VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

VUWLGP — An ANSI C++ Linear Genetic Programming Package

Christopher Fogelberg and Mengjie Zhang

Technical Report CS-TR-05/8
December 2005

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VUWLGP — An ANSI C++ Linear Genetic Programming Package

Christopher Fogelberg and Mengjie Zhang

Technical Report CS-TR-05/8
December 2005

Abstract

Linear Genetic Programming (LGP) is a recently researched form of genetic programming, the automatic evolution of computer programs which can solve problems. Traditionally, genetic programs have been represented as function trees. However, LGP programs are linear sequences of instructions (e.g. register machine instructions) and are not best represented as a tree of functions and terminals. Few publicly available packages designed to support research into LGP exist and those that do are often incomplete. VUWLGP has been written in C++ and is available for use under the GPL. It is designed to be easily customised and tweaked so that slightly different variants of different problems can be researched easily.

Contents

1 Introduction

Genetic programming has been formulated by Koza [5], Banzhaf [1] and others. By defining a numerical measure of how well a program can solve a problem the mechanisms of biological evolution can be repeatedly applied to a population of programs which was initially randomly generated. Through the power of natural selection this can lead to extremely good solutions to a problem being found relatively quickly. Genetic programming is importantly different from other approaches to evolutionary computation (such as genetic algorithms) in two important ways:

- The solutions evolved are directly executable or interpretable as executable programs. There is no translation step used.
- The size and structure of a genetic program is constrained only by the configuration — it does not need to be specified exactly by the user of genetic programming before any computational learning is performed.

VUWLGP is written in ANSI C++ and is designed to be highly customisable and easily modified. Hence research into a variety of forms of LGP can be carried out and each of these forms can be easily applied to a number of different problems. A general interface is defined by the classes contained in the `baseSrc` directory. For some of these classes sensible and general concrete implementations could not be provided. Therefore a number of specific implementations have been provided in the `implementations` directory. Reading and understanding these classes and the `baseSrc` classes should be the starting point for users wishing to create a customised VUWLGP *operating environment*. These classes are all well commented.

VUWLGP has been developed at the School of Statistics, Mathematics and Computer Science, at Victoria, University of Wellington in Wellington, New Zealand. It is a public tool released under the GPL and is available from `http://www.mcs.vuw.ac.nz/~mengjie/packages/vuwlgp` and `http://www.syntilect.com/vuwlgp`.

2 Using VUWLGP

When using VUWLGP two elements of it will need to be set up so that the runs against a problem can be attempted.

Firstly, the operating environment must be designed and written. The operating environment constitutes the algorithmic aspect of the run and is defined by the classes used in the runs. This process is described in more detail in section 2.3.

Secondly, the *configuration* must be specified. This typically involves assigning values to members of an instance of the class `Config`. This instance is then passed to the other classes as they are constructed. However, in the implementation of the `IPopulation` class provided (`GenerationalTournamentPopulation`) the weighted likelihood of each evolutionary operation except elitism must be hard-coded in the method `GenerationalTournamentPopulation::IteratePopulation`. The proportion of the population selected via elitism is defined by a public static member of `GenerationalTournamentPopulation`. In future versions of VUWLGP this may be integrated into the `Config` class. How to use this class is described in section 2.2.

The code itself is well commented and the readers are referred to this for documentation on the functions themselves.

Users of VUWLGP are assumed to have an at least passing familiarity with genetic programming. If unfamiliar with genetic programming then [5] or [1] are suggested. Those unfamiliar with LGP are referred to section 3.

2.1 VUWLGP Design

In this subsection the design and class structure of VUWLGP is briefly discussed and an overview of how the evolution "flows" from class to class is given.

The class structure of VUWLGP is given in figure 1. Implementations of the abstract classes (those classes whose name is prefixed with an I) are not shown in this figure. Examples can be found in the `implementations` directory.

