On the Translation of Linear Genetic Programs into Tree Genetic Programs and Vice Versa

Robyn Ffrancon[1]*

Abstract
In this project two schemes were introduced for translating from LGP to TGP and vice versa. As a result of these translation schemes intron instructions could be identified in LGP programs. By translating TGP programs, LGP programs could be formed which were composed entirely of exon instructions.
It was shown that intron instructions were formed from applying the crossover mechanism to a population of purely exon LGP programs over successive generations. After around five generations approximately 80% of the instructions within the population were introns. This percentage of introns has been observed in applying LGP to classifying medical data and also within the DNA sequences of some biological organisms.
A method of parallelising the execution of a single LGP program was outlined. It was estimated that the probability of decreasing the computation time of executing a 25 exon instruction program at least somewhat was between 87% and 90%.

Keywords
Linear Genetic Programming — Tree Genetic Programming — Introns — Parallel Processing

[1] *University of Gothenburg, Sweden*
***Corresponding author**: rffrancon@gmail.com

Contents

1. Introduction

Linear Genetic Programming (LGP) and Tree Genetic Programming (TGP) are two well known and well studied forms of Genetic Programming (GP). GP algorithms are Evolutionary Algorithms where the methods for finding a satisfactory answer to some problem are inspired from biological evolution. The extent to which biological evolution inspires the practical implementation of GP is limited by the computational architecture available and the complexity of biological evolution itself. Most popular evolutionary algorithms that have been inspired from biological evolution can, in their most simplistic form, be understood intuitively. A good example would be a simplistic, yet practically useful, TGP scheme.

The practical limitations of imitating the complex processes of biological evolution have contributed to the design of evolutionary algorithms that contain some purely artificial features. Parallel Linear Genetic Programming (PLGP) is an example of a scheme which contains purely artificial features; it will be discussed later on.

In general, GP is a method of generating a computer program for solving a particular problem. But instead of a programmer explicitly composing a program using a programming language, a ”population” of programs undergo evolution.

Biological evolution is a complicated subject. Wikipedia gives an an eloquent definition of evolution which, although incomplete, is useful in and of itself for our purposes: “Evolution is the change in the inherited characteristics of biological populations over successive generations.” We can expand on this by noting that genetic material is the information which is inherited by successive generations. Additionally, the characteristics of individuals within biological populations are expressions of genetic material. Crudely, we could say that evolution is the optimisation of a population’s genetic material over successive generations. The objective of this optimisation is to improve the ability of the individuals in the next generation to propagate their genetic material into future generations.

The environment of an individual is its restrictive adversary to the successive propagation of its genetic material. The environment can effectively be seen as the measuring tool against which the performance of an individual’s genetic material is measured. In other words, individuals in a biological environment are evaluated based on their ability to generate the most optimised genetic material.

In GP a population of programs (individuals) are evaluated based on their problem solving abilities. This is analogous to the evaluation of an individual based on it's ability to cope with it's environment whilst attempting to propagate it's genetic material. In GP, once each individual has been evaluated the next generation of offspring is generated. In forming the next generation a positive correlation exists between the probability of reusing the genetic material of an individual and the performance of an individual.

For our purposes, GP is a method of finding an unknown mathematical function which is composed of the operators: addition, subtraction, multiplication, and division. An individual's fitness (performance) is simply the inverse of an error evaluation based on the differences between an individual's proposed mathematical function and the actual mathematical function that is to be found. In other works GP techniques have been used extensively as classifiers and optimisation methods [1, 2].

LGP and TGP are two different methods of expressing and evolving programs for GP. Both schemes share evolutionary methods which are inspired from the same biological evolutionary processes. One example of such an evolutionary process is crossover, which will be discussed later on. Both schemes have some evolutionary methods which are procedurally similar. However, the resultant effect of their implementation in both schemes is usually very different.

We will briefly recap some of the basic principles involved with TGP and LGP which are important for our purposes. Literature such as Wahde (2008) give a more in depth and thorough overview of both GP schemes.

1.1 Tree Genetic Programming

TGP is probably the most commonly used GP technique. Often when people refer to GP they mean TGP. In many ways TGP is much simpler than LGP. TGP lacks some of the subtleties present in LGP such as structural introns which will be discussed later.

The TGP method uses a forest (population) of trees (programs, also called chromosomes). A single population is known as a generation. Using evolutionary inspired methods (described below) subsequent populations are generated, these are the subsequent generations. For our purposes each new generation will contain a new forest derived from the old forest. Generations will be generated within a single discrete time step.

TGP trees (programs) for our purposes are best represented as tree structures composed of a single node of degree two (root node), some nodes of degree one (leaf nodes), and usually some nodes of degree three (tree branch nodes).

Figure 1 shows an example of a single TGP tree program. The red node is the root node, the green nodes are the leaf nodes, and the light blue nodes are the tree branch nodes. Directed edges exist between the nodes. The blue and red nodes are each labeled by some operator from the set $\{+,-,/,*\}$.

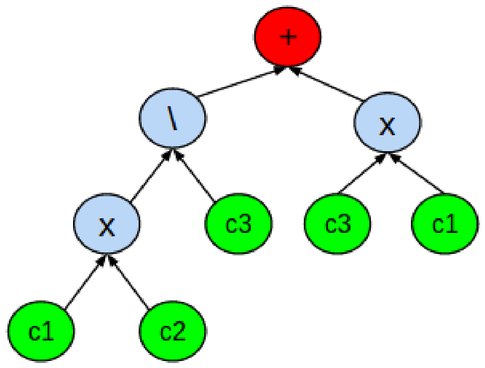

Figure 1. This figure shows a tree graph representation of a simple TGP program. The red node is the tree's root node, the light blue nodes are branch nodes, and the green nodes are the leaf nodes. The root and branch nodes have been labeled with operators. Each leaf node has been labeled with a constant memory register.

The green leaf nodes are labeled by the names of constant memory registers: $c_1, c_2, c_3 \ldots$. The value of these constant memory registers are used as operands. These constant memory registers are set once before the start of the program execution (evaluation) and are not modified during execution.

In order to understand the method of executing a TGP tree, consider a node p which is not a leaf node. Node p is the target node of two directed edges where the source nodes of those edges are the children, c_1 and c_2, of node p. Suppose that c_1 and c_2 are leaf nodes. The tree can be evaluated at node p by applying the operator associated with node p on the operands associated with nodes c_1 and c_2. Note that when the operator is *division* the "left" child node gives the numerator and the "right" child node gives the denominator. Memory is associated with node p so that the "left" and "right" status of the child nodes are noted and are not changed once they are first set. If the denominator is zeros when the division operator is used in the tree the performance (fitness) of the entire tree is set to zero (lowest possible performance).

Once the arithmetics derived from node p and its child nodes have been calculated the resultant answer is kept in memory associated with node p. This answer will be used as an operand if node p itself was a child node.

At this point it should be clear how the entire tree can be evaluated with the calculation associated with the red root node performed last. The first calculations to be made are of those nodes which have two leaf nodes as children.

Within the context of TGP, *mutation* is an evolutionary method where the operand or operator of a node within a tree is changed probabilistically. In the simplest case, a *mutation probability* is defined and each node is mutated with this probability. If a non-leaf node is to be mutated a random operator is chosen from the possible set of operators. If a leaf node is to be mutated, it's associated constant memory register is changed to another random constant memory register from the set of all constant registers.

Crossover is another evolutionary method which is essential to TGP. For our purposes we will only consider the simplest form of TGP crossover. During crossover in TGP two trees are selected. Then, one node from each tree is selected at random, we will name these nodes n and m. Finally, the edge between node n and it's parent (the edge is directed towards the parent) is removed. An edge is created between node n and the parent of node m. Similarly, the edge between node m and it's parent is removed and an edge is subsequently created between m and the previous parent of node n. In this way the nodes of the TGP tree can move between separate trees.

The processes of mutation and crossover effectively create new trees. This new forest will constitute the next generation. During the creation of a new generation the trees of the old generation are not overwritten or deleted. This means that a tree can be used twice in forming the new generation with crossover.

In describing the crossover method above it was said that two trees are "selected". The method of selection is called *tournament selection*. Tournament selection is used so as to increase the probability of using high fitness trees in generating the trees of the next generation.

1.2 Linear Genetic Programming

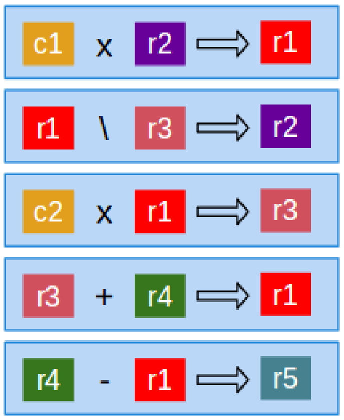

Figure 2. This figure shows a diagram of a simple LGP program. Each blue box represents a separate instruction. Within each blue box two operand registers are seen to the left of the arrow along with an operator. The arrow points towards the memory register which will store the resultant value of the left hand calculation. Each instruction would be executed in order from top to bottom.

