determining which point to mutate is done by randomly choosing a point along the chromosome. Each symbol (terminal, function, or pointer) in the chromosome may be the target of the mutation operator [21]. If a pointer was chosen, it can only be mutated to a random value between 1 and $n - 1$, where $n$ is the current node while
functions can be mutated to any function (and arguments) in the function set and a terminal or an expression can mutate to some value in the terminal set or any expression with valid pointers i.e. the pointers may only point to nodes that come before the node that is being mutated. Most importantly, to preserve the consistency of the chromosome its first gene must encode a terminal symbol [10, 21]. Consider the following example, whereby the bold symbols in a parent chromosome are selected for mutation:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: a</td>
<td>0: a</td>
</tr>
<tr>
<td>1: b</td>
<td>1: &amp; 0, 0</td>
</tr>
<tr>
<td>2:</td>
<td>0, 1</td>
</tr>
<tr>
<td>3: c</td>
<td>3: c</td>
</tr>
<tr>
<td>4: &amp; 3, 2</td>
<td>4:</td>
</tr>
<tr>
<td>5: &amp; 4, 0</td>
<td>5: &amp; 4, 0</td>
</tr>
</tbody>
</table>

As an example, from the above figure, pointer 2 can only be mutated to either pointer 0 or 1. Function (&) in pointer 4 can be mutated to any function in the function set and a terminal (e.g. b in pointer 1) or an expression (e.g. & 4, 0 in pointer 5) can mutate to any value in the terminal set or any expression with valid pointers i.e. the pointers may only point to nodes that come before the node that is being mutated.

### 2.3 Improved gene expression programming

GEP tends to provide simplicity in implementation and a flexibility in genetic operations compared to the other methods described in this dissertation. We improve on these strengths while largely preserving the simplicity of the implementation and flexibility of genetic operations.

In terms of the basic structure and application of genetic operators, the IGEP is the same as the traditional GEP; hence the details that were discussed in Section 2.1 are not repeated here. This section highlights only the improvements in IGEP. The only difference between GEP and IGEP is the MEP-like sub-structure (gene) reuse ability of IGEP which was implemented to improve efficiency. As with Automatically Defined Functions (ADFs), from an implementation point of view, the introduction of reusable genes adds an extra step to the preparatory steps from the GEP application. The extra step defines the architecture in terms of number of genes to be reused [24, p. 24]. However, gene-reuse greatly reduces the effort required in solving large complex problems while increasing the robustness of IGEP. The IGEP algorithm can be summarised as follows:

Let:

\[ T = \text{terminal set} \]
\[ n = \text{number of genes} \]
\[ N = \text{population size} \]

\( T \) denotes the set of terminals, however the set of terminals differs according to the gene position in the chromosome. A later gene may refer to an earlier gene \( i \) via the terminal symbol \( i \). Gene 1 begins with a base terminal set \( T_0 \) which does not reference any other gene.
Step 1: Population initialisation

for \( j = 1 \) to \( N \) do
   \( T \leftarrow T_0 \) (initialise the terminal set);
   for \( i = 1 \) to \( n \) do
      randomly generate gene \( i \) according to the GEP rules;
      \( T \leftarrow T \cup \{i\} \)
   end
end

Step 2: GEP algorithm

2.1: Apply selection based on roulette wheel selection and elitism algorithms such that 10% of the current population is transferred to the next generation (including the top 5% best chromosomes).

2.2: Apply genetic operators. The parent chromosomes are selected for reproduction using the roulette wheel selection.

2.3: Evaluate current population.

2.4: While the maximum number of iterations is not reached, repeat steps 2.1 to 2.3.

2.5: Output the best chromosome(s) found.

It is noteworthy that the terminal set is dynamically extended by a subset of references to the reusable genes. Each reference is given a unique terminal name. The reuse is done such that an upstream gene within a chromosome can only reference downstream genes within the same chromosome i.e. a gene cannot reference later (upstream) genes or itself. As in MEP, any chromosome in the population may only contain references to the preceding genes as discussed in the previous example. With this constraint it is ensured that no loops are created. Only five selected genetic operators were used for the purpose of this dissertation i.e. Mutation, Root transposition within genes, One- and Two-point recombination and Gene recombination within chromosomes. During mutation, any terminal or reference terminal in the head of the selected gene can only be mutated into a function or another terminal or reference terminal pointing to an earlier gene. Any terminal or reference terminal in the tail of the selected gene can only be mutated into another terminal or reference terminal pointing to an earlier gene. This constraint is necessary to avoid forward referencing. It should be noted that the reused genes are selected at random during the creation of genes and therefore it is possible for a particular gene not to reference any of the downstream genes at all.

The effect of encapsulation or referenced gene (sub-structure) is that the referenced gene in the newly created gene is no longer subject to the potentially disruptive effects of the crossover operator because it is now an indivisible single point. In effect, the encapsulated genes are potential building blocks for future generations and for solving the problem at hand which could be complex in nature. Note that they may proliferate in the population in later generations [15, p. 112], [7]. As discussed in Section 2.1, no function may appear in the tail of a gene in the traditional GEP. However, in IGEP, the gene reference may appear in the tail of the gene while keeping the structure of the traditional GEP intact. This feature provides an additional benefit to IGEP as symbols in the tail are implicitly allowed to represent functions. The main benefits of IGEP are: gene-reuse capability and that, through encapsulation, good genes are preserved from destructive nature of the crossover operator. On the other
hand, with IGEP block structures may be obscured and the complexity of a chromosome may be concealed as a result of encapsulation. Furthermore, from an implementation point of view, as the number of reused genes increases the performance of the IGEP algorithm slows down as a result of the increasing number of terminals to be used in the computation. As an example, consider the following three-genic chromosome in the population:

<table>
<thead>
<tr>
<th>Gene 1</th>
<th>Gene 2</th>
<th>Gene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>0 1 2 3 4 5 6 7 8</td>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>a &amp;</td>
<td>c a b b a b</td>
<td>b a A c a b A c</td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
</tbody>
</table>

The new terminals, A and B in the above genes reference genes in positions 1 and 2, respectively, within the chromosome. Using this representation, the maximum number of genes that can potentially be reused or referenced is 26, using alphabets A to Z corresponding to the genes in positions 1 to 26 in the chromosome. The maximum number of 26 referenced genes is considered more than enough taking into account the complexity of the problems considered in this study. Note that in this representation, the genes are not linked, except through referencing, and each gene within a chromosome represents a potential solution to the problem as in MEP. However, in this study the best gene in terms of fitness was chosen to represent the potential solution in which the less fit genes are allowed to act as building blocks of the best gene through referencing. Executing this process, genes are reused as building blocks for individual chromosomes in a MEP fashion. It is this ability of MEP to repeatedly use the same gene (sub-structure) in an expression without repeating it that this research aims to exploit to improve efficiency of the IGEP as it is expected to take shorter time to evaluate chromosomes; since reused genes are evaluated only once and the results are reused in the subsequent stages of evaluation.

### 2.4 Fitness function

Fitness is the driving force of Darwinian natural selection and, likewise, of both conventional genetic algorithms and genetic programming. In nature, the fitness of an individual is the probability that it survives to the age of reproduction and reproduces. The fact that individuals exist and survive in the population and successfully reproduce may be indicative of their fitness as is the case in nature. Fitness may be measured in various ways, some explicit and some implicit. The most common approach to measuring fitness is to create an explicit fitness measure for each individual in the population [15]. The goal of this study is to produce a fully functional design (i.e. one that produces the expected behaviour stated by its truth table) which minimises the number of functions used. On this note, the calculation of fitness function $F$ in this study is divided into two parts $f_1$ and $f_2$ that measure the functionality (i.e. compliance with the truth table) and the complexity (i.e. total number of functions used), respectively [12, 25].