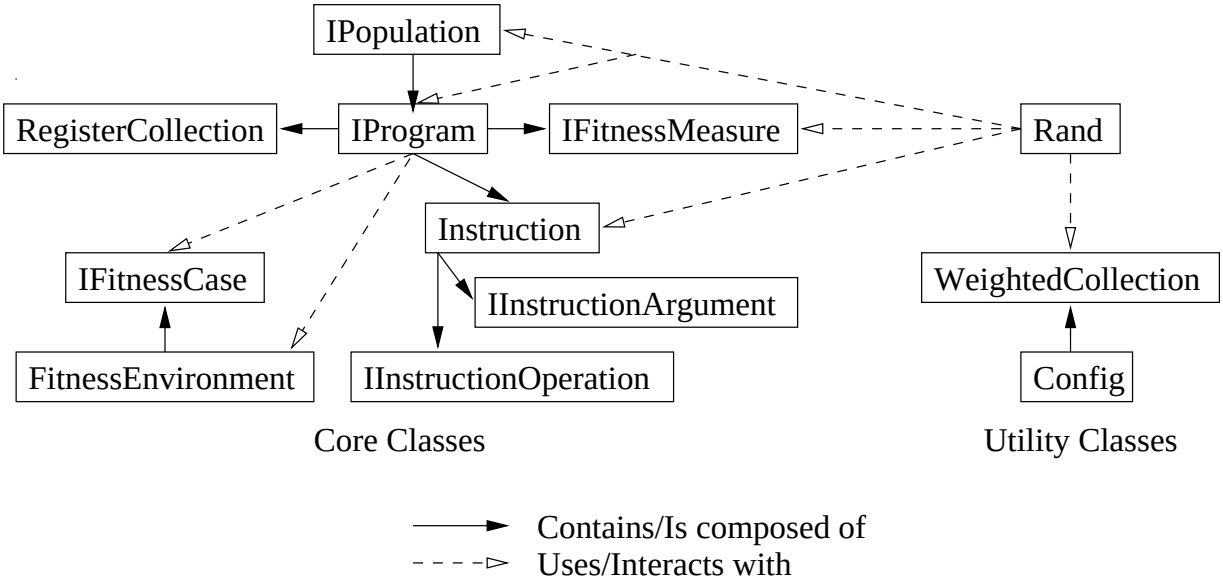

Figure 1: The VUWLGP class structure, showing which classes are composed of which others, and which classes use each other. None of the classes shown inherit from one another.

The reader is referred to the code itself and its comments for discussion regarding each function and its role. Many of the classes have a number of template parameters, where possible constraints are enforced so that bugs cannot occur. The role of each of these parameters is described in table 1. Users of VUWLGP should ensure that these constraints are met in all cases, or strange errors may occur during compilation or at run-time.

Table 1: Instruction/argument position to link sign mapping.

Parameter Name	Description
`T`	The type of the registers — i.e. what type the features should be, what type the instruction operations should return, etc.
`IProgramSubclass`	The type of the sub-class of `IProgram` being used in VUWLGP. The `T` of this class should match the `T` which is also necessary as a parameter to, for example, the sub-class of `IPopulation` being used.
`IProgSub`	A shortened name for `IProgramSubclass`. It should be the same class as `IProgramSubclass`.

With this discussion of the class design in mind the reasoning behind the design of VUWLGP will now be briefly explained. Following that the flow of control from class to class will be explained. The design and flow are at first counter intuitive but result in a simpler final code structure.

2.1.1 Design Rationale

The core component of a genetic programming representation must be the program. A number of elements are associated with a program. These include the pieces out of which it is made (in LGP, the instructions) and some measurement of its fitness. In many implementations of genetic programming the measure of fitness is hard-coded into the class as a double value, however this representation is not suitable for all problems. In some problems, such as multi-class classification, more fine-grained information on a program's fitness may be needed. Thus, in this design the fitness measure of a program has been seperated out from the program class. Note that a program also caches the final register values it wrote after it was last executed. This is an implementation detail which makes updating a program's fitness based on its performance on some fitness case much easier.

Similarly, the instructions out of which a program is composed can be broken down into three key components:

1. The function or operation which this instruction applies.
2. The arguments passed to the function or operation.
3. Optionally, the register the instruction writes its output to (not all instructions generate output — conditionals do not, for example).

In VUWLGP these three elements can be individually specified, changed and added to without altering the core `Instruction` class. This means that new operations can be added and trialled without modifying the core code, so long as they can take 0–2 parameters of type `T`, the type of the arguments. Similarly, new types of arguments can be added provided they are of type `T`. Note that the destination register must be a read-write register — it makes no sense for it to be a read-only register.