Figure 2 shows a depiction of a simple linear genetic program. For our purposes, a single program in LGP is composed of an ordered list of instructions. Each instruction includes: references to three memory registers, and an operator from the set $\{+,-,/,*\}$. As with the TGP scheme, the LGP scheme uses populations of programs which are separated into generations.

There are two possible types of memory registers in the LGP scheme, variable memory registers and constant memory registers. A single instruction within an LGP program deals with three memory registers. The contents of two of the three memory registers are used as operands, in analogy with the child nodes in a TGP program. The remaining memory register is used to store the answer of the arithmetic calculation. In the figure an arrow within each instruction points towards this variable memory register. We will call this memory register the *destination* memory register.

There are no restrictions on which memory registers can be used as operands, these registers may be the same or different, of constant or variable type. However, only a variable type memory register can be used to store the resultant value of the arithmetic calculation.

In the simplest scheme for the execution of a single program, each instruction is executed in order. Before the execution of any program within a population the contents of the constant and variable memory registers are set. The values stored in the constant memory registers do not change once set. The initial values of all memory registers before the execution of any program is always the same.

The constant memory registers can be used to store any values which are suspected to be important in calculating the correct desired answer. These values will not be overwritten and are therefore not lost during the execution of a program. The values of the variable memory registers are usually initially set to random values. There is much to say about choosing the number of registers and setting their initial values.

At this point, it should not be difficult to see that a LGP program modifies the values stored within registers during the course of it's execution. The final value given by a program is read from the same variable memory register for any given program. This is usually the first variable memory register: register 0. The fitness (performance) of a program is based on this value similar to the TGP scheme.

A simple implementation of the mutation evolutionary method within the context of LGP is similar to that seen in TGP. A mutation probability is set which gives the probability of modifying which register is used by an operand, the operator used, or which register is used as the destination register.

In this project the crossover mechanism used in the LGP scheme is two point crossover. In performing two point crossover the first step is to select two programs using tournament selection, we will refer to these two programs as A and B. Secondly, two random instruction are chosen within each program. We will refer to the instructions chosen within program A as a_1 and a_2; similarly for program B as b_1 and b_2. Thirdly, two new chromosomes are created for the next generation by copying programs A and B but swapping all of the instructions between and including instructions a_1 and a_2

with all of the instructions between and including b_1 and b_2.

In this way a new generation of chromosomes (programs) can be generated. Further more, due to the use of tournament selection it is likely that the chromosomes of the new generation will include a disproportionate proportion of the instructions from the best performing individuals of the current generation.

2. Translation

2.1 Translation from TGP to LGP

Now that the basics of TGP and LGP have been outlined, we can discuss some similarities and differences between them. These differences and similarities are best highlighted when translating programs from one scheme to another.

In its most basic form the scheme for translating TGP programs to LGP programs is very simple. Firstly, an instruction should exist in the complementary LGP program for each arithmetic calculation that is performed in executing a TGP program.

Consider again Fig. 1, but supposed we removed all of the green leaf nodes. The remaining network would represent the instruction dependency network of the complementary LGP program. In this new network nodes would represent instructions and directed edges would represent instruction dependancies. For a given directed edge, the target node would represent an instruction which would depend on the instruction represented by the source node. An instruction may well only immediately depend on one other instruction, in which case the second operand would be provided by a constant memory register and not from a calculation. Of course, the only other case is that both operands of an instruction would be provided by instructions. This is the trivial case of a single instruction program.

In an LGP program the variable memory registers are used in transferring information for processing from one instruction to another. This allows a program to compound it's calculations. A TGP program does not make use of memory registers in the same way as LGP programs. In fact, TGP programs only make use of a set of constant values which are set before the beginning of executing a TGP program and are not changed afterwards.

A TGP program does not use any variable memory registers. What then is the best method of assigning variable memory registers to a TGP program during its translation into a LGP program? One simple scheme would be to have as many variable memory registers as calculations. In this way the result of any calculation would be stored in a variable memory register and used again during any dependant calculations. Each register would be written to and read from once. One drawback of this approach is that a large amount of memory would be use. Another drawback of this scheme is that it does not provide flexibility in the number of different registers used. The number of registers used is completely dependant on the number of instructions (calculations).

An alternative scheme would be to use the minimum possible number of different memory registers whilst preserving the functionality of the original TGP program. This scheme would allow additional memory registers to be used at the users discretion. Alternatively, other memory registers could become functional later from mutation. This formation would present the LGP equivalent program of the TGP program in it's most concise memory form. In this project an algorithm was devised for assigning the minimum number of different variable memory registers to a TGP program.

The algorithm is best thought of in two parts. The first part leaves a 'trace' on each node in the TGP tree that is stored as a node property. The 'trace' information allows the second part of the algorithm to identify the order of assigning memory registers to the nodes.

The 'trace' information stored in each node is an array of two elements. Figure 3 shows a flowchart that describes how the 'trace' is calculated for each node in the TGP program network. The first step in setting the 'trace' is to calculate the depth of each leaf node. The depth of a node is the minimum number of edges between itself and the root node. That is to say, the depth of a node is the shortest edge path length between itself and the root node. The root node has a depth of zero.

Initially, the elements of the 'trace' array of every node are set to zero. The first element of the 'trace' array will be referred to as the 'max depth' and the second element will be referred to as the 'cumulative depth'. Once the leaf node depths have been calculated the next step is to process each leaf node.

I will refer to the current unprocessed leaf node under consideration as the node L. For a general node N, the parent of node N will be denoted as $P(N)$ and the grand parent of node N will be donated as $P(P(N))$ and so on. Node C is the current node under consideration.

The first unprocessed leaf node L is selected and the elements of it's 'trace' array values are both set to the leaf node's depth. We set the current node under consideration to the leaf node, $C = L$. Next the node $P(C)$ is considered, at this point node $P(C)$ is the parent of the leaf node. If it's 'max depth' value is greater than the 'max depth' of node C, the 'max depth' of node $P(C)$ is set to the 'max depth' of node C. Otherwise the 'max depth' value of node $P(C)$ remains unchanged. The 'cumulative depth' of node $P(C)$ is increased by the depth of the leaf node L (not necessarily node C). Finally, the current node under consideration is set to $C = P(C)$. The process is repeated with the modification of node $P(C)$ until C is the root node and no longer has a parent.

Figure 4 shows a flowchart of the second part of the algorithm which assigns the minimum number of variable memory registers to a TGP program. Note that a variable memory register may provide an operand for an instruction and also serve as a destination register for the same instruction.

The second part of the algorithm starts by ordering the nodes of the network into a list A which is sorted by depth from minimum to maximum. The root node is at the minimum depth. Initially, the chosen variable memory register of each node is set to 1. The current node C is chosen as the first unprocessed node in the list A. The child nodes of node C will be referred to as node X and node Y. If node C has no children it is simply marked as having been processed. In general the variable memory register assigned to the node N is given by $R(N)$.

If the 'max depth' of node X is greater than the 'max depth' of node Y we set $R(X) = R(C)$ and $R(Y) = R(C) + 1$ and the node C is marked as processed. Alternatively, if the 'max depth' of node X is smaller than the 'max depth' of node Y we set $R(X) = R(C) + 1$ and $R(Y) = R(C)$. If the 'max depth' values of nodes X and Y are equal we compare their 'cumulative depth' values. If the 'cumulative depth' of node X is greater than or equal to the 'cumulative depth' of node Y we set $R(X) = R(C)$ and $R(Y) = R(C) + 1$. Otherwise we set $R(X) = R(C) + 1$ and $R(Y) = R(C)$.

At this point the node C is marked as processed and a new node C is chosen as the first unprocessed node from the list A. The new C node is processed as described above and the next C node is chosen until there are no more unprocessed nodes in the list A.

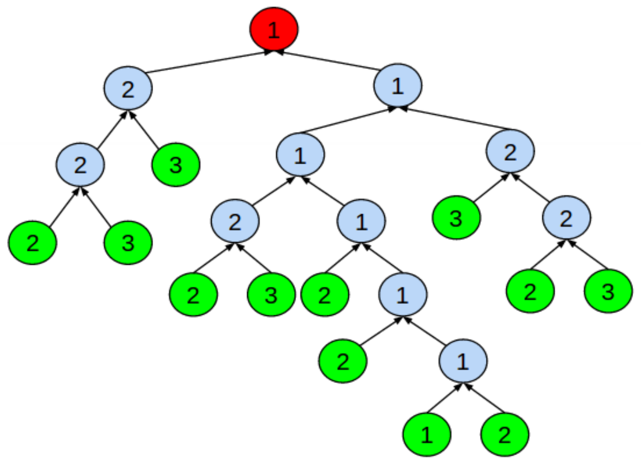

Figure 5. This figure shows a tree representation of a TGP program. The nodes have been labeled by the memory registers assigned to them during the TGP to LGP translation process.