Let $x_i$ be the bit string for each truth table row and $y_i$ the expected output. Let $g_i$ be the value computed by the gene for input $x_i$, such that:

$$f_1 = \sum_{j=1}^{r} \text{XOR}(y_i, g_i).$$

(2.5)

$f_1$ represents the total number of outputs produced by the IGEP circuit not matching with the expected values, according to the truth table (on a bit-per-bit basis). The fitness value of an individual gene ($G_i$) is a weighted
sum of the number of incorrect results ($f_1$) and the number of gates used ($f_2$) given by:

$$f_{G_i} = w_r f_1 + w_g f_2,$$

(2.6)

where $w_r, w_g > 0$ are “weights” indicating the relative importance of optimality versus correctness. The sum is weighted in order to give preference to correct solutions over short expressions. In this manner, the algorithm searches for a solution that gives a correct output for each given combination of inputs and is also small (i.e. using the lowest possible number of logic gates) [13, p. 88]. In turn the fitness of each chromosome in the population was taken as $F = \min\{f_{G_1}, f_{G_2}, f_{G_3}, \ldots, f_{G_n}\}$, where $n$ is the number of genes in the chromosome. Here, a lower “fitness” value $F$ indicates a fitter chromosome. Clearly, when using this fitness function individuals within the population are rewarded on the basis of optimality and functionality. For the purpose of this dissertation, a number of constants were tested and; 1000 and 1 were found to be suitable for $w_r$ and $w_g$, respectively. This clearly puts more emphasis on the correctness than the optimality of the chromosome.

2.5 Termination criterion

Termination criterion is a criterion by which the algorithm decides whether to continue searching or stop the search. There is a wide range of termination criteria one can choose from. These criteria can be summarised as follows [29, p. 59]:

- Maximum generations - The algorithm stops when the specified number of generations have evolved.
- Elapsed time - The genetic process will end when a specified time has lapsed.
- No change in fitness - The genetic process will end if there is no change to the population’s best fitness for a specified number of iterations.

The termination criterion in this study was specified in terms of maximum number of generations. However, it may not always be possible for the algorithm to converge to exactly one individual (solution) since there may be more than one competing individuals with the same fitness, in which case there is no basis for choosing one individual over another (we may choose arbitrarily).
Chapter 3

Review of literature

The problem of evolving digital circuits has been intensely analysed in the recent past. The focus has been largely on evolving efficient or inexpensive digital circuits with fewer gates and fewer gate inputs per gate [14]. In [15] combinational circuits were designed using GP. Koza has designed, for example, a two-bit adder, using a small set of gates (AND, OR, NOT), but his emphasis has been on generating functional circuits rather than optimising them [5].

Most recently, Deng, He and Huang [6] have developed a new variant of the GP algorithm called Multi-expression based Gene Expression Programming (MGEP). As the name suggests, the MGEP is a modified GEP algorithm based on MEP. This means that the MGEP follows exactly the same evolutionary process as the traditional GEP [6]. The MGEP addresses the weaknesses of both GEP and MEP as highlighted in detail in [6]. MGEP achieves this by combining the strengths of GEP and MEP i.e the simplicity of encoding method from GEP and the ability of MEP to store multiple solutions of a problem in a single chromosome. The gene representation in MGEP is the same as in the traditional GEP i.e. the head contains symbols that represent both functions and terminals while the tail contains only terminals. As in MEP, a gene in MGEP can be decomposed into multiple expressions representing multiple solutions to a problem and the gene fitness is taken to be the fitness value of the best sub-expression. Using the example presented in Section 2.1, in MGEP, the K-expression $\&bc\&a\&bc$ is decomposed into four ETs. The first ET is obtained by reading the K-expression from the first head character, which is exactly the same as with the traditional GEP while second ET is then obtained by reading the K-expression from the second character head and so on, see Figure 3.1:

![Figure 3.1: A K-expression translates into three unique sub-expressions.](image)

(a) $E_1 = (b \& c) \mid [a \& (b \& c)]$, (b) $E_2 = b \& c$, (c) $E_3 = a \& (b \& c)$, (d) $E_4 = b \& c$
Note that unlike MEP, the MGEP algorithm does not generate an ET for a single node and it allows explicit repetition of sub-expressions, see Figure 3.1. The MGEP algorithm uses exactly the same genetic operators as the standard GEP as discussed in Section 2.1.2. Based on the experimental results of the two symbolic regression problems presented in [6], the MGEP algorithm was found to have significantly improved the evolutionary performance when compared with both the MEP and the GEP algorithms owing to the improved way of decoding and assigning of fitness to a gene in MGEP [6].

Julian Miller, one of the pioneers in the field of evolvable digital circuits, used a special technique called Cartesian Genetic Programming (CGP) [19] for evolving digital circuits. CGP was invented by Miller in 1999 and was developed from a representation of electronic circuits devised by Miller and Thomson a few years earlier. CGP is a highly efficient and flexible form of GP that encodes a computer program as an acyclic directed graph whose nodes (gates) are organised in \( n_c \) columns and \( n_r \) rows. Depending on a particular application, the nodes can be elementary logic functions, transistors or high-level components such as adders or multipliers [4, 28]. CGP represents computational structures (mathematical equations, circuits, computer programs etc.) as a string of integers. These integers, known as genes determine the functions of nodes in the graph, the connections between nodes, the connections to inputs and the locations in the graph where outputs are taken from. As discussed in [4], originally CGP used a program topology defined by a rectangular grid of nodes with a user-defined number of rows and columns. However, later work on CGP showed that it was more effective when the number of rows was chosen to be one. Using a graph representation is very flexible as many computational structures can be represented as a graph. The results have shown that CGP was able to evolve digital circuits better than those designed by human experts [23]. In the context of the CGP, the chromosome shown in Figure 2.1 is represented as a list of integers as follows:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 0 & 3 & 0 & 1 & 4 & 3 & 5 \\
3 & 4 & 5 & \text{Output}
\end{array}
\]

Figure 3.2: An example of CGP representation as a list of integers

In CGP, the integers are mapped to acyclic directed graphs rather than trees. One motivation for the CGP is that it uses graphs that are more general than trees. Figure 3.3 gives a graphical representation of the above chromosome. Note that 0 and 1 refer to the logic operations AND (\&) and OR (\(||\)).
As correctly pointed out in [4] the benefit of this type of representation is that it allows the implicit reuse of nodes in the directed graph.

In the quest to improve the performance of Genetic Programming (GP), Koza introduced code-reuse in GP by using ADFs [15]. According to Koza et al. [16], an ADF is a function that is dynamically evolved during a run of genetic programming and that may be called by a calling program (or subprogram) that is concurrently being evolved. When ADFs are used, a program in the population consists of hierarchy of one (or more) reusable function-defining branches (i.e. ADFs) along with a main result-producing branch [16, 15]. As pointed by Qureshi [24, p. 24], from an implementation point of view, the introduction of ADFs adds an extra step to the preparatory steps from the GP application. The extra step defines the architecture in terms of number of ADFs and the arguments that they take, and the result-producing branch. Nonetheless, owing to the code-reuse capability of the ADFs, the use of the ADFs has been found experimentally to greatly reduce the computational effort required to generate correct computer programs [2].

