LGP Documentation

Jed Simson

Oct 25, 2021

Contents:

This project is an implementation of Linear Genetic Programming (LGP) as outlined by Brameier, M. F., & Banzhaf, W. (2007)[1].

The package intends to provide an easily-accessible platform for using LGP to solve problems, through a system that offers a cross-platform, modern, and robust interface.

This document includes an introduction to the concepts used in LGP, details regarding the design of this system, and various tutorials and examples.

For those unfamiliar with LGP, it is recommended to start with the *LGP* section to gain an understanding for the rest of the documentation. Otherwise, the *Guide* details how to get started using the system.

This system was developed as partial fulfilment towards a Bachelor of Computing and Mathematical Sciences (Honours) degree at the University of Waikato in 2017. The accompanying report can be viewed here.

[1] Brameier, M. F., & Banzhaf, W. (2007). Linear Genetic Programming. Springer Science & Business Media. https://doi.org/10.1007/978-0-387-31030-5

CHAPTER 1

Linear Genetic Programming

This section provides background information regarding Linear Genetic Programming (LGP) which is necessary to get started using this system. The concepts used in LGP as described here are given by Brameier, M. F., & Banzhaf, W. (2007)[1].

It should be noted that this section is not designed to be a complete reference for LGP, it merely gives enough information to get started using this system.

The fundamental concept of Genetic Programming (GP) borrows ideas from evolution in order to take a set of randomly generated programs and train them towards a particular target function through selection, reproduction, and mutation.

The process has a large amount of randomness to it, in that the initial program solutions are random and then those initial programs are randomly selected, reproduced and mutated to form other solutions.

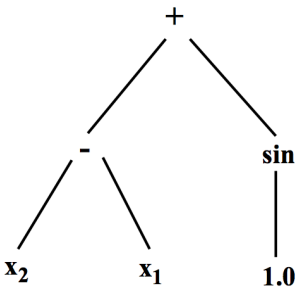
1.1 Representation

1.1.1 Programs

Linear Genetic Programming (LGP) operates with imperative programs that consist of a variable-length *sequence of instructions* that operate on and manipulate the contents of a *set of registers*.

This is significantly different to other Genetic Programming approaches, such as tree-based GP, which represents programs as a tree where the interior nodes perform operations on the values stored in the leaf nodes. The figure below illustrates the difference between a tree-based GP program and an LGP program.

[1] Brameier, M. F., & Banzhaf, W. (2007). Linear Genetic Programming. Springer Science & Business Media. https://doi.org/10.1007/978-0-387-31030-5




```
void gp(double r[6]) {
    r[2] = r[1] - r[0];
    r[3] = sin(r[6]);
    r[2] = r[2] + r[3];
}
```

In this example, the registers contain the values $\{x1, x2, 1.0, 1.0, -1.0, 0.0, 1.0\}$ which map directly to the zero-index based registers. The LGP program's output is taken from `r[2]` making the two program's outputs equivalent (i.e. the function $f(x_1, x_2) = (x_2 - x_1) + sin(1.0)$).

In essence, an instruction in an LGP program performs an *operation* on a set of *operand (source) registers* and assigns the result of that operation to a *destination register*.

Generally, instructions operate on one or two registers, but our LGP system provides the ability to define instructions that operate on any number of operand registers.

Note: A higher number of operands/operators does not necessarily mean that better programs will be produced, as these instructions would represent more complex expressions that are less able to be manipulated by genetic operations during the evolution process.

1.1.2 Registers

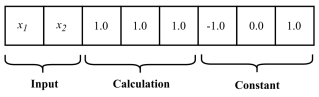
In LGP a number of variable registers, the *register set* are provided to each program. The register set is split into three sections that represent the three different types of registers that are used by a program:

Input Registers Hold program inputs before execution. Generally these will be loaded from a fitness case that the program is being evaluated on.