Figure 5 shows an example tree where the minimum number of variable memory registers have been assigned. In this figure the variable memory registers are numbered from 1 to 3 and the nodes are labeled by their assigned memory registers. The green leaf nodes have also been assigned variable memory registers by the algorithm. The final step in assigning memory registers to the TGP program is to change the memory registers of the leaf nodes from variable type to those of constant type. This is easy to do as the original TGP program already had constant memory registers assigned to the leaf nodes that can be reused.

Having assigned memory registers to the TGP program, the final step in the translation process is to construct and order instructions for the LGP program. The algorithm for determining the order of the instructions makes use of which memory registers were assigned to which nodes in the TGP tree. The algorithm begins by finding the unprocessed node of maximum depth starting from the root node (the root node has the minimum depth). In moving down the tree from parent to child, the unprocessed child node is chosen first. If both child nodes are unprocessed, priority is given to the child node which has the lowest memory register ID number. If a node has no children it is simply marked as processed and the next node is searched for. If an unprocessed node is found where both child nodes have been processed an instruction is formed using the assigned memory register information of the parent and two child nodes in addition to the operator information associated with the parent node. This instruction is appended to the chromosome and the parent node is marked as processed. This process is repeated until all nodes are processed. The root node will be the final node to be processed.

2.2 Translation from LGP to TGP

Unlike LGP programs, TGP programs do not use variable memory registers to facilitate the flow of information through a program. Instead, the flow of information through a TGP program is implicit in it's tree like structure. Therefore, in order to translate a LGP program into a TGP program we must examine how information flows through that LGP program. In analysing the LGP program we must form a tree structure which would facilitate the transfer of information through the equivalent TGP program.

In investigating the information flow in a LGP program we are only concerned with access to writing or reading memory registers. We purposely neglect whether the act of writing to a memory register actually changes its value. As a result, the information flow network that is produced from this analysis will not depend on the operators of any instructions or the initial value of any memory register. This is important as we do not know how mutation might change the instruction operators. Additionally, we do not know how the initial values of the memory registers will be set.

Figure 6 shows the memory register access network of a LGP program. Each node signifies a memory register access event (read or write) within the LGP program. Directed arrows exist between the nodes. For any particular edge, the source node represents a read event and the target nodes represents a write event.

When a memory register is read it's stored value is being used as an operand by an instruction. A writing operation on a memory register means that an instruction is storing the resultant value of some calculation in that memory register.

The nodes of the network are labeled by the memory register ID and the instruction ID in which the access event occurs (in brackets). The instructions of an LGP program are ordered and their ID values are simply incremental integers

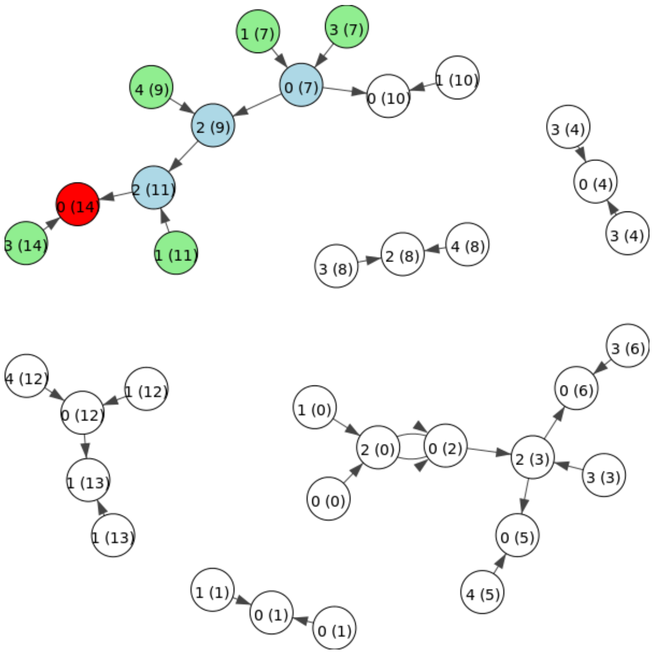

Figure 6. This figure shows the memory register access graph for an LGP program. The white nodes are intron instruction access events. The coloured nodes are exon instruction access events. For a given directed edge, the source node is a register read event and the target node is a register write event. The fitness value of the LGP program is derived from only considering the value stored in the 0 memory register.

starting from 1. Some nodes have outgoing and incoming edges. These nodes represent memory registers which were written to and then read from. The bracketed instruction ID value for these nodes are the last instructions to write to the associated memory registers. The two pieces of information which are used in labeling the nodes are sufficient for uniquely identifying the memory register access events.

Sometimes a single node may have multiple outgoing edges. These source nodes represent multiple read events which have occurred for their associated memory registers. Every node has either two inbound edges or none. This is because all instructions use exactly two operands. Some nodes have two outgoing edges to the same target node. This corresponds to an instruction which uses the same memory register for both operands.

Before executing an LGP program memory register(s) are designated as the *final read out register(s)*. The number of final read out registers is the same as the number of objective function outputs. In this project the objective function had a single output. Register 0 was chosen as the final read out register for all LGP programs in every generation. After the execution of an LGP program is completed the value stored in the final read out register is used to evaluate the performance of the LGP program using the method that has been previously described.

In Fig. 6 we can see that the final read out register has been written to several times in several different subgraphs. This means that the content of the final read out register has been overwritten several times. We are interested in the final value stored in register 0 after the execution of the LGP program finishes. The instruction ID value of each node determines which node represents the last write event to register 0. Of the nodes which write to register 0, the node with the highest instruction ID value signifies the last write event. We will name this node the final read out node.

If an LGP program does not contain a final read out node there does not exist an equivalent TGP program.

Once the final read out node has been determined, we can gather the nodes which will be used in generating the final TGP program. These nodes will be gathered in the unique set S. The procedure for gathering these nodes is similar to a breadth-first search [3]. The search starts at the final output node and moves along the edges from neighbour to neighbour gathering nodes. We restrict our movements along the edges so that we only ever move in the opposite direction of an edge.

The coloured nodes in Fig. 6 are the nodes of set S for this particular LGP program. We have seen examples of TGP trees with this colour scheme before. The red node is the final read out node and is equivalent to the root node of a TGP tree. The light blue nodes are equivalent to the branch nodes, and the green nodes are equivalent to the leaf node. The rest of the nodes in the figure are coloured white and are not present in the set S. These nodes are ignored by the translation scheme and can be deleted from the graph.

It is easy to see how the subgraph formed by the colour nodes can correspond to a TGP tree. The functionality of the registers is replaced by the use of directed edges. The leaf nodes can take on constant memory registers whose values correspond to the values of the variable memory registers previously assigned to them. Each node has an associated instruction and therefore operator. If a node signifies a write access event then it's label can be changed to it's associated operator.

In this way the corresponding TGP tree of a LGP program can be created. However there is one caveat which we will address using Fig. 7. This figure shows an example memory access network of a different LGP program where all nodes absent from the set S (white nodes) have been removed. In this figure we can see that the node labeled "2 (4)" has two outgoing edges. But, in a TGP tree each node has one outgoing edge with the exception of the root node which has zero. How then can we rectify the graph seen in Fig. 7 so that it complies with the criteria of a TGP tree?

The solution is simple once operators and constant memory registers have been assigned to the nodes as described above. The solution is best thought of as a four step process. Firstly, identify any node J which has h outgoing edges where $h > 1$. Secondly, identify the subgraph which constitutes the nodes that can be reached from the node J (this can be easily achieved using a modified breadth-first search proce-

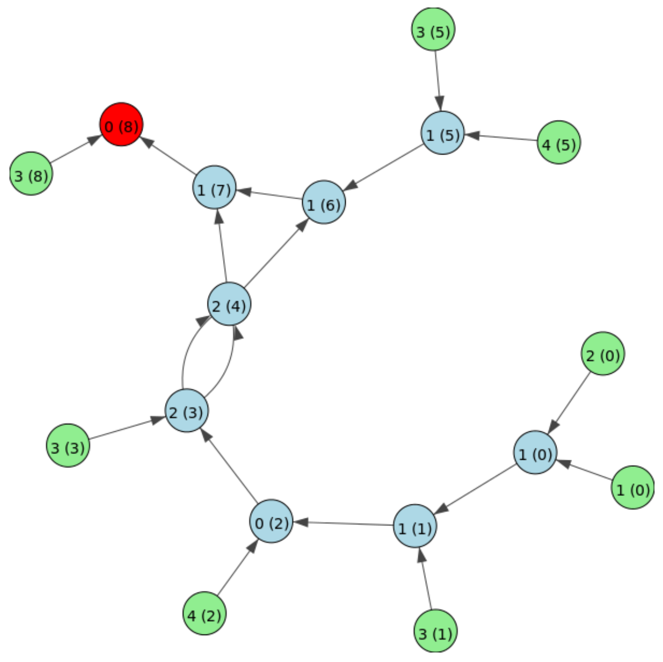

Figure 7. This figure shows the memory register access graph for an LGP program. This figure is included to highlight the presence of multiple outgoing edges in the access graphs of some LGP programs. Note that the node labeled ”2 (3)” and the node labeled ”2 (4)” has two outgoing edges.

dure). Again, we restrict ourselves to only travelling along the reverse direction of the directed edges.