The `FitnessEnvironment` class stores register values and the fitness cases a population of programs is being evaluated against. It provides methods for iterating through fitness cases and for managing them. If a test set is needed in addition to a training set, it should be loaded into a seperate FitnessEnvironment.

Finally, the sub-class of `IPopulation` being used determines how a population is evolved when the fitness of all of its members is accurate. It is important to note that VUWLGP is structured so that members of the population changed by an evolutionary operation are marked. Then, only these programs need to be executed to ensure that all members of the population have an accurate fitness. This means that the full efficiencies of steady state genetic programming can be realised. In addition, the sub-class of `IPopulation` defines how a member of the population is selected for an evolutionary operation based on the (presumed accurate) fitness of the programs in the population.

This selection algorithm is defined by a sub-class of `IPopulation`. The method this sub-class must implement is called `SelectProgByFitness`. It should select some programs, based on the fitness of the programs in the population, and return a pointer to it.

The second method which the sub-class of `IPopulation` must implement is called `IteratePopulation`. When called this method replaces some (perhaps all) members of the population with new programs generated by evolutionary operations which operate on selected programs. The changed programs should be marked as needing their fitness updated.

These two methods are not implemented in `IPopulation` itself because they tend to vary quite substantially from problem to problem. On the other hand, other elements of the evolutionary are specified in `IPopulation` as they will not need to be changed except in very rare circumstances.

For more code-oriented documentation see the HTML documentation (`doc/index.html` or `http://www.syntilect.com/vuwlgp/doc/index.html`).

2.1.2 The Flow of Control

VUWLGP's flow of control can best be understood by reading the code of one of the simple example problems. In brief though, a user of VUWLGP calls `Evolve` on an `IPopulation`, passing it a `FitnessEnvironment` which the population of programs will be evolved against.

In `IPopulation::Evolve` the population is iterated up to `maxGenerations` times and the flagged programs have their fitness evaluated after each iteration of the population. `EvaluateFlaggedPrograms` in turn calls `UpdateFitness` on each flagged program, passing it the `FitnessEnvironment` passed to `Evolve`. In `IProgram::UpdateFitness` the set of fitness cases in the fitness environment is iterated over and the program's fitness is updated after each case.

2.2 The `Config` Class

The `Config` class defines the configuration of a VUWLGP run, e.g. the data set specific elements of a problem. Users of VUWLGP can inherit from `Config` and add more members, however almost all VUWLGP classes require an instance of `Config` or a subclass of it to be passed into their constructors. The same instance of `Config` should probably be passed to every constructor in a run which needs one, else strange bugs might occur.

2.2.1 Constructors and Other Methods

Config() This method sets up a `Config` object for its member variables (specified below) to be initialised by direct assignments in code in which the instance of `Config` is in scope.

Config(int argc, char argv, bool printAsParses)** If all or a known sub-set of the configuration variables are going to be provided on the command line then the command line arguments can just be passed to this constructor (although the `Instruction` configuration variables will still have to be manually initialised). This constructor uses the `Init` method, below. Read there for more information.

void Init(int argc, char argv, bool printAsParses)** This method takes what are assumed to be the command line arguments passed to the application — therefore, the 0'th is not considered as it is expected to be the name of the executable. All configuration variables *excluding* the `Instruction` configuration variables can be specified on the command line and then parsed by this method. The format the variables are expected to be in is `nameOfVariableAsGivenBelow:value`. If `printAsParses` is true then the configuration variables which are parsed by it will be printed out.

Copy Constructor, Destructor The copy constructor is made private, because instances of a Config class do not need to be copied and because doing so introduces complications. The destructor is defined as would be expected.

2.2.2 Program Configuration Variables

These variables are used to configure programs-level elements of the configuration. Some of these also have a bearing on the instruction-level elements of the configuration as well, as programs and instructions are tightly interrelated.

unsigned int numRegisters This variable defines the number of read-write registers available.

unsigned int numFeatures This variable defines the number of read-only feature registers which describe the problem. VUWLGP is currently structured to be most easily used for feature-based problems such as multi-class classification, which is what it was originally designed for. If using VUWLGP for a non-feature based problem then this configuration variable can ignored.

unsigned int initialMinLength This variable defines the minimum length programs can be initially generated at. Programs are created uniformly across the range specified by this configuration variable and by `initialMaxLength`.

unsigned int initialMaxLength This variable defines the maximum length programs can be initially generated at. Programs are created uniformly across the range specified by this configuration variable and by `initialMinLength`.

unsigned int maxLength This variable defines the maximum length a program can have during the evolutionary process. No minimum is specified in the current implementation of VUWLGP although if one was needed it could be trivially added by modifying the `Config` class and altering the evolutionary operations in `IPopulation` to ensure that it was respected.