Calculation Registers A variable number of additional registers used to facilitate calculations. These registers should be loaded with some constant value before each time a program is evaluated.

Constant Registers A number of registers which are loaded with a constant value and are write-protected, so that they are always available to an LGP program.

One or more of the input or calculation registers are defined as *output register(s)*. This provides another benefit over Tree-Based GP as the imperative structure allows the use of multiple program outputs.



1.1.3 Instruction Set

The *instruction set* given to an LGP system defines the programming for programs that are evolved within that system.

Generally, an instruction set would include instructions for Arithmetic, Exponential, Trigonometric, and Boolean operations but it is possible in our system to define instructions as necessary for the problem domain (for example, vector operations).

With a certain probability, an instruction can have a constant register as one of its operands, indicating the application of that instructions operation to a constant value.

When using Genetic Programming, there are two invariants of the programs that need to be maintained — *syntactic correctness* and *semantic correctness*.

Syntactic Correctness It must be ensured that when modifying a program or combining two programs, that the operation creates a valid program. In LGP, this is done by ensuring that combination does not combine parts of instructions, they are treated as atomic. Furthermore, mutation can only change certain parts of instructions, for example, an instructions operation cannot be changed to a register.

Semantic Correctness Some operations have undefined behaviour for certain inputs, and this needs to be addressed to ensure programs are valid when executed. This is generally done by creating *protected* versions of operations that have undefined behaviour, which give a high constant value when used on invalid input ranges. This has the effect of penalising programs which use these instructions.

In our implementation, because it is possible to define instructions as needed, this allows for instructions that introduce *side-effects* to some environment the problem is contained within — for example, a problem that involves moving some agent through a space.

The ability for LGP to find a solution is dependent upon the expressiveness of the instruction set used — however, the dimension of the search space increases exponentially with the number of instructions available to an LGP program.

1.1.4 Control Flow

Branching

Branching is an important concept in programming languages, and it is similarly powerful in the context of Genetic Programming.

Programs in LGP have a linear control flow and a data flow which is representable as a directed graph. Using conditional branches, the linear control flow can be different for various input situations, which can have an effect on the inherit graph created from the data flow.

In LGP (and in this implementation), conditional branches are evaluated such that the instruction following the branch is executed only if the branch is true. That is to say, if the branch is false then a *single* instruction will be skipped.

The logical AND connection between two branches can be created with two branches directly next to each other in the programs sequence:

```
if (<cond1>)
if (<cond2>)
    <oper>
```

That is, `<oper>` will only be reached when both `<cond1>` **AND** `<cond2>` evaluate to a logically true value.

The logical OR connection can be represented in a similar way:

```
if (<cond1>)
    <oper>
if (<cond2>)
    <oper>
```

In this case, `<oper>` will be evaluated either when `<cond1>` **OR** `<cond2>` evaluates to a logically true value.

These fairly basic concepts facilitate a basic branching architecture, but to allow for more complicated control flows more advanced concepts are needed.

Some examples of more advanced branching concepts and how they are realised in LGP can be found in the following:

Nested Blocks

```
if (<cond1>)
    <...>
endif
```

Any instructions between the initial conditional and the *endif* instruction are executed when the condition is true, allowing a nested code block that may care for certain input situations.

Labeled Blocks

```
if (<cond1>) goto <label X>
<...>
<label X>
```

Similar to a *goto* instruction in the C programming language, this form of branch allows for a chunk of instructions to be skipped depending on the value of the condition.

Iteration

There are two main iteration concepts in LGP - *conditional loops* and *finite loops*.

Conditional Loops Similar to a `while` loop like that in most imperative programming languages (e.g. C, Java), a conditional loop describes a loop that jumps backwards in a program and evaluates a condition to determine whether to stay in the loop. One problem with these kind of loops is that they are prone to becoming infinite in terms of their iterations - which is impossible to detect due to the Halting problem. A solution to this is to limit the number of instructions that can be executed for an LGP program (hence artificially limiting the runtime)

Finite Loops Similar to a `for` loop, a finite loop has a finite number of iterations as described by the instruction itself. This means that the instructions in the loop body will be executed some finite amount of times.