On the other hand, Oltean [22, 23] conducted a comparative study between CGP and MEP. The comparison was done on the basis of computation effort spent by CGP and MEP on two problems that are well-known benchmark instances used for assessing the performance of the algorithms evolving circuits i.e. two-bit multiplier and two-bit adder with carry. The results of the numerical experiments show that MEP outperforms CGP on some of the considered test problems. In some cases the MEP was found to perform better than CGP with more than one order of magnitude. Furthermore, Oltean [23, p. 7] draws attention to some significant differences between MEP and CGP. The differences show a significant advantage to the MEP over the CGP. A detailed discussion on strengths and weaknesses of several linear genetic programming techniques, including CGP, MEP and GEP can be found in [22].
An improved version of MEP i.e. IMEP was introduced in [11]. The improvement involved rearranging the nodes of the original MEP representation. To improve efficiency, in IMEP all terminals were kept in the first positions (genes) and no other genes containing terminals were allowed in the rest of the chromosome. The IMEP representation of the chromosome shown in Figure 2.1 would then be as follows:

\[
\begin{align*}
0: & \quad b \\
1: & \quad c \\
2: & \quad a \\
3: & \quad \& 0, 1 \\
4: & \quad \& 2, 3 \\
5: & \quad | 3, 4
\end{align*}
\]

Figure 3.4: Prefix IMEP representation of the chromosome in Figure 2.1

In addition to this new representation, another mutation function was added to allow the replacement of the worst individual in the population with the worst mutated individual, reason being that sometimes the worst individual may contain good genes to be exploited, so by mutating this individual its fitness may improve and therefore it will have a better chance to be selected. On the basis of the experiments performed, the IMEP outperformed the MEP and CGP [11].

Yan, Wei, Liang, Hu and Yao [33] implemented an improved GEP for electronic circuits using CGP. A representation was adopted in which a chromosome was represented as an \( n \times m \) geometry of uncommitted logic cells with inputs, outputs and netlist numbering of integers that are mapped to directed graphs rather than trees. Based on the outcome of case studies conducted (one-bit full adder, two-bit half adder and two-bit full adder), this version of GEP produced optimum circuits and, most importantly, guaranteed the populations diversity and hence causing the population not to trap into the local optima.

Furthermore, a new technique for evolutionary design of digital circuits by way of GP with subtree mutation was proposed in [3]. Subtree mutation replaces a randomly selected subtree with another randomly created subtree [15, p. 106]. In this technique a mutation point is chosen and the subtree connected to that point is removed and it is then replaced with a newly generated subtree. The proposed technique, helps to simplify and speed up the process of designing digital circuits, discovers a variation in the field of digital circuit design where optimised digital circuits can be successfully and effectively designed [3]. The results obtained using this technique demonstrate the potential capability of genetic programming in digital circuit design with limited computer programs.

AL-Saati and AL-Assady performed a thorough assessment of GEP in [1]. A number of some inherent weaknesses with GEP in its original form as proposed by Ferreira were identified and potential solutions were proposed.
The weaknesses identified include the problem of:

a. choosing the best parameter settings,

b. using only one linking function,

c. gene flattening,

d. illegal operations in genes and

e. lack of function biasing.

The first two drawbacks were successfully addressed by applying the so called multi-population feature to the GEP. The feature involves the creation of numerous populations (P) of a given size (S) with G number of generations. The use of multi-population features enables the GEP to use different settings for each population and can therefore reduce the parameter-setting problem. Similarly, each population has its own local linking function.

The problem of flat genes is avoided by imposing some monitoring process on the application of the IS operator, so that, when the number of functions in the head is zero, an emergency mutation is forced after that IS operator to ensure that the existence of a function in the head of the modified gene. Similarly, the illegal operations in genes are avoided by adding an emergency mutation in the fitness calculation such that when an invalid operation is about to abort the fitness calculation, the operation is simply mutated to one of the remaining functions in the function set. The case studies conducted in [1] found that, to the extent possible, all these improvements enhanced the rate of successful runs.
Chapter 4

Results

Both IGEP and the standard GEP algorithms were implemented to allow comparisons of the results for some of the known problems in the area of evolutionary algorithms. The results were compared with the aim of evaluating the performance of the enhanced algorithm i.e. IGEP. Both algorithms were applied to the fundamental logic expressions (AND, OR, NOT, NAND, XOR and NOR), one-bit HA, AND-OR ALU and one-bit FA using the NAND gate. As discussed in [17, p. 85] the NAND gate possesses a special property: it is universal. That is, given enough gates, it is able to mimic the operation of any other gate type. For example, it is possible to build a circuit exhibiting the OR function using three interconnected NAND gates. The ability for a single gate type to be able to mimic any other gate type is one enjoyed only by the NAND and the NOR. This property of NAND and NOR gates is very important because, from a manufacturing point of view, it is a lot cheaper in practice to manufacture a large number of similar gates than a large number of different gates. It is for this reason that digital control systems have been designed around nothing but either NAND or NOR gates, all the necessary logic functions being derived from collections of interconnected NANDs or NORs [17, p. 85].

The comparison of the results from GEP and IGEP was done on the basis of the fitness (i.e. optimality and functionality) of the final logic circuits produced as well as the success rates of the two algorithms. The success rate is defined as the proportion of trials or runs in which the termination criterion is met [34]. In line with this definition, the success rate in this dissertation is defined as the proportion of the number of correct solutions obtained to the total number of runs.

As mentioned the NAND gate is universal and the advantage of this property was discussed. As proof of the universality property, in this section it is shown how all the basic gate types, the HA, the AND-OR ALU and the FA were formed using only NAND gates for two reasons:

- To investigate the potential efficiency of using one type of logic gate by exploiting the universality property of the NAND gate.
- To test the efficiency of the IGEP program.

Table 4.1 below shows the parameters used in testing and comparing the performance of GEP and IGEP when applied on fundamental logic expressions, HA, AND-OR ALU and FA using the NAND gate. The symbol “+” in the examples below denotes the NAND operation.
### Table 4.1: GEP and IGEP parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head length</td>
<td>8</td>
</tr>
<tr>
<td>Number of genes</td>
<td>4</td>
</tr>
<tr>
<td>Root transposition of IS elements rate</td>
<td>0.10</td>
</tr>
<tr>
<td>Two-point recombination rate</td>
<td>0.23</td>
</tr>
<tr>
<td>One-point recombination rate</td>
<td>0.70</td>
</tr>
<tr>
<td>Gene recombination rate</td>
<td>0.23</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>0.05</td>
</tr>
<tr>
<td>Selection mechanism</td>
<td>Roulette wheel selection</td>
</tr>
<tr>
<td>Population size</td>
<td>100</td>
</tr>
<tr>
<td>Number of generations</td>
<td>500</td>
</tr>
<tr>
<td>Number of runs</td>
<td>50</td>
</tr>
</tbody>
</table>

### 4.1 Example 1: Designing logic circuits for basic logic gates

#### Table 4.2: Truth table for AND, OR, NAND, NOR, XOR and NOT

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

To exploit the universality property of the NAND gate, all the basic logic gates were implemented using the NAND gate only. Tables 4.3 and 4.4 below, show the results for all the basic logic functions obtained using IGEP and the standard GEP, respectively. Note that the bold gene represents the solution.
Table 4.3: Results for basic logic gates obtained using IGEP