2.2.3 Instruction Configuration Variables

These variables are used to configure the instructions which get created, including the operations (functions) available and the kinds of arguments (registers) which can be created and applied to them. An instruction (at least in the current implementation of VUWLGP, future implementations and extensions are not constrained in this way) consists of one destination register, which must always be a read-write register, two arguments and an operation. Note that some or all of the arguments may not be used by the operation. The kinds of arguments and operations which can be generated are set by these configuration variables.

WeightedCollection<ArgumentGeneratorFunc>* argumentGenerators This definition declares a `WeightedCollection` which stores pointers to functions which return arguments to instructions — for example, one such generator function may return a feature register, another might return an ephemeral constant. By changing the function pointers stored in this collection and the weights assigned to them the instructions generated randomly will be changed.

WeightedCollection<OperationGeneratorFunc>* instructionOperations As with the argumentGenerator, this configuration variable is a `WeightedCollection` of functions which generate the operations of an instruction, such as a plus-operation, a sine-operation or an if-less-than-operation.

2.2.4 Population Configuration Variables

unsigned int populationSize This variable specifies the number of programs which make up the population. If a deme-based `IPopulation` sub-class was being used then this variable would need to be extended, changed or ignored.

2.2.5 Evolutionary Configuration Variables

double epsilon For a solution to be considered good enough (and thus for evolution to be terminated early) a program must have fitness less than or equal to this. Note that VUWLGP currently only supports lower-is-fitter evolutionary structures. This constraint is not considered a significant problem[1] but may be addressed in future versions of VUWLGP.

unsigned int maxGenerations The maximum number of generations, after which evolution is forcibly terminated. The exact semantic meaning of this configuration variable depends on how the sub-class of `IPopulation` which is being used iterates the population, as the "generation count" is incremented after each call to `EvaluateFlaggedPrograms` and `IteratePopulation` in `IPopulation::Evolve`. In a generational model it would presumably be the number of generations. Alternately, if steady-state evolution was being used it would count the number of programs replaced.

2.2.6 Logging and Rand Configuration Variables

unsigned int popLogInterval The initial, final and every `popLogInterval`'th population of programs will be logged to file.

std::string popLogFilePath Every `popLogInterval` iterations (as specified and defined by the implementation of `IPopulation` which is being used) the entire population will be logged to a file. The filepath as specified does not need to include any extension — the generation and then `.txt` will be automatically appended.

std::string statsLogFilePath Generation-by-generation statistics about the evolutionary run (the best program, it's fitness, the number of genetically unique programs in the population, the average fitness, size etc.) are appended to this file. The filepath as specified does not need to include any extension — `.txt` will be automatically appended.

std::string runLogFilePath The overall statistics of the run (the number of generations and length of time used, the training fitness of the fittest program and its test fitness) are appended in csv format and in that order to the file specified by this configuration variable.

unsigned int randSeed This variable specifies the seed to be used for the random number generator. If it is not set (see `seedSpecified`) then a time-dependent seed is used.

bool seedSpecified This variable is special and does not need to be specified on the command line — only the randSeed needs to be specified there. If that is encountered `Init` will set `seedSpecified` to true. However, if the seed is specified manually or through some other method then this will need to be manually set to true. This is because there is no value of an unsigned int which can be used to signify "not set to a specific seed". The `Rand::Init()` method checks whether or not the seed has been specified by checking the value of this variable and if it has not been uses a time-dependent seed.

[1] If there is no other way to achieve a lower-is-better fitness then they all fitnesses can just be multiplied by -1 for calculation purposes. This effectively eliminates the "lower-is-better" constraint.

2.3 Customising VUWLGP

A user who wishes to customise VUWLGP may wish to customise a number of facets of it. In this subsection of the technical report a description of how this can be done for a number of specific modifications will be given.