1.2 Evolution

1.2.1 Evolutionary Algorithm

In LGP (and other GP techniques) there are three main phases to the evolutionary algorithm — *Initialisation*, *Selection*, *Variation*.

Usually LGP uses **Algorithm 1**, which is taken from (Brameier, M., Banzhaf, W. 2007). Our system provides an implementation of this algorithm, but also allows for other evolutionary models to be used to allow extension of the core LGP principles.

Algorithm 1 (LGP algorithm)

1. Initialize a population of random programs.
2. Randomly select 2 × n individuals from the population without replacement.
3. Perform two fitness tournaments of size n.

4. Make temporary copies of the two tournament winners.
5. Modify the two winners by one or more variation operators for certain probabilities.
6. Evaluate the fitness of the two offspring.
7. If the currently best-fit individual is replaced by one of the offspring validate the new best program using unknown data.
8. Reproduce the two tournament winners within the population for a certain probability or under a certain condition by replacing the two tournament losers with the temporary copies of the winners.
9. Repeat steps 2. to 8. until the maximum number of generations is reached.
10. Test the program with minimum validation error again.
11. Both the best program during training and the best program during validation define the output of the algorithm.

The main phases of the evolutionary algorithm as used by LGP will be described to allow for an understanding so that custom models can be used.

1.2.2 Initialisation

The evolutionary algorithms first phase is to build an initial population of individuals, which is generally done by randomly creating valid LGP programs.

LGP defines an upper and lower bound on the *initial program length*, so that the length of programs created is a randomly chosen from within these bounds with a uniform probability.

It is important to find a balance between too short and too long initial programs, as short initial programs can cause an insufficient diversity of genetic material. Comparatively, initial programs that are too long may reduce their variability significantly in the course of the evolutionary process.

1.2.3 Selection

Selection refers to the process of choosing individuals from a population to be used in the next population after they undergo *variation*.

Generally, the selection method used should be a function of the fitness of each individual so that the population maintains some amount of fit and unfit individuals, in order to increase the diversity of the population.

Some popular selection methods are Tournament Selection, Fitness Proportionate Selection, and Reward Based Selection.