<table>
<thead>
<tr>
<th>Logic gate</th>
<th>Fitness</th>
<th>Success rate (%)</th>
<th>Prefix Logic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>11+10001000100111</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0101+1+++110011101</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>100</td>
<td>01a11+0a0a01010a0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>100</td>
<td>bbbbbb1+bb00b10bb</td>
</tr>
<tr>
<td>NOT</td>
<td>1</td>
<td>100</td>
<td>+0+++++0a0001a01</td>
</tr>
<tr>
<td>NAND</td>
<td>1</td>
<td>100</td>
<td>+ab++bbaa0b10bb0b</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
<td>100</td>
<td>+ab1+1+110b0b1010</td>
</tr>
<tr>
<td>OR</td>
<td>3</td>
<td>100</td>
<td>+a10aaa010b1b0010</td>
</tr>
<tr>
<td>XOR</td>
<td>5</td>
<td>4</td>
<td>+ba00bbbaab1b00abb</td>
</tr>
<tr>
<td>NOR</td>
<td>4</td>
<td>34</td>
<td>+aaaaaa1+ba0bb11a</td>
</tr>
</tbody>
</table>

Prefix Logic Expression:
- 010AA1A+1101A0AA1
- A01+001+BABABB0AB
- aa01A+aA1aA00a0A
- B1bA+0+011bb001b
- aa10a1++AAA1A00A0B
- B+aaBBab0aAA0ab
- ++ab1+1aabbabba0a
- +A+1b11a0a1Aa0Aa
- ++b1b+ba0b10aaBB
- 1a0bAa+ba0b1Aaa
- +++0bb+B1abaA0a
<table>
<thead>
<tr>
<th>Logic gate</th>
<th>Fitness</th>
<th>Success rate (%)</th>
<th>Prefix Logic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>+1101+1+110110010</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1101+111010000100</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>100</td>
<td>10a0++00a011aaaa</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>100</td>
<td>00b0b+b1b1b0b0b0b0b0b</td>
</tr>
<tr>
<td>NOT</td>
<td>1</td>
<td>100</td>
<td>+1a0101aa11aa0aa0</td>
</tr>
<tr>
<td>NAND</td>
<td>1</td>
<td>100</td>
<td>+++101aa01b0a11a1</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
<td>100</td>
<td>++ab110ab100aa0b0</td>
</tr>
<tr>
<td>OR</td>
<td>4</td>
<td>98</td>
<td>++b+0b+a0a1a111b</td>
</tr>
<tr>
<td>XOR</td>
<td>5</td>
<td>2</td>
<td>++b+a1++baab00aa</td>
</tr>
<tr>
<td>NOR</td>
<td>5</td>
<td>32</td>
<td>++b+a1++aba1baa01</td>
</tr>
</tbody>
</table>

Table 4.4: Results for basic logic gates obtained using GEP
Comparing Tables 4.3 and 4.4 in terms of fitness (i.e. the same fitness function is used for both methods), it is clear that for simple problems the performance of IGEP is comparable to that of the standard GEP. However, for a slightly more complex problems i.e. OR and NOR, IGEP yielded a better solutions compared to GEP. Furthermore, the results show that the success rate of IGEP is about 2% more compared to GEP for the OR, NOR and XNOR logic functions. This is due to the extensive gene-reuse in IGEP as evident in the results. As expected, the level of gene-reuse in IGEP increases with the complexity of the circuit being evolved, see the bold genes in Table 4.3.

4.2 Example 2: Logic circuits for an AND-OR ALU and one-bit HA

In this example, the IGEP was used to derive the logic expressions for the AND-OR ALU and HA. We used 4 genes, each with a head length of 8. As in the previous example only the NAND gate was used. The AND-OR ALU performs the AND and OR logic functions. The design has two inputs, a and b, and one select bit (c) to select between the two functions, AND and OR.

<table>
<thead>
<tr>
<th>Table 4.5: Truth table for an AND-OR ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

The HA adds two single binary digits, a and b. It has two outputs, sum (c) and carry (d). The carry signal represents an overflow into the next digit of a multi-digit addition.

<table>
<thead>
<tr>
<th>Table 4.6: Truth table for one-bit HA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

The results presented in Tables 4.7 and 4.8 suggest that in terms of fitness, for more complex problems i.e. AND-OR ALU and HA, IGEP yielded better solutions compared to GEP with the exception of the SUM part of the HA. In fact GEP could not solve the AND-OR ALU. Again, the IGEP was found to be more successful in finding correct solutions than GEP, specifically for the AND-OR ALU and HA(Carry). These results prove that the performance of IGEP has been enhanced through gene-reuse.
Table 4.7: Results for an AND-OR ALU and one-bit HA obtained using IGEP

<table>
<thead>
<tr>
<th>Logic gate</th>
<th>Fitness</th>
<th>Success rate (%)</th>
<th>Prefix Logic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>AND-OR ALU</td>
<td>5</td>
<td>4</td>
<td>cbb1ac0ac11abcc</td>
</tr>
<tr>
<td>HA (Sum)</td>
<td>2</td>
<td>100</td>
<td>+abbbababbaa1bb11</td>
</tr>
<tr>
<td>HA (Carry)</td>
<td>4</td>
<td>12</td>
<td>++bab+1+aaba0b0ba0</td>
</tr>
</tbody>
</table>

Table 4.8: Results for an AND-OR ALU and one-bit HA obtained using GEP

<table>
<thead>
<tr>
<th>Logic gate</th>
<th>Fitness</th>
<th>Success rate (%)</th>
<th>Prefix Logic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>AND-OR ALU</td>
<td>1 002</td>
<td>0</td>
<td>+c+baccb11aaa00bc</td>
</tr>
<tr>
<td>HA (Sum)</td>
<td>2</td>
<td>100</td>
<td>++++ba11+babb0110b1</td>
</tr>
<tr>
<td>HA (Carry)</td>
<td>5</td>
<td>4</td>
<td>++++abbb++baa00abaa</td>
</tr>
</tbody>
</table>
4.3 Example 3: Logic circuits for one-bit FA

The FA circuit adds three one-bit binary numbers (a b c) and outputs two one-bit binary numbers, a sum (d) and a carry (e). As in the previous examples, the circuit was derived using the NAND gate only.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c</td>
<td>d e</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

The optimum logic expression for the SUM part of the FA was obtained using IGEP owing to the extensive gene-reuse as observed in Table 4.10. In this particular instance the IGEP was not successful in finding the solution for the CARRY part of the FA while on the other hand the standard GEP algorithm failed to solve both the SUM and CARRY of the FA as shown in Table 4.11.
### Table 4.10: Results for one-bit FA obtained using IGEP

<table>
<thead>
<tr>
<th>Logic gate</th>
<th>Fitness</th>
<th>Success rate (%)</th>
<th>Prefix Logic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>FA (Sum)</td>
<td>6</td>
<td>4</td>
<td>+1a+a10+ab11a01c1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>++bcA110bbAaccb0c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>++bB+acbc00AAB0cb</td>
</tr>
<tr>
<td>FA (Carry)</td>
<td>2006</td>
<td>0</td>
<td>++acb00+10cbb1111</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+c+1b++b0Aacba0cb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cc1ccB1bcBab1bceA</td>
</tr>
</tbody>
</table>

### Table 4.11: Results for one-bit FA obtained using GEP

<table>
<thead>
<tr>
<th>Logic gate</th>
<th>Fitness</th>
<th>Success rate (%)</th>
<th>Prefix Logic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>FA (Sum)</td>
<td>2000</td>
<td>0</td>
<td>cc0b0a0110ac1aaab</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1ab+a++0c01aa0b1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ac100abaac0cc0bc1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0+ca0c0bc1c0cb0</td>
</tr>
<tr>
<td>FA (Carry)</td>
<td>3002</td>
<td>0</td>
<td>+a+bc++b00110ba01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+b+bc++b00110ba01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+c++ +caca1acabcc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cc01aa00c1cb0b011</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusion

In this dissertation, two forms of genetic programming, GEP and MEP techniques, were explored in detail in order to demonstrate and understand the two techniques with the aim of integrating them. In line with the objectives of the study, this research has successfully integrated the features of MEP into the standard GEP algorithm to improve the performance and efficiency of GEP, notably the code-reuse functionality of MEP. As in MEP, the code-reuse in the IGEP was achieved through a representation which allows multiple references to a single sub-structure.