2.3.1 Implementing a New Kind of Instruction Operation (Function)

To implement a new kind of instruction operation, two key tasks must be completed:

1. A sub-class of `IInstructionOperation` must be created and included in the source of VUWLGP being used.

2. The generator function for the new operation must be added to `Config::instructionOperations` in the instance of `Config` used, described below.

The second of these tasks is relatively easily achieved. `instructionOperations` is a `WeightedCollection` — simply pass a pointer to the generator function to `WeightedCollection::AddElement(T elem, double weight = 1)`. Note that the weight must be >= 0 (although a zero weight element of the collection can never get randomly selected).

To complete the first task, `IInstructionOperation` must be sub-classed. If the operation is a conditional instruction operation though then `IInstructionOperationConditional` must be sub-classed instead. `IInstructionOperationConditional` is itself a sub-class of `IInstructionOperation`.

The sub-class must implement the following methods:

Execute This method takes three registers as arguments, and should write the result of the operation (if any) to the `dest` register. This method also takes a `FitnessEnvironment`, as this is necessary to provide feature or case-specific values to the arguments (it is taken as a parameter in their `Value` method). The method should return true if the next non-conditional instruction should be executed, false otherwise. This allows conditional instructions to be integrated easily.

Clone Clone is basically a virtual copy-constructor which allows copies of an instruction operation to be made without knowing the exact sub-class of `IInstructionOperation` an instance is. It can probably just use the sub-classes' copy constructor, as in the body of the `Clone` method the type is known.

Generate This static method works similarly to `Clone` but instead returns a new instance of this class.

ToString As every operation will have a different string representation (e.g. "+", "<" etc.) each sub-class of `IInstructionOperation` must implement this method. Note that in the current implementation of VUWLGP the operation will always be placed between the arguments (which makes, e.g., sine confusing when converted to string format) but if the instruction operation is a conditional then `if(` and `)` will be wrapped around the string representation of the instruction.

2.3.2 Implementing a New Kind of Instruction Argument

Implementing a new kind of instruction argument is very similar to implementing a new kind of instruction operation. Three key tasks need to be completed to implement a new kind of instruction argument:

1. A sub-class of `IInstructionArgument` needs to be created.

2. The `ArgumentType` enum in `baseSrc/IInstructionArgument.h` should be updated to include the value returned by the just-created sub-classes' `ArgumentType` method. `IProgram::MarkIntrons` may also need to be modified or extended (although unless the new type of argument is very similar to a standard read/write register it probably will not need to be).

3. A pointer to the generator method should be added to the `argumentGenerators` member of the instance of `Config` being used.

Extending the `ArgumentType` enum is fairly simple. It is important each argument or type of argument have a unique `ArgumentType` though.

Similarly, there should be a method which returns a pointer to a new instance of the class for every `IInstructionArgument` sub-class. Ideally this requirement would be enforced by having a pure-virtual static method in `IInstructionArgument` which each sub-class must implement but unfortunately this is not possible in C++. Regardless, for the new kind of instruction argument to appear in instructions in a program a pointer to the generation function must be added to the weighted collection `argumentGenerators` in the instance of `Config` being used.

Finally, the sub-class of `IInstructionArgument` should implement 6 methods:

Value This method returns the value of the argument for this case, position in the code and type or index. This method takes a `FitnessEnvironment` so that — if needed — the current case or general characteristics of the `FitnessEnvironment` can be referenced when determining the value of the argument.

Clone This method acts as a virtual copy constructor and creates a copy of this `IInstructionArgument`

ToString This method returns a string representation to the argument, e.g. register 3 might be represented as an index into an array — "`r[2]`".

ArgumentType and ArgumentIndex These methods are used in `MarkIntron`. In the current implementation of the method the index of read-write registers needs to be tracked, and these methods are used to check wheck a particular argument to an instruction is for a read-write register and if it is, which specific register it is. If another form of read-write register is created (i.e. one which might not be just an entry point to the directed acyclic graph an LGP program forms) then `IProgram::MarkIntron` will need to be overridden and updated.

Generate The creation of this method cannot be enforced by the compiler, but some method which returns a pointer to a new instance of the class should be created somewhere. Making it a public static member of the new class is suggested.