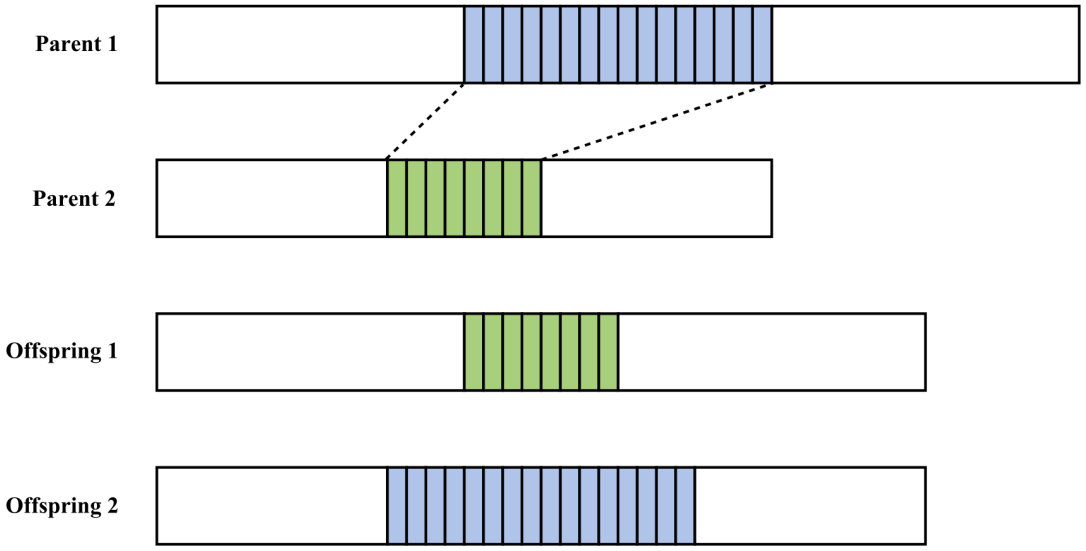
1.2.4 Variation

Genetic Operators alter the programs in an LGP population in some way, usually by either adding, removing or exchanging instruction(s), or altering the effect an instruction has — so called macro and micro mutations respectively.

Macro Operations

Macro operations are generally used for *recombination* and *mutation* of individuals and operate at the instruction level (i.e. they treat instructions as atomic units and don't modify them directly).

Recombination involves a way of combining two individuals from a population, usually using some form of crossover. This has the effect of altering the two individuals length through sharing of genetic material (i.e. the segments of instructions that are exchanged). The operation of linear two-point crossover is illustrated below in the context of two LGP programs:



Mutation involves a way of altering a single individual and either increasing or decreasing the length of the program. This is usually done by adding/removing one or more instruction(s) from the program.

Micro Operations

Micro mutations are performed below the instruction level and consist of ways to:

- Change an instruction's operation.
- Change a register used by an instruction.
- Change a constant value used by an instruction.

1.3 Execution

1.3.1 Evaluation

In LGP — as in other forms of Genetic Programming — the cost of computation is dominated by the evaluation of the programs evolved on the fitness cases defined by the problem.

A program must be executed for each fitness case, and if evaluating the fitness of a problem is quite computationally demanding this can cause the majority of the time to spent evaluating fitness.

One method LGP uses to speed up the execution of programs is related to the linear representation of programs. This representation leads to instructions in a program that are not *effective*, that is, they don't effect the programs output.

These instructions can be found efficiently and removed from the program so that when evaluating the program, only those instructions which directly effect the output of the program are executed.

This unique aspect of LGP can lead to a drastic performance improvement as on average, it decreases the number of instructions that are executed for each program, which is advantageous when there are a large number of fitness cases or evaluating a programs fitness is a computationally demanding task.

To test a program on a set of fitness cases, the basic steps are:

1. Load a set of inputs from a fitness case into the programs input registers.
2. Execute the program with the inputs given by executing each effective instruction.
3. Store the output of the program for later fitness evaluation.
4. Go to step 1. while there are still fitness cases left, otherwise go to step 5.
5. Compare each fitness cases desired output with the output predicted by the program for that fitness case using some fitness function.

1.3.2 Translation

Because the primary objective of GP is to find the best program that generalises a solution to the problem, it is important that the program found by LGP can actually be used in a context outside of the LGP system.

Internally, an LGP program is interpreted using a simple virtual machine that can execute a programs instructions. It is not advantageous to only programs generated by LGP to be used within this context, so programs need to be translated to another format.

In typical LGP, all instructions are able to be translated to equivalent instructions in the C programming language, but our system allows for custom representations to be defined so that a program can be translated to any format required (e.g. assembly, other imperative languages, etc).

CHAPTER 2

Guide

This document describes the design of the LGP system and how to use it to get started with LGP.

It is assumed that the reader has prior familiarity with LGP, or has read the section on *Linear Genetic Programming*.

It should be noted that the code examples provided in this documentation are not extensive and a good reference for example usages is the `nz.co.jedsimson.lgp.examples` package provided. These examples can be viewed in the LGP-examples repository.

For a complete, low-level API reference, see the dokka API documentation, as this document will primarily discuss the concepts of the API and how to use them at a high level.

2.1 Environment

The first step in using the LGP system is to build an environment for the problem being solved.

2.1.1 Overview

This environment acts as a central repository for core *components* of the LGP system, such as configuration information, modules for various operations performed during evolution, etc.