Thirdly, duplicate this subgraph $h-1$ times. As a result we will have $h-1$ nodes which are duplications (with the exception of outgoing edges) of node J. We will label the node J and its duplications as $J_1, J_2, \dots J_h$ where node J_1 is the original J node. The final step is to reassign the source nodes of the h outgoing edge of node J to the nodes $J_1, J_2, \dots J_h$ so that each node has exactly one outgoing edge. This procedure is repeated until each node has either one or no outgoing edges.

This completes our procedure for translating LGP programs into TGP programs.

2.3 Insights into TGP and LGP from translation

Whilst translating LGP programs to TGP programs we came across the problem of some nodes having more than one outgoing edges. In solving this problem we duplicated some nodes of the graph. This problem exemplifies how LGP programs can reuse the resultant values of their instruction executions multiple times. TGP programs do not share this property. Once a calculation is made in the execution of a TGP program its resultant value can only be used again once.

It is possible to imagine at least one advantage and one possible disadvantage of reusing calculations. One advantage is an increase in execution speed. One disadvantage is that less nodes are used in comparison to a network where the TGP scheme is used. This means that the number of different states the graph can occupy is smaller. The number of states of a graph is dependant on the number of nodes and their labelings, the number of edges, and the graph structure among other things. As a result it should be more difficult for the evolutionary processes (e.g. mutation) to make small changes to a program. Or in other words, it is more likely that even a small change made to the program will result in large change in the output value of the program. Programs should in effect become less pliable.

The white nodes in the memory register access graph produced during the LGP to TGP translation process need closer inspection. These white nodes represent instructions whose executions did not affect the final value of the program. We can say this with confidence as our purely access based analysis was independent of which operators or values were used in the arithmetic calculations. Instructions which do not affect the final value of the program are called *introns*. And the white nodes which represent them are called *intron nodes*. The other coloured nodes represent instructions call *exons*. These instructions do affect the final program value. Or, to be more correct, they may affect the final program value given the right operator and memory register values. Exon instructions are represented in the graph with *exon nodes*.

Introns and exons will be discussed further in the next section. However, there is one more comment on introns that should be included in this section: There are many possible ways of defining intron instructions which will be detailed later. For our uses we will define an intron instruction as an instruction which may never modify the final program value regardless of operator type or memory register value used in its calculation. The inability of an intron instruction to affect the final program value is due to access rights.

From our definition of introns it is clear that TGP trees can never feature introns. LGP memory access subgraphs can not exist as part of a single TGP program. Furthermore, it is always possible to create a path from any node in the graph to the root node whilst always moving in the direction of the edges. Every node in the TGP tree will eventually contribute to effecting the final program output if the operators and memory registers allow.

Therefore, it is possible to represent intron instructions under the LGP scheme but not using the TGP scheme. If introns exist in a LGP program, that program's translation into a TGP tree will be lossy. Intron instructions will be lost during the translation.

As a final remark for this section, a further comment on the translation from TGP to LGP: None of the nodes in the memory register access networks of the generated LGP programs have more than one outgoing edge. This of course means that those LGP programs have no access introns. Consequently it may be the case that these programs do not make full use of all the features the LGP scheme provides. That is to say, none of the instructions reuse a calculation more than once. This could be seen as a drawback in the translation scheme proposed. However, there is no requirement for LGP programs to reuse any calculation more than once.

3. Introns in LGP

In the previous section we specified how introns are defined in this project. To reiterate: An instruction within an LGP program is an intron if it's execution could not affect the final value of the program output regardless of the operators or memory register values used.

Other works have specified different definitions for introns. In Brameier (2001) two types of introns are specified: structural introns and semantical instrons. Structural introns are instructions which manipulate variable memory registers that are not used in formulating the final program output. Semantical introns are instructions which manipulate memory registers that may be used in formulating the final program value but they do so in such a way so as to not change the values of those memory registers. This paper also identifies introns in TGP based on operator and memory register values.

The definition of structural introns in Brameier (2001) is equivalent to the definition of introns as used in this project. In Brameier (2001) an algorithm was used for identifying introns which had a worst case linear runtime of order $O(n)$. Where n is the number of instructions.

According to Nordin (1996) and Brameier (2001) introns provide structural protection to high fitness instruction blocks within a population. In effect, intron instructions can act as buffers between groups of high fitness instructions. In doing so intron instructions provide points where crossover cutting can occur without separating the instructions within the high fitness instruction groups. In the presence of introns, crossover cutting points are less likely to be chosen so as to disrupt high fitness instruction groups. This means that high value instruction groups are more likely to be intact during their inheritance into the next generation because of the presence of introns.

Hence, introns serve a protective mechanism against the destructive process of crossover. Note that in some trivial cases the objective program is not complicated enough to warrant the existence of groups of high fitness instructions. In these cases the presence of introns may not be beneficial at all. Introns therefore waist processing time during the execution of a single individual but can quicken the process of finding the best solution over successive generations [2].

However, things are not so clear cut. It was hypothesised in Nordin (1996) that introns may well assist individuals in getting stuck at local minima. An excessive number of introns can provide over protection from crossover. It is reasonable to suggest that the optimum number of introns should depends on the sizes of high fitness instruction groups.

In Nordin (1996) introns were artificially introduced into programs. Successful speedups have also been achieved in genetic algorithms via the insertion of introns [4].

In this project introns were not injected. Instead, introns were created within successive generations by the application of crossover. An initial population of LGP programs were generated which did not have any introns. This was done so as to ensure that any introns present in successive generations must have been generated by the evolutionary mechanisms.

The population of intron-less LGP programs were generated from the translation of a population of TGP trees. This is because TGP trees can not feature any structural introns, a point which has already been discussed. The translation scheme has also been introduced in a previous section. During the evolutionary process, tournament selection was replaced by uniformly random selection. This meant that the fitness of programs did not matter and no particular objective function was specified.

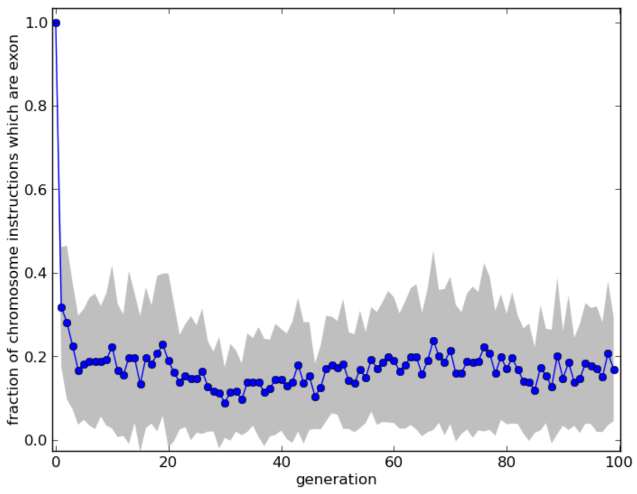

Figure 8. For each generation, this figure depicts the percentage of exon insturctions present in a single LGP program as averaged over a population of 500 programs. The gray shadow gives the standard deviation upper and lower boundaries. The iteration took place over 100 generations. The initial population of LGP programs were entierly composed of 50 exon instructions. TGP trees were translated into their LGP equivalent in forming the first generation. Crossover occoured with a probability of 20%. Three variable registers and two constant registers were used.

With successive generations, the percentage of introns present within the individuals were measured. Figure 8 shows the mean percentage of exon instruction within the programs of each generation. The figure shows that initially all of the instructions within the first generation were exons. Within the next five generations the mean percentage of exons has fallen to around 20%. For the remainder of the 100 generations the mean percentage of exons fluctuates around 20%.

This figure shows that introns are generated by the crossover mechanism. It also shows how the mean percentage of introns remains relatively consistent with successive application of crossover. It is not unreasonable to suggest that the crossover mechanism has a self regulating aspect. It generates a type of instruction which limits its effect. The accumulation of introns are commonly referred to as ”bloat” [2].