To assess the performance of IGEP, the IGEP algorithm was applied on nine known problems in the area of circuit design, ranging from simple to more complex problems i.e. AND, NAND, NOT, OR, XOR, NOR, AND-OR ALU, one-bit HA and one-bit FA. The performance of IGEP was compared with that of the standard GEP. These two techniques produced comparable results for simple problems such as basic logic functions. However, for complex problems such as one-bit HA(CARRY), FA(SUM) and AND-OR ALU the IGEP performed better than the standard GEP. Furthermore, the success rate of IGEP was found to be generally higher than that of GEP due to the gene-reuse capability implemented in IGEP.

As part of future work, I suggest we further explore and improve the robustness of IGEP such that it can solve any problem, even more complex problems such as the one-bit FA(CARRY). Furthermore, it would also be useful to develop an estimate for the optimum head length needed to successfully implement a given a logic circuit.
Appendix A

Implementation of the IGEP algorithm

The C++ program given in A.2 implements the IGEP. The program takes an input file and produces two output files. One can switch between IGEP and GEP by specifying “igep” or “gep”, respectively, when asked to choose between the two algorithms. The input and output files are described in Section A.1 below.

A.1 Input and output description

Table A.1 below summarises the structure of the input file by providing a description of what each line in the input file represents.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of states (rows) of the truth table</td>
</tr>
<tr>
<td>2</td>
<td>Number of columns (number of inputs and outputs) of the truth table</td>
</tr>
<tr>
<td>3</td>
<td>Number of inputs (terminals), excluding 0 and 1</td>
</tr>
<tr>
<td>4</td>
<td>Number of outputs of the truth table</td>
</tr>
<tr>
<td>5</td>
<td>Number of inputs (terminals), including 0 and 1</td>
</tr>
<tr>
<td>6</td>
<td>List of inputs (terminals), including 0 and 1</td>
</tr>
<tr>
<td>7</td>
<td>The seventh row onwards gives the truth table</td>
</tr>
</tbody>
</table>

As an example, the input file for the “AND” logic operator would look as follows:

4
3
2
1
4
ab01
000
100
010
111
Table A.2 below shows the list of logic operations considered in this dissertation. However, it should be noted that the results presented in this dissertation were produced using the “NAND” operation only but this program, with minor modification, can be executed using any operation or a combination of operations given in Table A.2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation (Logic gate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>NAND</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>XOR</td>
</tr>
<tr>
<td>$</td>
<td>NOR</td>
</tr>
</tbody>
</table>

There are two output files. One of these files is the main output file which contains the fittest chromosome in the population for each run. The file also gives, for each fittest chromosome, the fitness value as well as the iteration on which the chromosome was generated. Also provided in the file is the position of the gene which represents the solution and an indication of whether or not the algorithm was successful in finding the solution. The second output file contains the average fitness for each run.

A.2 The IGEP algorithm

```cpp
#include <sstream>
#include <cstdio>
#include <string>
#include <cstring>
#include <iostream>
#include <fstream>
#include <stack>
#include <vector>
#include <cmath>
#include <algorithm>
#include <ctime>
using namespace std;

ofstream outputFile;//define the output file (1st & last iterations)
ofstream oFile;//define the output file for avg fitness
string filename_out;//define the output file

//define control parameters
int head_length = 8,
    num_genes = 4,
    tail_length,
```
gene_length,
outputs, // number of outputs
popsize = 100,
max_generations = 500,
num_runs = 50,
um_replicates = popsize*0.1, // 10% of the population, including the cloning of best chromosome
num_terminals, // specify number of terminals
num_functions, // number of functions
table_rows, // specify number of rows for truth table
table_cols, // specify number of columns for truth table
first_output_col, // specify the input column
num_inputs, // specify number of inputs
fs_ter_gr_len,
gen_count;
bool *truth_table_pointer; // pointer to the truth table
bool mutate = true;
bool elitism = true;
double *avg_fitness;

string functions = "+*/!+-&->NAND &->AND !->NOT |->OR *->XOR $->NOR"
ref_genes = "ABCD", // referenced genes
fs_ter_gr, terms,
reuse; // this variable is used for switching between the Std GEP and GEP with reuse capability

// reuse = igep => invoke IGEP, otherwise invoke the Std GEP.

// random number generator (returns an integer between 0 and n-1
int rand_int (const int &n){
    return int(double(n)*rand()/RAND_MAX)%n;
}

// test if the input string is a function
bool isFunction(const char &f){
    for (int i = 0; i < num_functions; i++){
        if (functions[i] == f) {
            return true;
        }
    }
    return false;
}
//random number generator (returns a float random between 0 and 1
double randomOne(){
    return rand()/(double)RAND_MAX + 1;
}

#define a NAND function
bool nand(const bool &b1, const bool &b2){
    return !(b1 && b2);
}

#define a NOR function
bool nor(const bool &b1, const bool &b2){
    return !(b1 | b2);
}

#define an XOR function
bool xorr(const bool b1, const bool b2){
    return !(b1 == b2);
}

//function to convert a CHAR to STRING
string Char_to_String (const char &ch){
    stringstream str1;
    string str;
    str1 << ch;
    str1 >> str;
    return str;
}

//this function creates a gene.
string create_Gene_String(int gene_counter){
    string gene;
    int i = 0;
    if (reuse != "igep"){
        gene_counter = 0;
    }
    for (int h = 0; h < head_length; h++) {//head
        gene.append(Char_to_String(fs_ter_gr[rand_int(num_functions+num_terminals+
              gene_counter)]));//head
    }
    for (int t = 0; t < tail_length; t++) {//tail
        gene.append(Char_to_String(fs_ter_gr[num_functions+rand_int(num_terminals+}
// this function maps a CHAR to an INTEGER.
int backref_to_int(const char &x){
    return x - 'A';
}

// this function creates an expression tree based on a string
void create_exp_helper(const string &str, int str_len){
    s++;
    char symbol = str[s];
    if (isFunction(symbol)){
        if (symbol == '!'){
            exp_tree.append(Char_to_String(symbol));
            create_exp_helper(str,str_len);
        }
        else {
            exp_tree.append(Char_to_String(symbol));
            create_exp_helper(str,str_len);
            create_exp_helper(str,str_len);
        }
    }
    else {
        exp_tree.append(Char_to_String(symbol));
    }
}

// this function creates an expression tree given a string
string create_expr_tree(const string &str){
    s = -1; // reset the string index
    exp_tree.clear(); // reset the global vector: exp_tree
    create_exp_helper(str,str.size()); // create the expression tree from a string
    return exp_tree;
}

// this function maps a CHAR to an INTEGER.

vector<bool> chrom_values;
vector<bool> v;