It can be thought of as the *context* in which the LGP system is being used, as the environment used will directly influence the results.

The components needed to build an environment are split into three main categories:

1. *Construction Components*
2. *Initialisation Components*
3. *Registered Components*

The order the components are enumerated in *is* important, as the environment needs certain components in order to be built, whereas other components depend on the environment being constructed first. The order of components will be discussed further in the following sections.

2.1.2 Construction Components

These components are required when building an `Environment` instance and should be passed to the constructor (hence construction components). These components form the base information required to resolve any further components.

To build an environment, the following construction components are required:

- `ConfigurationLoader`
- `ConstantLoader`
- `OperationLoader`
- `DefaultValueProvider`
- `FitnessFunction`

Note: To find further information about these components, see the API documentation.

These components are primarily those related to loading information into the environment (at initialisation time), or functionality that is used throughout the system but is customisable.

Example

Building up an `Environment` instance with the correct construction components is the main initiation step to getting started using the LGP system, and as such requires a bit of dependency gathering.

Note: The type for the various loaders is specified explicitly in the example, but generally the type will be inferred from the arguments when using the Kotlin API. This examples uses the *Double* type, meaning that programs generated will operate on registers containing Double-precision floating-point format numbers.

```
// Configuration.
// Here, we load configuration information from a JSON file.
val configLoader = JsonConfigurationLoader(
    filename = "/path/to/some/configuration/file.json"
)

// Pre-load the configuration so we can use information from it.
val config = configLoader.load()

// Constants.
// Load constants from the configuration file
// (although they could come from anywhere).
val constantLoader = GenericConstantLoader<Double>(
    constants = config.constants,
    // Parse the strings in the configuration file as doubles.
    parseFunction = String::toDouble
)

// Operations.
// We're using the operations specified in the config file.
val operationLoader = DefaultOperationLoader<Double>(
    operationNames = config.operations
)
```

(continues on next page)

(continued from previous page)

```
// Default register value provider.
// Constant value provider always returns the value given
// when it is initialised.
val defaultValueProvider = DefaultValueProviders.constantValueProvider<Double>(1.0)

// Fitness function. We'll use the classification error
// implementation from the fitness functions module.
val ce = FitnessFunctions.CE({ o ->
    // Map output to class by rounding down to nearest value
   kotlin.math.floor(o)
})


// We've declared all our dependencies, so we can build an LGP
// environment. When constructing an environment, any
// initialisation components will be resolved.
val env = Environment<Double, Outputs.Single<Double>>(
    configLoader,
    constantLoader,
    operationLoader,
    defaultValueProvider,
    // We must pass a function that can provide the FitnessFunction.
    fitnessFunctionProvider = { ce }
)
```

This will create an environment with the construction components given and begin the process of loading any initialisation components.

Note: The `Environment` constructor offers an optional parameter `randomStateSeed` which can be used to provide a fixed seed to the systems random number generator. The parameter accepts either a long-type value (e.g. 1, -24, etc.), which will be used as a fixed seed; or `null`, which tells the system to randomly seed the RNG. By default, the system will use a randomly generated initial seed.

2.1.3 Initialisation Components

These components are automatically loaded by an environment when a set of suitable construction components have been given. The components are generally associated with a `ComponentLoader` and are a sort of *global state* that isn't affected by the LGP system, for example:

- Configuration
- Constants
- Operations
- Register Set
- Random State

The Register Set is slightly different in that it depends on information provided by the construction dependencies and is initialised internally as a *global reference* register set, so that programs can acquire a fresh register set at any time.

Nothing special needs to be done for initialisation components — provided that the construction components given were valid, the components will be automatically loaded as appropriate and operate behind-the-scenes.

During initialisation, the environment will construct a random number generator instance. This RNG is a globally accessible value and should be used whenever a RNG is required. This allows the system to provide determinism where it is required.

2.1.4 Registered Components

Registered components are essentially those that have a circular dependency graph.