2.3.3 Implementing a New Kind of Instruction

More substantial changes are necessary to implement a new kind of instruction. A new sub-class of `Instruction` which represents the new sort of instruction needs to be created, Depending on the extent of the changes this sub-class may need to override any of the following methods:

Execute If the underlying core of the class is changing — i.e. if the instruction no longer always has an operator and two arguments — this method will need to be updated. If

the core is augmented — for example if a weight is added to an instruction so that hill climbing can be used alongside the evolutionary search — then this method may need to be overridden as well.

Clone This method will need to be overridden so that the copy constructor of the right kind of `Instruction` is called.

Mutate If the underlying core of the class is changing — i.e. if the instruction no longer always has an operator and two arguments — this method will need to be updated. If the core is augmented — for example if a weight is added to an instruction so that hill climbing can be used alongside the evolutionary search — then this method may need to be overridden as well.

ToString If the underlying core of the class is changing — i.e. if the instruction no longer always has an operator and two arguments — this method will need to be updated. If the core is augmented — for example if a weight is added to an instruction so that hill climbing can be used alongside the evolutionary search — then this method may need to be overridden as well.

DestinationIndex, DestinationType etc. If the underlying core of the class is changing — i.e. if the instruction no longer always has an operator and two arguments — these methods will need to be updated. Similarly `IProgram::MarkIntrons` may also need to be updated.

See the code for more details on these methods. As well as sub-classing `Instruction`, a new sub-class of `IProgram` must be created. This is discussed in the next sub-subsection.

2.3.4 Implementing a New Kind of Program

Note that in the current design of VUWLGP a new kind of program will need to be implemented whenever a new sub-class of `Instruction` is being used. Similarly, if a new kind of fitness measure is being used a new sub-class of `IProgram` will be necessary in the current design of VUWLGP. It is likely both of these will become template parameters in future versions of VUWLGP.

To implement a new kind of program a class which is a sub-class of `IProgram` must be created with 4 methods. This class should probably be a template on `T`, as `IProgram` is, unless this type of program is only compatible with some specific kind of register.

The four methods which must be created in this sub-class are as follows:

Constructor Every sub-class of `IProgram` is expected by `IPopulation` to have a constructor which takes two arguments. The first argument should be an unsigned int and should specify the size of the program which should be constructed. The second argument should be the `Config` object being used.

Copy Constructor Your sub-class of `IProgram` should implement a copy constructor. This copy constructor does not need to be very complex as most of the work is done in the copy constructor of `IProgram`, however the fitness measure (and not just the pointer to it) does need to be deep-copied.

static Instruction<T>* InstructionFactory(Config<T>* c) All sub-classes of `IProgram` should implement a method with this name and signature as it is used in `IPopulation` (on the template parameter it takes and which is assumed to be a sub-class of `IProgram`) and your code will not compile if it is not present. This method is used to create pointers to new instances of whatever type of `Instruction` this type of program uses.

MakeAbstract This method is never called, however it is set as pure virtual in `IProgram` to force that class to be abstract. Sub-classes of `IProgram` must implement it so that they are not. The suggested implementation is as an inline method which does nothing (`...{ }`)

2.3.5 Implementing Steady State LGP with Proportional Selection

One kind of new sub-class of `IPopulation` which might be usefully implemented is a population class which performs proportional-selection and steady-state evolution.

When implementing a new sub-class of of `IPopulation` two methods must be implemented — one for each of these elements of functionality.

IteratePopulation This method is called by `IPopulation::Evolve` whenever all programs in the population have updated and correct fitnesses and the evolution can be performed (it is assumed the evolutionary operations should happen as soon as possible). The method is void and should work by altering programs stored in the current population or by replacing the member which stores the population (`IPopulation::programs`) with the new population.

SelectProgByFitness This method returns a const pointer to one program in this population. It is assumed that all evolutionary operations return only one program, as this makes the most sense. Having crossover return two is confusing and atypical. The program returned should be selected based on its fitness.

2.3.6 Implementing new types of `IFitnessMeasure` and `IFitnessCase`

A fitness case has two key roles which any implementation of `IFitnessCase` must meet. Firstly, it must return the values of the various features for this specific fitness case.