//this function evaluates expression tree
bool evaluate_ET(const string &expression, const int &row){
    stack<bool> s;
    bool x, y, n; // a boolean to be pushed on the stack
    bool result; // a result after performing logic operation
    string exp = expression;
    reverse(exp.begin(),exp.end());
    const int &len = exp.size();
    for(unsigned int i = 0; i < len; i++){
        char c = exp[i];
        switch(c){
            case '0': s.push(false); break;
            case '1': s.push(true); break;
            case 'a': s.push(*((truth_table_pointer + table_cols * row + 0)))); break;
            case 'b': s.push(*((truth_table_pointer + table_cols * row + 1)))); break;
            case 'c': s.push(*((truth_table_pointer + table_cols * row + 2)))); break;
            case 'd': s.push(*((truth_table_pointer + table_cols * row + 3)))); break;
            case 'e': s.push(*((truth_table_pointer + table_cols * row + 4)))); break;
            case 'f': s.push(*((truth_table_pointer + table_cols * row + 5)))); break;
            case 'A': s.push(v[backref_to_int('A')]); break;
            case 'B': s.push(v[backref_to_int('B')]); break;
            case 'C': s.push(v[backref_to_int('C')]); break;
            case 'D': s.push(v[backref_to_int('D')]); break;
            case 'E': s.push(v[backref_to_int('E')]); break;
            case 'F': s.push(v[backref_to_int('F')]); break;
            case 'G': s.push(v[backref_to_int('G')]); break;
            case 'H': s.push(v[backref_to_int('H')]); break;
            case 'I': s.push(v[backref_to_int('I')]); break;
            case 'J': s.push(v[backref_to_int('J')]); break;
            case 'K': s.push(v[backref_to_int('K')]); break;
            case 'L': s.push(v[backref_to_int('L')]); break;
            case 'M': s.push(v[backref_to_int('M')]); break;
            case 'N': s.push(v[backref_to_int('N')]); break;
            case 'O': s.push(v[backref_to_int('O')]); break;
            case 'P': s.push(v[backref_to_int('P')]); break;
            case 'Q': s.push(v[backref_to_int('Q')]); break;
            case 'R': s.push(v[backref_to_int('R')]); break;
            case 'S': s.push(v[backref_to_int('S')]); break;
            case 'T': s.push(v[backref_to_int('T')]); break;
        }
    }
    return result;
}
    case 'U': s.push(v[backref_to_int('U')]); break;
    case 'V': s.push(v[backref_to_int('V')]); break;
    case 'W': s.push(v[backref_to_int('W')]); break;
    case 'X': s.push(v[backref_to_int('X')]); break;
    case 'Y': s.push(v[backref_to_int('Y')]); break;
    case 'Z': s.push(v[backref_to_int('Z')]); break;
    case '+':
        x = s.top();
        s.pop();
        y = s.top();
        s.pop();
        result = nand(x, y);
        s.push(result); break;
    case '&':
        x = s.top();
        s.pop();
        y = s.top();
        s.pop();
        result = x && y;
        s.push(result); break;
    case '|':
        x = s.top();
        s.pop();
        y = s.top();
        s.pop();
        result = x || y;
        s.push(result); break;
    case '*':
        x = s.top();
        s.pop();
        y = s.top();
        s.pop();
        result = xorr(x, y);
        s.push(result); break;
    case '$':
        x = s.top();
        s.pop();
        y = s.top();
        s.pop();
        result = nor(x, y);
        s.push(result); break;
    case '!':
        y = s.top();
        s.pop();
        result = !y;
        s.push(result);
    default: s.push(false); break;
}
```cpp
//this function removes duplicates in a string
string remove_duplicates(string & s1)
{
    int n = s1.size();
    for (int i = n-1;i != -1;--i)
    {
        for (int j=0;j<i;++j)
        {
            if (s1[i]==s1[j]){
                int k = i;
                while (k != n){
                    s1[k]=s1[k+1];
                    k++;
                }
            }
        }
    }
    return s1;
}
```
vector<int> g_complex;
int c_generation;
double c_fitness;
int c_fitness_index;
int c_matches;

//functions
void setChromosome(vector<string>);
void printChromosome();
void printChromosomeTree();

void Chromosome::setChromosome(vector<string> genes_){
    vector<string> gETs(num_genes);
    gene_values.clear();//ensure that there are no gene values before computations
    genes = genes_; //initialize the genes
    for(int j = 0; j <num_genes;j++){
        string str = create_expr_tree(genes[j]);
        gETs[j] = str;
    }
    g_expr_trees = gETs;
    vector<string>().swap(gETs);//release memory
    //for each state, calculate the value of each gene and store in gene_values for reuse in later steps
    for(int j = 0; j<table_rows;j++){
        v.clear();
        for (int i = 0; i < num_genes;i++){
            v.push_back(evaluate_ET(g_expr_trees[i],j));
        }
        gene_values.push_back(v);
    }
    //calculate fitness
    g_fitness.clear();
g_complex.clear();
g_matches.clear();
    for (int p = 0; p<num_genes; p++){
        int num_non_matches = 0;
        for (int j = 0; j < table_rows;j++){
            vector<bool> va = gene_values[j];
            //count the number of truth table values not matching the output
            if (va[p] != *(truth_table_pointer + table_cols * j + first_output_col)){
                num_non_matches++;
            }
        }
string gexp = g_expr_trees[p];
int num_functions = 0;
for (string::iterator i = gexp.begin(); i != gexp.end(); ++i) {
    if (isFunction(*i)) {
        num_functions++;//measures complexity
    }
}
g_complex.push_back(num_functions);
string str = remove_duplicates(gexp);
int rr = 0;
for (int w = 0; w < str.size(); w++) {
    int q = backref_to_int(str[w]);
    if (q >= 0 && q < 26) {
        rr = rr + g_complex.at(q);
    }
}
g_matches.push_back(table_rows - num_non_matches);
g_complex[p] = num_functions + rr;
//fitness = w_functions x number of functions + w_errors x non-matches
double w_functions = 1, w_errors = 1000;
g_fitness.push_back(w_functions*((double)(num_functions + rr)) + w_errors*((double)
    num_non_matches));
c_fitness_index = index_for_max_fitness(g_fitness, num_genes);
c_fitness = g_fitness[c_fitness_index];
c_matches = g_matches[c_fitness_index];
c_generation = gen_count;
}

void Chromosome::printChromosome() {
    outputFile << ref_genes[c_fitness_index] << " : ";
    for (int i = 0; i < num_genes; i++) {
        outputFile << genes[i] << " ";
    }
}

void Chromosome::prntChromosomeTree() {
    outputFile << ref_genes[c_fitness_index] << " : ";
    for (int i = 0; i < num_genes; i++) {
        outputFile << g_expr_trees[i] << " ";
    }
}
Chromosome *populace; // declare population of chromosome

Chromosome create_Chromosome()
{
    vector<string> c_;
    for(int i = 0; i<num_genes; i++)
    {
        c_.push_back(create_Gene_String(i));
    }
    Chromosome member;
    member.setChromosome(c_);
    vector<string>().swap(c_);
    return member;
}