The figure shows that soon after the first few generations around 80% of the instructions within an individual are introns. As an interesting comparison within biology, up to 70% of the DNA sequence of eucaryotic cells are not expressed in amino acids [2]. The paper Brameier (2001) provides several other observation of introns whilst detecting six different medical disease using six different LGP instances. The paper states that typically the percentage of introns was around 80%. The paper goes on to state that by ignoring these intron instructions during execution a decrease in classification runtime by a factor of around 5 was achieved.

Figure 8 shows fluctuations both upwards and downwards in the percentage of exons. This suggests that instructions change state from intron to exon and back again as the crossover mechanism is applied over successive generations. In investigating this the intron status of each instruction was recorded as they were copied and moved through the generations.

Recording the status of each instruction is a nontrivial process because instructions can be overwritten by other instructions and therefore cease to exist in the population. Additionally, several copies could be made of an instruction so that the original instruction may now exist as an intron and an exon. To overcome these difficulties only the status histories of the instructions which survived into the final generation were analysed. Also, if an instruction was copied, its copies would be treated as different instructions that would have the same status histories as the original instruction up until the copying event.

Figure 9 shows the results of examining the intron status histories of instructions within a population over successive generations. In generating this figure, the same random fitness and crossover mechanism was used as for Fig. 8. This figure shows the probability distribution of the status group lengths. A status group length is the number of consecutive generations an instruction exists as either an intron or an exon. The figure shows that an instruction had around a 11% probability of remaining either an intron or an exon for a single generation only. An instruction had a probability of around 1% of consecutively remaining in the same state for 25 of the 50 generations.

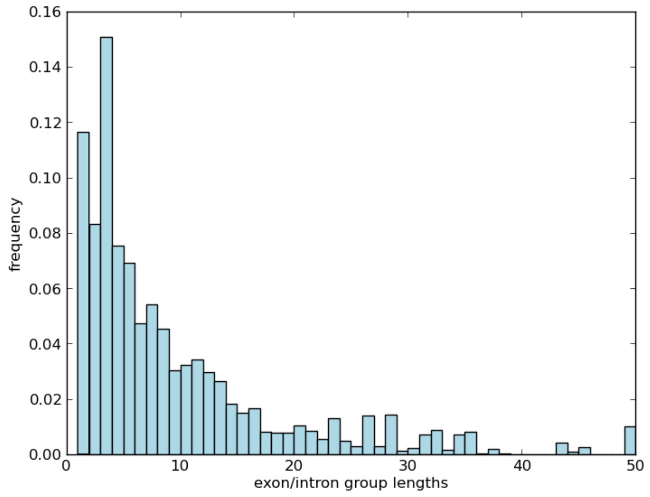

Figure 9. This figure shows the probability distribution for the instruction status group lengths. The iteration took place over 50 generations. The initial population of 500 LGP programs were randomly generated. Crossover occoured with a probability of 20%. Three variable registers and two constant registers were used.

4. Parallel Processing of LGP

There are many forms of parallel processing possible with both the LGP an TGP schemes. The most computationally intensive part of GP is evaluating the performances (fitness) of individuals. The aim of most parallel processing methods is to decreasing the computational burden of this aspect of GP.

One method which is applicable to both schemes, is to parallelise at the population level. In this way, each core (node/CPU) within the parallel processing architecture performs GP on separate populations. This formulation provides more opportunity for avoiding local minima and discovering an acceptable answer quicker.

In Cantú-Paz (1998) this type of parallelization is discussed in the context of genetic algorithms and referred to as *multiple-population coarse grained* parallelisation. The paper discusses a formulation which allows individuals to migrate between populations.

Another method, that is applicable to both LGP and TGP schemes, is to parallelise at the data level. It is usually the case that an individual program is evaluated based on it’s overall performance in dealing with multiple input data sets. In addition, a single input data set can be dealt with by an individual completely independently of any other input data sets. In executing a single individual, every core executes the same individual simultaneously but different input data sets are used by each core. In Cantú-Paz (1998) this type of parallelization is called *single population fine-grained* parallelization.

Yet another method that is again applicable to both LGP and TGP schemes, is called *global single-population master-slave* parallelization. In this formulation each core has a copy of the entire population and one input data set. In this way the fitness of each individual is partially computed by each core.

The method that will be discussed in this project involves parallel computation at the level of a single individual. That is to say, the execution of a single individual for a single input data set is performed over multiple cores. LGP programs are normally difficult to execute in parallel as their memory components exhibit complicated dependencies. In Downey (2011) a scheme called Parallel Linear Genetic Programming

(PLGP) is introduced.

In PLGP a single program is composed of multiple ordered lists of instructions which are called *factors*. The memory registers of each factor during execution are distinct and independent. These factors can in effect be seen as separate and distinct conventional LGP programs. As there are no dependencies between factors, each can be executed on separate cores. The output of the program as a whole is then simply the sum of the outputs from the subprograms.

PLGP showed significant speedup on LGP specifically for larger programs (typically greater than 25 instructions). The speedup during program execution from parallelization is not surprising. However, some speedup in finding a reasonable answer also came as a result of using factors with the crossover mechanism. The use of the factors in effect serve the same use as introns by keeping the destructive crossover mechanism in check. Only some factors of a program were subjugated to crossover and mutation. This limited the effect of those evolutionary mechanisms.

It is easy to see that the execution of a single TGP tree can be parallelized. Separate branches of the tree never have any interdependencies and can therefore be executed by different cores. Two seperable branches join at a node which will be called the *split point*. A tree can have multiple split points.

In this project, the idea of splitting simultaneously executable branches at split points has been applied to LGP programs. As we have seen before, if we were to take a TGP tree and convert it into a LGP program, the structure of it's memory register access graph would look identical to the original TGP tree. Every LGP program created using this method would be composed of branches that lacked any interdependence. If two branches do not have any interdependencies they are said to be separable.

However, as we have seen before, in randomly generated LGP programs it is possible for memory register graph nodes to have multiple outgoing edges. This means that the branches of these memory register graphs may have interdependencies.

We make the assumption that when branches are separable the shortest of both branches (least number of instructions) always has a shorter computational time. In effect, we are assuming that the computational time of executing a set of instructions is linearly proportional to the number of instructions in that set. It is therefore always possible to execute two separable branches within the time it takes to execute the longest branch when threading is used. Note also that branches can stem from other branches. Therefore, the effective computational length of a branch can be reduced by identifying further separable branches (split points) lower down the branch (closer to the leaf nodes).

Figure 10 shows a simple memory register graph where the nodes have been labeled by integers for convenience. The leaf nodes of this memory register graph have been removed; the nodes represent instructions. The red nodes labeled 3 and 4 are split points. Consider the split point at node 4 first. The orange nodes represent the nodes of the longest branch

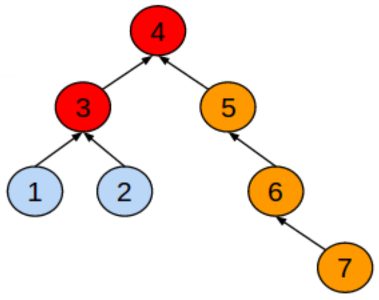

Figure 10. This figure depicts a memory register access graph of a simple LGP program where the node represent instructions. For any given directed edge the source node instructions are the dependancies of the target node instruction. The red nodes represent split point instructions. The orange nodes represent the largest set of instructions which must be executed in sequence on the main thread.

for this split point. The shortest branch for this split point is composed of the nodes 1, 2, and 3.

The effective length of this shortest branch is actually two nodes (nodes 1 and 3 say) and not three (nodes 1, 2, and 3). This is because a second split point exists on the shortest branch of node 4's split point. The second split point is at node 3. At the second split point, one branch (they are both the same length so it does not matter which one) can be executed at the same time as the other branch. Or more explicitly, node 1 and node 2 can both be executed simultaneously in different threads.

Therefore, in executing the program depicted in Fig. 10 the nodes 5, 6, and 7 would all be executed in the main thread. Nodes 3 and 2 would be executed in another thread. And node 1 would be executed in yet another thread. The computational runtime would be given by the longest runtime of the three threads plus the runtime of node 4.

If a node is not part of the longest runtime thread then that node represents an *off main thread* instruction. Off main thread instructions do not contribute to the computational runtime of the largest thread. The off main thread instructions can be executed during the execution time of the largest thread. In Fig. 10 we can see that there are three off main thread instructions, nodes 1, 2, and 3. In the best case scenario, at most 50% of the instructions could be off main thread instructions (bar the last executed split point instruction).

Figure 11 shows another example memory register access graph but this time with leaf nodes. The nodes in this graph therefore represent memory access events. The dark blue, light blue, and red nodes can be treated as representing instructions. Effectively, the red node represents the final exon instruction and the dark blue nodes represent split point instructions. This figure shows two off main thread instructions.