That is, a registered component requires a reference to the environment in order to operate, but the environment also needs a reference to the component itself so that it can be accessed within the context of the LGP system — hence these components have to be resolved after the environment has been built.

Generally, registered dependencies will be custom implementations of core components used during the evolution process, such as custom generation schemes for instructions and programs, or custom search operators.

The reason these components generally have a dependency on the environment is that they are designed to be as flexible as possible, and thus enabling custom components access to the entire environment is useful.

When registering these components, it is done by associating a module type (i.e. the type of component) with a builder for that module. A builder is really just a function that can build a new instance of that module. The builder function takes a single argument of type `Environment`, which allows the module to be given an appropriate environment reference when it is used. In the general case, the template `{ environment -> Module(environment) }` will be sufficient. It is important to note that the environment argument does not refer to the previously constructed environment, it simply defines the way in which the module is built (e.g. it needs an environment to be built).

Example

To illustrate how registered components are used — continuing from the above example.

```
...

// Our environment.
val env = Environment<Double>(
        configLoader,
        constantLoader,
        operationLoader,
        defaultValueProvider,
        fitnessFunction = ce
)

// Now that we have an environment with resolved construction
// and initialisation dependencies, we can resolve the
// registered dependencies.

// Build up a container for any modules that need to be registered.
// The container acts as a way for the environment to resolve
// dependencies in bulk.
val container = ModuleContainer<Double>(
    modules = mapOf(
        CoreModuleType.InstructionGenerator to
        { environment -> BaseInstructionGenerator(environment) },

        CoreModuleType.ProgramGenerator to
        { environment -> BaseProgramGenerator(environment) },

        // More module registrations as necessary
```

(continues on next page)

(continued from previous page)

```
        ...
    )
)

// Inform the environment of these modules.
env.registerModules(container)

// Alternatively, we can register modules one-by-one.
env.registerModule(
    CoreModuleType.SelectionOperator,
    {
        environment -> TournamentSelection(
            environment,
            tournamentSize = 2
        )
    }
)
```

With all components resolved, the environment is ready to be used for the main process of evolution: execution of the evolutionary algorithm.

Note: It is only necessary to provide a builder for modules types that are guaranteed to be requested from the environment (i.e. they are a dependency)

If the environment is being used by some custom consumer, then it is permitted to only provide builders for module types that it will request.

If a module is requested that hasn't been registered with a builder then an exception detailing the missing module will be thrown.

2.1.5 API

See nz.co.jedsimson.lgp.core.environment.

2.2 Evolution Models

Once an environment has been defined for the problem to be solved, the next step is to set up an evolutionary model using that environment.

2.2.1 Overview

An `EvolutionModel` describes the way that the modules provided to the environment are used to evolve a *solution* to the problem.

All this really means is that an evolution model is an implementation of some evolutionary algorithm (EA) that produces solutions — such as a steady-state or generational algorithm.

The system allows custom models to be created so that the evolution process can be adapted and tuned to the problem being solved. In general, the models provided by the `Models` module will provide good performance, as the parameters offer a high degree of adjustment through the environment being used.

To use an EA is as simple as providing the model with an environment and training on a data set for the particular problem being solved. Once the EA has been trained, it can be tested on an arbitrary data set in order to form a prediction.

Note: It must be ensured that the environment built provides all the components that the evolution model requires. Because the model has complete access to the environment, it can make use of any component the environment is aware of.

Generally, the model will work by using an evolutionary algorithm to train a population of individuals. The best individual from training will be used by the model to form predictions when testing using the model.

It is possible to use a model directly to solve a problem, but in general it is better to define a `Problem` as described in the *next section*.