// this function implements Roulette Wheel Selection method
int roulette_selection()
{
    int winner;
    vector<double> probs;
    vector<double> cumulative;
    double sum = 0.0;
    for (int i = 0; i < popsize;i++){
        sum = (double)(sum + (1/(double)(populace[i].c_fitness)));// take the inverse of the fitness value because the individual with smallest fitness is the fittest
    }
    for (int i = 0; i < popsize;i++){
        // Again, take the inverse of the fitness value because the individual with smallest fitness is the fittest
        probs.push_back(((double)(1/((double)populace[i].c_fitness)))/sum);  
    }
    for (int m = 0; m < probs.size(); m++){
        double sum_probs = 0.0;
        for (int p = 0; p <= m ; p++){
            sum_probs = (double)(sum_probs + probs.at(p));
        }
        cumulative.push_back(sum_probs);
    }
    double r1 = randomOne();
    int index1 = 0, index2 = 0;
    for (int w = 0; w < cumulative.size(); w++){
        if (cumulative.at(w) >= r1) {
            index1 = w; break;
        }
    }
```c
//this function calculates average fitness
double calc_avg_fitness(){
    double sum = 0;
    for (int i = 0; i < popsize;i++){
        Chromosome member = populace[i];
        sum = sum + (double)populace[i].c_fitness;
    }
    return sum/popsize;
}

//this function searches for the worst chromosome & replaces it with
//the newly created one provided it is better than the worst
void replace_worst_chr(Chromosome member) {
    Chromosome member1, member2;
    member1 = populace[0];
    double worst = member1.c_fitness;
    int index = 0;
    for (int m = 1; m < popsize; m++) {
        if (populace[m].c_fitness > worst) {
            worst = populace[m].c_fitness;
            index = m;
        }
    }
    if (member.c_fitness <= worst){
        populace[index] = member;
    }
}

// this function finds the fittest chromosome in the population
Chromosome find_fittest(){
    Chromosome fittest = populace[0];
    for(int i = 1; i < popsize; i++){
        if (fittest.c_fitness > populace[i].c_fitness){
            fittest = populace[i];
        }
    }
    return fittest;
}
```
//this function performs mutation
Chromosome Mutation(Chromosome cr){
    vector<string> genes = cr.genes;
    for (int g = 0; g < num_genes; g++){
        if (randomOne() < 0.05) {
            string rg = genes[g];
            int rr, gg = g;
            if (reuse != "igep"){
                gg = 0;
            }
            if (rand_int(gene_length) < head_length) {
                rr = rand_int(head_length);
                rg[rr] = fs_ter_gr[rand_int(num_functions+num_terminals+gg)];
            } else{
                rr = head_length+rand_int(tail_length);
                rg[rr] = fs_ter_gr[num_functions+rand_int(num_terminals+gg)];
            }
            genes[gg] = rg;
        }
    }
    cr.setChromosome(genes);
    return cr;
}

//this function performs Root transposition
void Root_transposition(){
    int winner1 = roulette_selection();
    Chromosome member = populace[winner1];
    if (randomOne() < 0.1) {
        vector<string> genes = member.genes;
        int r = rand_int(num_genes);
        string rg = genes[r];
        string head_ = rg.substr(0,head_length),
                        tail_ = rg.substr(head_length,tail_length);
        int pos = 1 + rand_int(head_length - 1);
        int flag = 0;
        while (pos > 0) {
            string ris_;
ris_.append(Char_to_String(head_[pos])); // construct the ris element

if (isFunction(head_[pos])){
    flag = 1; break;
}

pos--;
}

if (flag == 1){
    reverse(ris_.begin(),ris_.end());
    int ris_length = ris_.size();
    string su_ = head_.substr(0,head_length-ris_length);
    string du_ = ris_ + su_ + tail_;
    genes[r] = du_;
    member.setChromosome(genes);
}

if (mutate) {
    member = Mutation(member);
}

replace_worst_chr(member);

void OnePoint_recombination(){
    int winner1 = roulette_selection();
    // choose a random point in the chromosomes and exchange material
    Chromosome member1 = populace[winner1];
    int winner2 = roulette_selection();
    while(winner1==winner2){
        winner2 = roulette_selection();
    }
    Chromosome member2 = populace[winner2];
    vector<string> g1 = member1.genes;
    vector<string> g2 = member2.genes;

    if (randomOne() < 0.7){ // 1pt recombination rate = 0.7
        string chr1, chr2;
        for (int j = 0; j < num_genes; j++){
            string g_1 = g1[j], g_2 = g2[j];
            chr1.append(g_1);
            chr2.append(g_2);
        }
        int pos1 = rand_int(num_genes * gene_length),
pos2 = chr1.size();

swap_ranges(chr1.begin()+pos1, chr1.begin()+pos2, chr2.begin()+pos1);

for (int i = 0; i<num_genes;i++){
    string genes1, genes2;
    for(int k = gene_length*i; k < gene_length*(i+1); k++){
        genes1.append(Char_to_String(chr1[k]));
        genes2.append(Char_to_String(chr2[k]));
    }
    g1.at(i) = genes1;
    g2.at(i) = genes2;
}

member1.setChromosome(g1);
member2.setChromosome(g2);

if (member1.c_fitness > member2.c_fitness){
    member1 = member2;
}
if (mutate) {
    member1 = Mutation(member1);
}
replace_worst_chr(member1);

//this function performs 2 point recombination

void TwoPoint_recombination(){
    int winner1 = roulette_selection();
    Chromosome member1 = populace[ winner1 ];
    int winner2 = roulette_selection();
    while( winner1 == winner2 ){
        winner2 = roulette_selection();
    }
    Chromosome member2 = populace[ winner2 ];
    vector<string> g1 = member1.genes;
    vector<string> g2 = member2.genes;

    if (randomOne() < 0.233333333) { //2pt recombination rate = 0.233333333
        string chr1, chr2;
        for (int j = 0; j < num_genes; j++){
            string g_1 = g1[j], g_2 = g2[j];
            chr1.append(g_1);
        }
    }
}
chr2.append(g_2);
}

int pos1 = rand_int(num_genes * gene_length),
pos2 = rand_int(num_genes * gene_length);
if (pos1 > pos2){
    int temp = pos2;
pos2 = pos1;
pos1 = temp;
}

swap_ranges(chr1.begin() + pos1, chr1.begin() + pos2, chr2.begin() + pos1);
for (int i = 0; i < num_genes; i++){
    string genes1, genes2;
    for (int k = gene_length*i; k < gene_length *(i+1); k++){
        genes1.append(Char_to_String(chr1[k]));
genes2.append(Char_to_String(chr2[k]));
    }
g1.at(i) = genes1;
g2.at(i) = genes2;
}
member1.setChromosome(g1);
member2.setChromosome(g2);
if (member1.c_fitness > member2.c_fitness){
    member1 = member2;
}
if (mutate) {
    member1 = Mutation(member1);
}
replace_worst_chr(member1);

//this function performs Gene recombination within a chromosome
void Gene_recombination(){
    int winner1 = roulette_selection();
    // choose a random point in the chromosomes and exchange material
    Chromosome member1 = populace[winner1];
    int winner2 = roulette_selection();
    while(winner1==winner2){
        winner2 = roulette_selection();
    }
    Chromosome member2 = populace[winner2];
    vector<string> g1 = member1.genes;
vector<string> g2 = member2.genes;

if (randomOne() < 0.233333333) { // Gene recombination rate = 0.233333333
    int r = rand_int(num_genes);
    string random_gene1 = g1[r],
    random_gene2 = g2[r];
    g1[r] = random_gene2;
    g2[r] = random_gene1;
    member1.setChromosome(g1);
    member2.setChromosome(g2);
}

if (member1.c_fitness > member2.c_fitness) {
    member1 = member2;
}

if (mutate) {
    member1 = Mutation(member1);
}

replace_worst_chr(member1);
}

void print_Pop(Chromosome c[], int & length) {
    for (int m = 0; m < length; m++) {
        Chromosome member = c[m];
        member.printChromosome();
        outputFile << "\n---"d member.c_fitness"d member.c_generation<<endl;
    }
}

// this function sort the population in ascending order w.r.t fitness
// this function will assist in implementing the elitism algorithm
void sorting_function() {
    for (int i = 0; i < popsize; i++) {
        for (int j = 0; j < popsize; j++) {
            if (populace[i].c_fitness < populace[j].c_fitness) {
                Chromosome temp = populace[i];
                populace[i] = populace[j];
                populace[j] = temp;
            }
        }
    }
}

// this function performs replication and elitism
void Replication(){
    Chromosome *replicates;
    replicates = new Chromosome[num_replicates];
    for(int i = 0; i < num_replicates; i++){
        int winner = roulette_selection();
        replicates[i] = populace[winner];
    }
    if (elitism) {
        sorting_function(); // sort the population in ascending order such that the best chromosome are placed at the top
        int elite = popsize*0.05; // Apply elitism - transfer the top 5% chromosomes (unchanged) to the next generation
        for(int j=0; j<elite; j++){
            replicates[j] = populace[j];
        }
        for(int j = 0; j < num_replicates; j++){
            populace[j] = replicates[j];
        }
    }
}