Figure 12 is a flowchart of the algorithm used for identifying all the possible split point instructions within an LGP program. Before running this algorithm all intron instructions must be, at least temporarily, removed. In identifying the split point instructions the algorithm tests each memory register access graph instruction node (nodes which are not green leaf nodes) for two necessary split point conditions. Firstly, a split point instruction node must be the parent of two other instruction nodes. Secondly, there may not be an edge path from one child instruction node to the other without including their parent instruction node (the potential split point) in the path.

The computation time of the off main thread instructions can be ignored when calculating the effective execution time of the LGP program. Figure 13 shows the probability distribution for the number of split points found in LGP programs of various exon instruction lengths. Figure 14 shows the probability distribution of the number of off main thread instructions found in exon LGP programs of different lengths.

Both figures shows that there is approximately a 10% to 13% probability that a 25 exon instruction length program would not benefit from this proposed parallelization method. There is therefore a 87% to 90% probability that at least some speedup can occur. In some cases up to 11 of the 25 exon instructions (44%) could be executed off the main thread.

Applied to ordinary LGP this parallelization method has several downsides compared to PLGP. The split points have to be refound within every individual at every generation. Therefore the proposed method is only beneficial if it is assumed that the sum computational time of executing the instructions over all input data sets is significantly greater than the computational time of finding the split points. This is not an unreasonable assumption.

However, this proposed parallelization method can be used with the PLGP scheme. We have previously mentioned the concept of factors within PLGP programs. During execution, factors are effectively LGP programs with distinct and independent memory registers. These factors could benefit from this proposed parallelization method.

In future work it would be necessary to investigate the possibility of tracking split points. Faster algorithms for testing the second split point criteria could be developed by specifically investigating branch nodes which have multiple outgoing edges. Testing the second criteria as it currently stands is somewhat computationally intensive as it effectively involves performing a modified breadth-first search.

5. Conclusion

In this project the basics of LGP and TGP were discussed. Then two schemes were introduced for translating from LGP to TGP and vice versa. As a result of these translation schemes intron instructions could be identified in LGP programs. By translating TGP programs, LGP programs could be formed which were composed entirely of exon instructions.

It was shown that intron instructions were formed from applying the crossover mechanism to a population of purely exon LGP programs over successive generations. After around five generations approximately 80% of the instructions within the population were introns. This percentage of introns has been observed in applying LGP to classifying medical data and also within the DNA sequences of some biological organisms.

The different methods of GP parallelization were briefly discussed with special attention given to the PLGP scheme. A method of parallelising the execution of a single LGP program was outlined. Estimations were made of the potential speedup that such a scheme could provide. It was estimated that the probability of decreasing the computation time of executing a 25 exon instruction program was between 87% and 90%.

Acknowledgments

I would like to thank my supervisor Professor Mattias Wahde (Chalmers, Sweden) for helpfull discussions and advice throughout this project.

References

[1] Brameier, M. and Banzhaf, W., A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining, IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 5, NO. 1, 2001.

[2] Nordin, P., Francone, F., Banzhaf, W., Explicitly defined introns and destructive crossover in genetic programming, Advances in genetic programming, Pages 111 - 134, MIT Press Cambridge, 1996.

[3] Knuth, D. E., The Art Of Computer Programming Vol 1. 3rd ed., Boston: Addison-Wesley, ISBN 0-201-89683-4, 1997.

[4] Levenick, J. R., Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology, Proceedings of the Fourth International Conference on Genetic Algorithms, 1991.

[5] Downey, C., Explorations in Parallel Linear Genetic Programming, Victoria University of Wellington, 2011.

[6] Cantú-Paz, E., A survey of parallel genetic algorithms, Calculateurs paralleles, reseaux et systems repartis, Vol. 10, 1998.

[7] Wahde, M. Biologically inspired optimization methods, WIT Press, 2008.

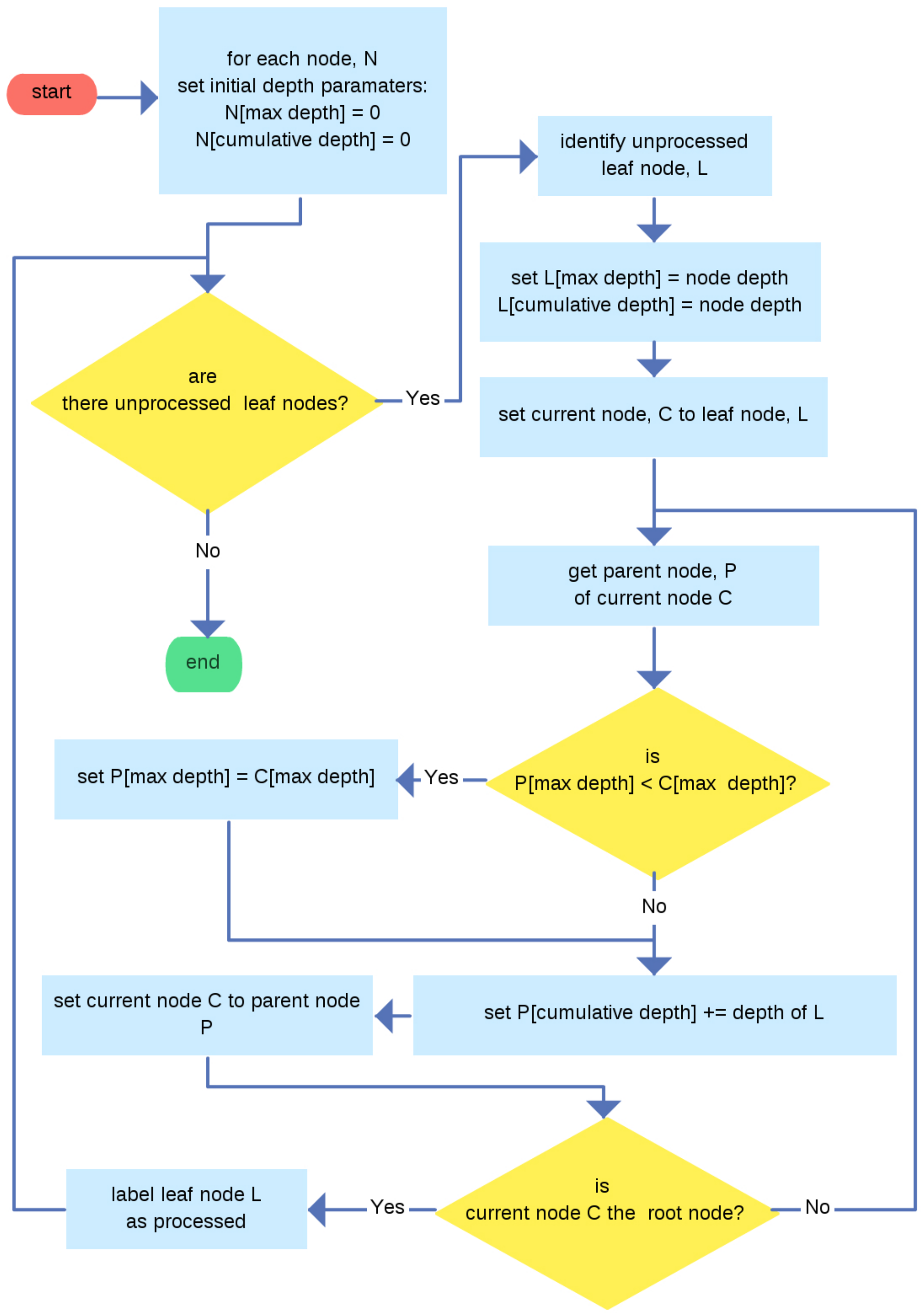
start
for each node, N
set initial depth paramaters:
N[max depth] = 0
N[cumulative depth] = 0
are
there unprocessed leaf nodes?
Yes
No
end
identify unprocessed
leaf node, L
set L[max depth] = node depth
L[cumulative depth] = node depth
set current node, C to leaf node, L
get parent node, P
of current node C
is
P[max depth] < C[max depth]?
Yes
No
set P[max depth] = C[max depth]
set P[cumulative depth] += depth of L
set current node C to parent node
P
is
current node C the root node?
Yes
No
label leaf node L
as processed

Figure 3. This figure shows a flowchart of the first part of the algorithm that is used in translating TGP programs to LGP programs. The algorithm associates each node in a TGP tree with 'max depth' and 'cumulative depth' statistics. These statistics are used by the second part of the algorithm in forming the node processing order.