Secondly, and crucially, in a class-dependent manner, an instance of a sub-class of `IFitnessCase` must make available to a type of `IFitnessMeasure` the correct register values. This need not literally be the correct value of all of the registers — for example, if a winners-take-all algorithm is used then the information which is needed by the fitness measure is the register which should be the largest for this case. In addition to meeting these two roles a sub-class of `IFitnessCase` should probably implement a static generation method which can parse a string representation of a fitness case.

The role of a fitness measure is two-fold. Firstly it must track the fitness of some program. This is typically a double value, representing the program's overall fitness and a part of the abstract base class, `IFitnessMeasure`, but may also include augmentary information used when logging or when calculating the overall fitness.

In addition, a fitness measure is responsible for implementing the function `IFitnessMeasure::UpdateError` which takes a `RegisterCollection` (of the final values of the registers after execution of the program was terminated) and the fitness case the program was executed against. Using the problem-dependent functions and case-specific features of the fitness case, the fitness measure's measure of fitness should be updated when this method is called.

Note that the class trees rooted at `IFitnessCase` and `IFitnessMeasure` are some of the most tightly coupled code in VUWLGP. The two classes must really be developed in parallel and the reader is strongly urged to examine existing code and also read the code-oriented documentation.

2.3.7 Implementing a Fitness Environment Not Based on Cases

This kind of modification is a more substantial modification and is mentioned here as an example of the kinds of changes that might be necessary for such a modification.

It is important to note that most problems which don't appear to be fitness-case based can in fact be represented as a single fitness case, and so the existing structure can be altered quite easily by changing the implementation of `IProgram::Execute`. For example, the San Mateo trail problem [5] could be represented as a single case, and the features (the ant's sensory capacities) would obviously return different values depending on where the ant currently was. There is no reason a feature must be constant throughout an entire run of an LGP program. Similarly there is no reason that the program must be run through only once in `IProgram::Execute`.

Regardless, if a truly non-fitness case-based problem must be addressed a number of tasks would need to be completed.

1. `IProgram::UpdateFitness` would need to be altered, as it can (presumably) no longer iterate through the fitness cases in the `FitnessEnvironment`. Note that `UpdateFitness` is called from `IPopulation::EvaluateFlaggedPrograms`, which is in turn called from `IPopulation::Evolve`. Neither of these two methods should need to be updated.

2. The `FitnessEnvironment` class would also need to be changed. The methods related to fitness case iteration might be removed, and how the value of a feature was determined (it currently depends on the current fitness case) would also need modification. Similarly, the fitness environment can probably no longer be initialised by adding fitness cases individually or from a file and so new initialisation methods need to be created.

3. The fitness measure class used might need to be changed, perhaps so that the implementation of `IFitnessMeasure::UpdateError` takes a `FitnessEnvironment` instead of an `IFitnessCase`.

3 Linear Genetic Programming — An Overview

In this section LGP is briefly described. Only register-machine LGP [1] (hereafter just LGP) is considered, although VUWLGP could be customised so that other variants could be implemented. Readers are referred to [1] for discussion on other forms of LGP, and to [5] and [1] if unfamiliar with genetic programming.

In this form of LGP each program is a sequence of register machine instructions. This is typically expressed in human-readable form as C-style code. Other variants of linear genetic programming are discussed in [4], but they have not been explicitly considered in the design of VUWLGP.

Prior to the execution of an LGP program, the registers which it can read from or write to are zeroed. The features representing the objects to be classified are loaded into predefined registers. The program is executed in an imperative manner and represents a *directed acyclic graph* (DAG, see figure 2 for a simple example). This is different from tree-based GP which represents a program as a tree. Any register's value may be used in multiple instructions during the execution of the program.

Instructions in an LGP program may also be *introns* — i.e. code whose execution has no impact on the output of the program. The existence of introns can significantly improve the evolvability of a solution by allowing good building blocks to exist non-destructively in a distributed manner across several individuals, each of which has a small part of the building block. A partial building block may have a negative impact on fitness. This impact

is eliminated when it is present as an intron. As with biological evolution, introns also reduce the frequency of destructive evolutionary operations which slow the evolution of fit individuals, as defined and discussed in [7] and [6]. After the final instruction in the program has been executed the values of the registers are interpreted as the output.

```
//r[1] = r[1] / r[1];
//r[3] = cf[0] + cf[5];
//if(r[3] < 0.86539)
//r[3] = r[3] --- r[1];
r[0] = 0.453012 --- cf[1];
//r[3] = r[2] * cf[5];
r[1] = r[0] * 0.89811;
if(cf[6] < cf[1])
r[2] = 0.453012 --- cf[3];
r[3] = cf[4] --- 0.86539;
```