//====================================================================

void next_Generation (){ Replication();
    int j = 0;
    while (j < (popsize - num_replicates)){
        if (j < (popsize - num_replicates)) {Root_transposition(); j++;};
        if (j < (popsize - num_replicates)) {OnePoint_recombination(); j++;};
        if (j < (popsize - num_replicates)) {TwoPoint_recombination(); j++;};
        if (j < (popsize - num_replicates)) {Gene_recombination(); j++;};
    }
}

//====================================================================

// this function returns a minimum or maximum element in a vector
int min_max(vector<int> v, string maxmin){
    if (v.size()==0){ // if there is no success
        return -1;
    }
    int aa = v.at(0);
    if(maxmin == "min"){
        for(int i=0; i<v.size(); i++){
            if (v.at(i) < aa){
                aa = v.at(i);
```cpp
    }
  }
  
  else{
    for(int i=0; i<v.size();i++){
      if (v.at(i) > aa){
        aa = v.at(i);
      }
    }
    return aa;
  }

  //====================================================================
  void createPopulation(){
    cout<<"------------------------------------------------------"<<endl;
    cout<<endl;
    cout<<"Please wait while GEP generates the initial population...!"<<endl;
    cout<<endl;
    
    for (first_output_col = num_inputs; first_output_col < (outputs + num_inputs);
        first_output_col++){
      vector<int> success_iter;
      int success_cnt = 0;
      double fittest_per_run[num_runs][max_generations];  // declare an array to save fitness value for best individual in every run and generation
      for(int qq=0; qq < num_runs; qq++){
        populace = new Chromosome[popsize];
        avg_fitness = new double[max_generations];
        for (int m = 0; m < popsize; m++){
          Chromosome member = create_Chromosome();
          populace[m] = member;
        }
        avg_fitness[0] = calc_avg_fitness();
        int p;
        Chromosome best_chromosome = find_fittest();
        fittest_per_run[qq][0]=best_chromosome.c_fitness;
        for(p = 1; p < max_generations; p++){
          cout<<p<<" : "<<calc_avg_fitness()<<endl;
          avg_fitness[p] = calc_avg_fitness();
          next_Generation();
          Chromosome temp = find_fittest();
          if (best_chromosome.c_fitness > temp.c_fitness){
```
best_chromosome = temp;

fittest_per_run[qq][p]=temp.c_fitness;
}
if (best_chromosome.c_matches == table_rows){
    outputFile<<"=============== Run number : "<<qq+1<<" ==============

    best_chromosome.printChromosome();
    outputFile<<"---- -> "<<best_chromosome.c_fitness" "<<best_chromosome.
    c_generation<<"(Solution found!)"<<endl;
    success_iter.push_back(best_chromosome.c_generation);
    success_cnt++;
    outputFile<<endl;
}
else {
    outputFile<<"=============== Run number : "<<qq+1<<" ==============

    best_chromosome.printChromosome();
    outputFile<<"---- -> "<<best_chromosome.c_fitness" "<<best_chromosome.
    c_generation<<"(No solution found!)"<<endl;
    outputFile<<endl;
}
}
outputFile<<endl;
int summ = 0;
for(int pp=0;pp<success_iter.size();pp++){
    summ = summ+success_iter.at(pp);
}
outputFile<"Average generation: "<<(double)summ/num_runs<<endl;
outputFile<"Minimum : "<<min_max(success_iter,"min")<<endl;
outputFile<"Maximum : "<<min_max(success_iter,"max")<<endl;
outputFile<"Success rate (%) : "<<100*success_cnt/num_runs<<endl;
vector<double> avgs;
for(int j=0;j<max_generations;j++){
    double sum_ = 0;
    for(int i=0;i<num_runs;i++){
        sum_=sum_+fittest_per_run[i][j];
    }
    avgs.push_back((double)sum_/num_runs);
}
for(int r = 0; r < max_generations; r++){
    oFile<avgs.at(r)<<"\n";
int main() {
    /*
    The input dataset is structured as follows:
    no. rows
    no. cols
    no. inputs
    no. outputs
    no. terminals, including 0 and 1
    data*/
    srand(time(0));
    ifstream inputFile;
    string filename_in;
    cout << "Please enter the name of your input file: ";
    cin >> filename_in;
    inputFile.open(filename_in.c_str());
    if (inputFile.fail()) {
        cout << "Input file could not be opened. Try again." << endl;
        return 1;
    }
    cout << "Please enter the name of your output file: ";
    cin >> filename_out;
    outputFile.open(filename_out.c_str());
    if (outputFile.fail()) {
        cout << "Output file could not be opened. Try again." << endl;
        return 1;
    }
    string avg_fit;
    cout << "Please enter the name of your avg file: ";
    cin >> avg_fit;
    ofstream oFile(avg_fit.c_str());
    if (oFile.fail()) {
        cout << "Avg file could not be opened. Try again." << endl;
        return 1;
    }
    cout << "Choose between IGEP and GEP - igep/gep?: ";
    cin >> reuse;
}
int rows = 0, inputs = 0, cols = 0, num_terms;
inputFile >> rows;
inputFile >> cols;
inputFile >> inputs;
inputFile >> outputs;
inputFile >> num_terms;
table_rows = rows;
table_cols = cols;
um_inputs = inputs;
num_terminals = num_terms/*-2*/; /*exclude 0 and 1 from the terminal set*/
num_functions = functions.size(); //number of functions

if(inputFile.eof()){
    cout << "Error reading input file contents." << endl;
    return 1;
}
string *names = new string[table_rows + 1];
bool *values = new bool[table_rows];

for(int i = 0; i < table_rows + 1; i++){
    inputFile >> names[i];
}

terms = names[0];
string terms_excl_1_0 = terms.substr(0, terms.size()-2);
fs_ter_gr = functions + terms/*terms_excl_1_0*/;

if (reuse == "igep"){
    fs_ter_gr = fs_ter_gr + ref_genes.substr(0, ref_genes.size()-1);
}
fs_ter_gr_len = fs_ter_gr.size();
bool table[table_rows][table_cols];

for (int m = 1; m < table_rows + 1; m++) {
    string binaries = names[m];
    for (int n = 0; n < cols; n++) {
        if (binaries.at(n) == '0') {
            table[m - 1][n] = false;
        } else{
            table[m - 1][n] = true;
        }
    }
}
truth_table_pointer = &table[0][0];
first_output_col = num_inputs;

int arity = 2;
tail_length = head_length * (arity - 1) + 1;
gene_length = head_length + tail_length;

outputFile << "Head size: " << head_length; outputFile << "\n";
outputFile << "Arity: " << arity; outputFile << "\n";
outputFile << "Tail size: " << tail_length; outputFile << "\n";
outputFile << "Gene size: " << gene_length; outputFile << "\n";
outputFile << "Terminals & functions: " << fs_ter_gr_len; outputFile << "\n";
outputFile << "Number of genes: " << num_genes; outputFile << "\n";
outputFile << "# Table rows: " << table_rows; outputFile << "\n";
outputFile << "# Table cols: " << table_cols; outputFile << "\n";
outputFile << "Logic function: " << filename_in; outputFile << "\n";
outputFile << "---------------------------------------------------------------------";

outputFile << "\n";
outputFile << endl;
createPopulation();
outputFile.close();
oFile.close();
}
Bibliography