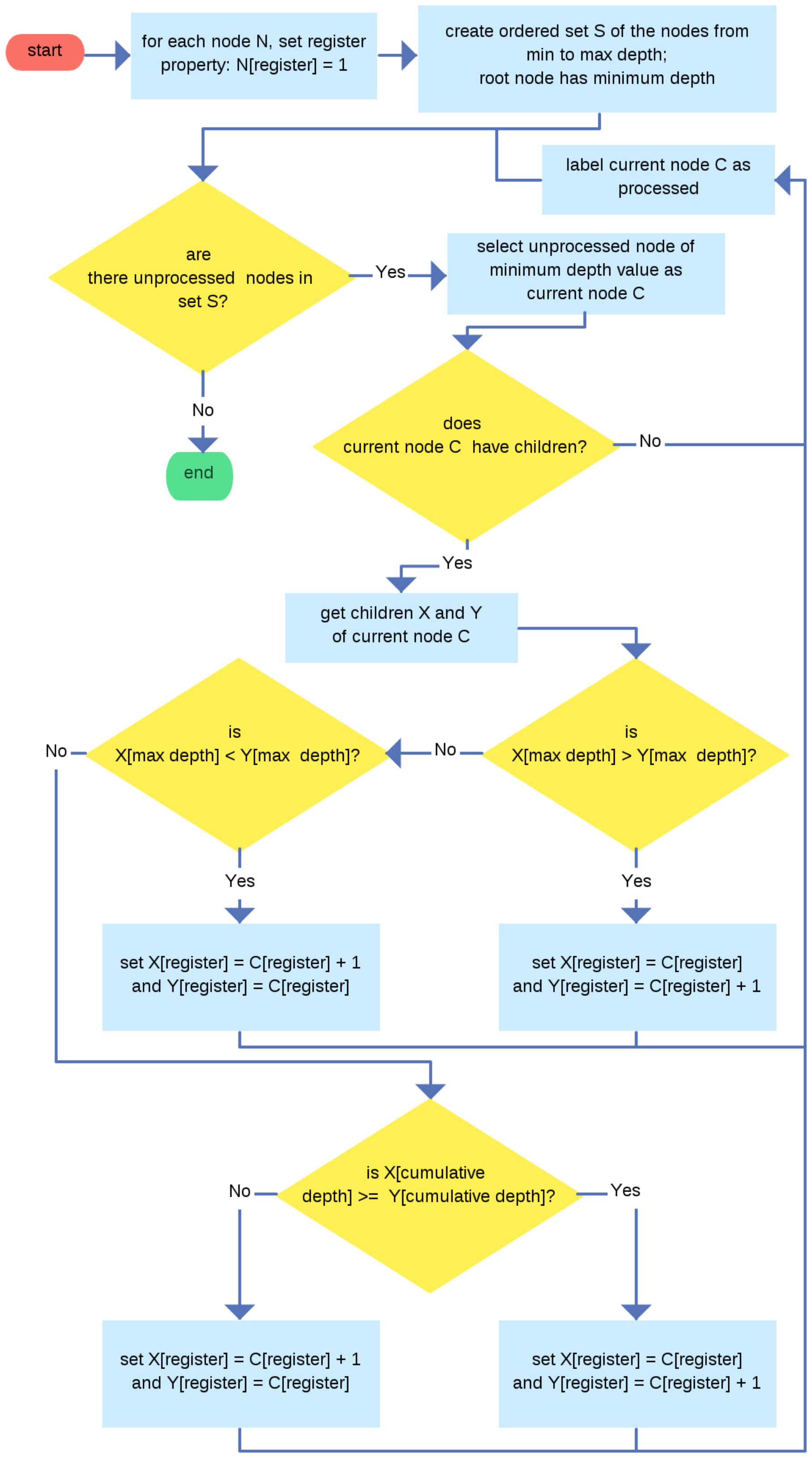
start
for each node N, set register property: N[register] = 1
create ordered set S of the nodes from min to max depth; root node has minimum depth
label current node C as processed
are there unprocessed nodes in set S?
Yes
No
end
select unprocessed node of minimum depth value as current node C
does current node C have children?
No
Yes
get children X and Y of current node C
is X[max depth] > Y[max depth]?
No
is X[max depth] < Y[max depth]?
No
Yes
Yes
set X[register] = C[register] + 1 and Y[register] = C[register]
set X[register] = C[register] and Y[register] = C[register] + 1
is X[cumulative depth] >= Y[cumulative depth]?
No
Yes
set X[register] = C[register] + 1 and Y[register] = C[register]
set X[register] = C[register] and Y[register] = C[register] + 1

Figure 4. This figure shows a flowchart of the second part of the algorithm used in translating TGP programs to LGP programs. The algorithm assignes memory registers to each node in the tree. The minimum possible number of different memory registers are assigned to the tree.

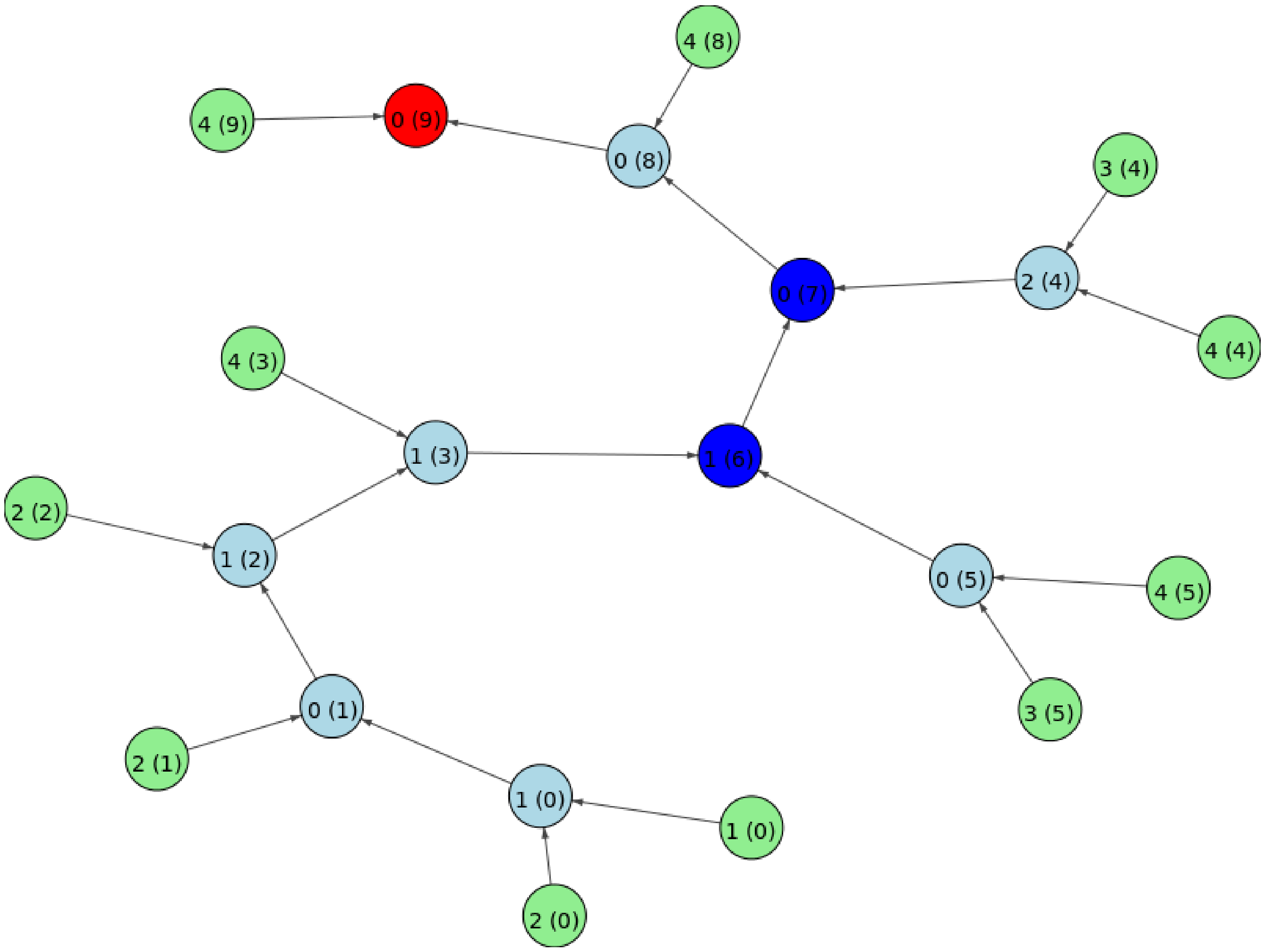

Figure 11. This figure shows a memory register access network for a LGP program. Each node represents a memory register access event. The instructions represented by dark blue nodes have been identified as split points.