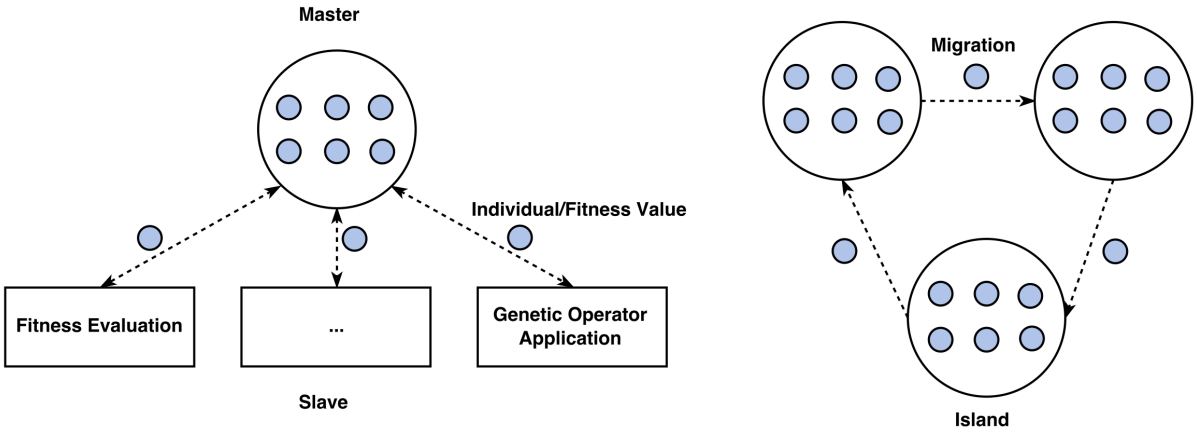
2.2.2 Provided Models

The system offers three built-in EAs which can be used to solve problems without the need to define custom logic. The three models — `SteadyState`, `MasterSlave`, and `IslandMigration` — are contained in the `Models` module.

The `SteadyState` model is the most basic algorithm and offers fairly conservative performance.

Building upon the `SteadyState` algorithm, the `MasterSlave` EA adds parallel processing to the base algorithm in an attempt to improve runtime performance. Generally, the `MasterSlave` EA will provide better average runtime performance, but will not have any effect on the quality of the solutions.

The `IslandMigration` technique is designed to alleviate issues with limited diversity that can arise as the EA is executed. The population of solutions is split across multiple islands, and solutions are exchanged between islands at a regular interval. Problems that suffer from early convergence on local optima can benefit from the increased diversity the `IslandMigration` EA gives.



Note: For those interested in a detailed comparison of the performance characteristics of the three techniques, feel free to view the accompanying thesis *(page 56)*.

Example

An environment provides a context for evolution, and we can build a model within that environment easily:

```
// The environment from the previous section
// without any of the registered dependencies.
val env = Environment<Double, Outputs.Single<Double>>(
    configLoader,
    constantLoader,
    operationLoader,
    defaultValueProvider,
    fitnessFunctionProvider = { ce }
)

// Register the modules that are needed to use
// the model we wish to use (i.e. every core module type).
val container = ModuleContainer(
    modules = mutableMapOf(
        // Any modules the system needs, as determined by the model
        ...
    )
)

environment.registerModules(container)

// Build a steady-state EA around this environment.
val model = Models.SteadyState(environment)

// Define a data set to be used for training. What this data
// set contains will depend on the problem.
val trainingDatasetLoader = DatasetLoader<Double> {
    // Could be loaded from a file or built directly.
    ...
}

// Train the model on the training data set.
val result = model.train(trainingDatasetLoader.load())

// Output the fitness of the best individual from training.
println(result.best.fitness)

// To perform a prediction using the trained model is easy:
// Define a data set to be used for testing. This data set
// will generally be different to that used for training in order
// to evaluate the solutions generalisation.
val testDatasetLoader = DatasetLoader<Double> {
    ...
}

// Gather the models predictions for this data set.
val predictions = model.test(testDatasetLoader.load())
```

2.2.3 API

See nz.co.jedsimson.lgp.core.evolution.model.