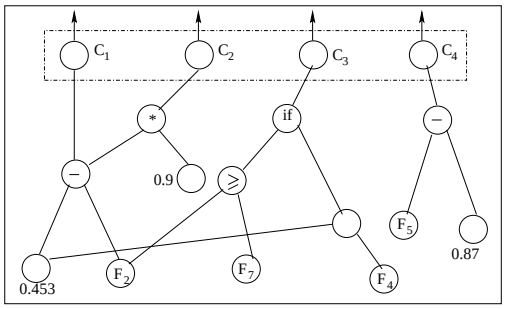

Figure 2: A very simple sample program evolved by LGP and a DAG representation of it. Structural introns are commented out with `//` and do not need to be represented in the DAG representation.

4 Summary and Future Work

VUWLGP is an ANSI C++ implementation of register-machine based linear genetic programming. In this report the design of VUWLGP and how it might be customised has been described. In addition, linear genetic programming has been briefly explained.

The most significant features and benefits of VUWLGP can be summarised as follows:

- The registers, operations (functions), program structure, fitness measurement and instructure structure are all easily modified.
- The population class can be easily augmented so that it is steady state or generational, and deme-based or monolithic.
- The type or types of the registers is generic.
- Different variants of the above elements of an operating environment are "pluggable", and can be easily mixed and matched.
- Once the operating environment (described in the four previous bullets) has been defined it is easily modified slightly for different variants of a problem being researched.

The class structure of VUWLGP allows the operating environment to be easily customised. Future versions of VUWLGP could include more implementations of the base classes, including steady state-style Population classes and state-based fitness classes (such as the San Mateo trail [5]). Similarly, different kinds of programs and instruction could be implemented and VUWLGP made more general. Finally, the generic structure is designed so that strongly-typed LGP should be implementable in an extension of VUWLGP. This might be achieved by setting the generic type of the registers to be some new user-defined abstract class which is the super class of all other types of register.

5 Acknowledgements

This research was supported by the University Research Fund (URF 6/9, Grants ID 10650) and built on research carried out by the authors while researching the Honours project [2] [3].

6 Bibliography

References

[1] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*. San Francisco, Calif. : Morgan Kaufmann Publishers; Heidelburg : Dpunkt-verlag, 1998. Subject: Genetic programming (Computer science); ISBN: 1-55860-510-X.

[2] Fogelberg, C. Linear genetic programming for multi-class classification problems. BSc (Honours) Research Project Report, School of Mathematics, Statistics and Computer Science, Victoria University of Wellington, New Zealand., November 2005.

[3] Fogelberg, C., and Zhang, M. Linear genetic programming for multi-class object classification. In *AI 2005: Advances in Artificial Intelligence: Proceedings of the 18th Australian Joint Conference on Artificial Intelligence, Lecture Notes in Computer Science, Vol. 3809.* (Sydney, Australia, December 2005), S. Zhang and R. Jarvis, Eds., Springer, pp. 369–379.

[4] Kantschik, W., and Banzhaf, W. Linear-tree GP and its comparison with other GP structures. In *Genetic Programming, Proceedings of EuroGP'2001* (Lake Como, Italy, 18-20 Apr. 2001), J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds., vol. 2038 of *LNCS*, Springer-Verlag, pp. 302–312.

[5] Koza, J. R. *Genetic programming : on the programming of computers by means of natural selection*. Cambridge, Mass. : MIT Press, London, England, 1992.

[6] Nordin, P., Banzhaf, W., and Francone, F. D. Introns in nature and in simulated structure evolution. In *Bio-Computation and Emergent Computation* (Skovde, Sweden, 1-2 Sept. 1997), D. Lundh, B. Olsson, and A. Narayanan, Eds., World Scientific Publishing.

[7] Nordin, P., Francone, F., and Banzhaf, W. Explicitly defined introns and destructive crossover in genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* (Tahoe City, California, USA, 9 July 1995), J. P. Rosca, Ed., pp. 6–22.