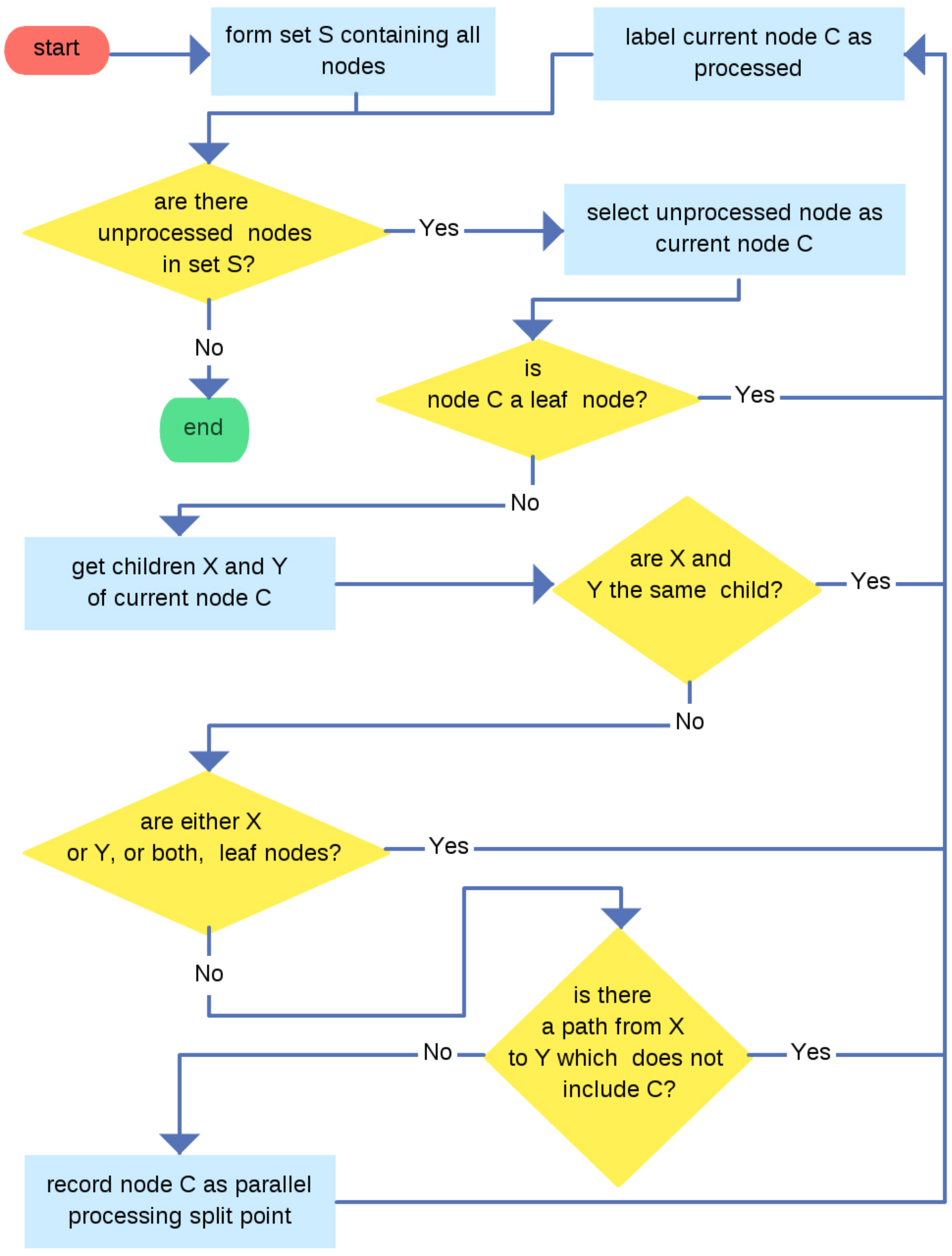
start
form set S containing all nodes
label current node C as processed
are there unprocessed nodes in set S?
Yes
select unprocessed node as current node C
No
end
is node C a leaf node?
Yes
No
get children X and Y of current node C
are X and Y the same child?
Yes
No
are either X or Y, or both, leaf nodes?
Yes
No
is there a path from X to Y which does not include C?
No
Yes
record node C as parallel processing split point

Figure 12. This figure shows a flowchart of the algorithm used in identifying split point instructions for the parallel execution of individual LGP programs.

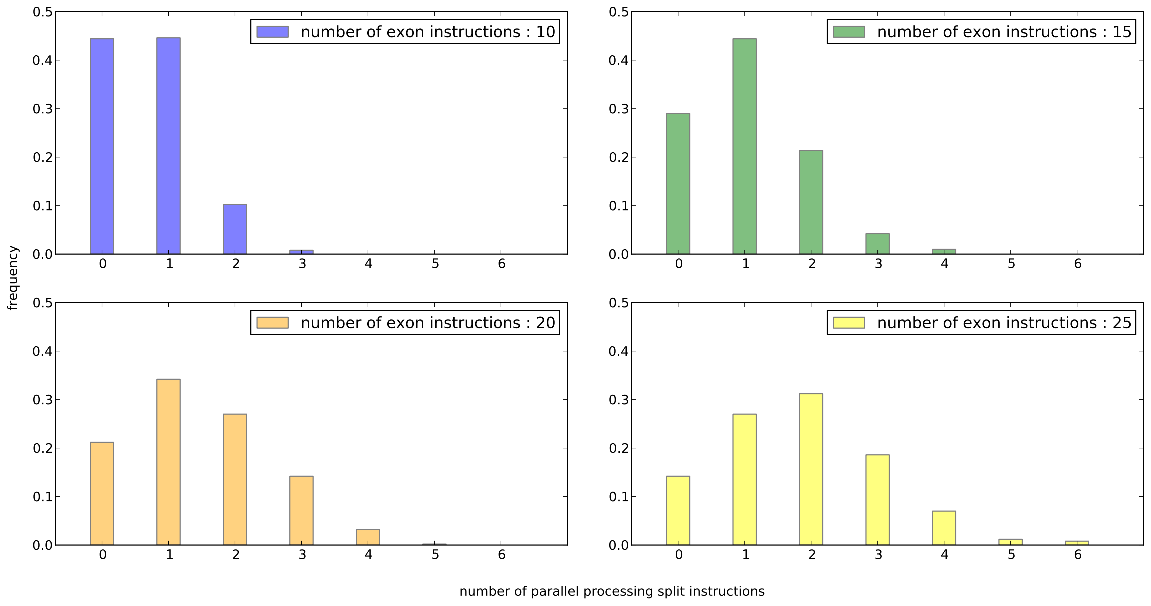

Figure 13. This figure shows four probability distributions of the number of split points identified in LGP programs which are composed of 10,15,20, and 25 exon instructions. 500 LGP programs were used in creating each probability distribution.

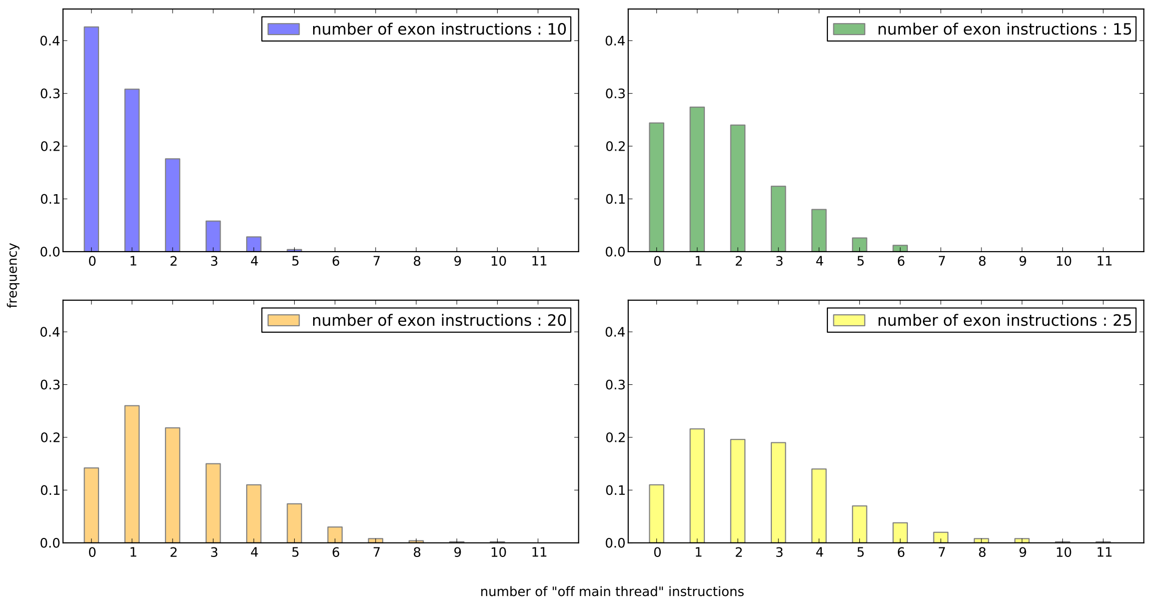

Figure 14. This figure shows four probability distributions of the number of off main thread instructions in LGP programs which are composed of 10,15,20, and 25 exon instructions. 500 LGP programs were used in creating each probability distribution.