2.3 Problems and Solutions

The combination of an environment and evolution model provides everything necessary to start searching for solutions for a problem.

However, in the name of clean code the system provides a few mechanisms which can be used to make the task of defining the problem easier and clearer.

2.3.1 Overview

Problem

A `Problem` encapsulates the details of a problem and the components that can be used to find solutions for it.

At the highest level a problem is a wrapper that has a set of data attributes (name, description, training/testing datasets), and a collection of dependencies.

The general operation of a problem is to define the components needed by filling in the problem skeleton:

- Name
- Description
- Configuration Loader
- Constant Loader
- Operation Loader
- Default Value Provider
- Fitness Function
- Registered Modules

It may be noticed that these are essentially the components required to build an `Environment`. This is no coincidence, as the next step of filling out a problem's skeleton is to implement a method to initialise an environment for that problem — provided by the `initialiseEnvironment()` method.

Following that, a model for the problem should be defined and initialised using the `initialiseModel()` method. This method should build an EA around the environment initialised previously.

Finally, a method for solving the problem can be defined. This functionality can use the environment and model of the problem to search for solutions. The return value of the method is left open so that the solution can be adapted for the problem as necessary. The `solve()` method contains the skeleton necessary for implementing this final part of the problem. For example, one could save results directly to a file or perform analysis of the results to produce plots.

Note: It must be ensured that the environment built provides all the components that the evolution model requires. Because the model has complete access to the environment, it can make use of any component the environment is aware of.

Solution

A `Solution` to a problem is left as open as possible to allow for arbitrarily complex solutions. In general, a solution will contain the result of a prediction using the model trained for the problem, but there are situations where it makes sense to return multiple predictions or statistics.

Example

The `lgp.examples` package provides examples of how to define a problem and how solutions to that problem can be obtained. The package is available for viewing on GitHub.

2.3.2 API

See nz.co.jedsimson.lgp.core.evolution

2.4 Trainers

Although an evolution model can be used directly to perform evolution, generally the model will be re-used multiple times as different results are achieved with different runs due to the random nature of LGP.

2.4.1 Overview

The concept of trainers provides a common interface for creating ways to evaluate a model. A `Trainer` is provided with an environment and a model and implements some logic to evaluate that model depending on the situation.

Generally this will involve training a number of instances of a model and evaluating each model. This way we can gather statistics about how models are performing on average.

The `nz.co.jedsimson.lgp.core.evolution.training` module provides the base concept as well as trainers for various situations. Generally, a trainers main task is to train a set of models and gather results for each model evaluation so that consumers can get information about the different evaluations.

Example

A typical way to use an evolutionary algorithm is to evaluate it multiple times and use that to find an *average best fitness* value to ensure that the result of a *single run* wasn't a one-off occurrence.

Evaluating the model multiple times is supported by two built-in trainers that differ in the method of evaluation - *parallel* or *sequential*. Both these implementations will train multiple instances of the model and provide training results for each.

`DistributedTrainer` spawns a set of threads and trains the models in parallel, which results in a faster runtime as multiple runs occur at once. In contrast, `SequentialTrainer` trains the model in a single thread and only when one model is trained is another training session begun.

To train 10 instances of the model from the previous section in a parallel manner we can use the `DistributedTrainer`:

```
// Build a trainer that will evaluate 10 parallel instances of the model.
val trainer = Trainers.DistributedTrainer(
    environment,
    model,
    runs = 10
)

// Gather the results.
// Assuming we have a data set loader as in previous sections.
val result = trainer.train(trainingDatasetLoader.load())
```

(continues on next page)

(continued from previous page)

```
// Output the best (effective) program for each run.
result.evaluations.forEachIndexed { run, evaluation ->
    println("Run ${run + 1}")

    evaluation.best.effectiveInstructions.forEach(::println)

    println("\n(fitness = ${evaluation.best.fitness})")
}
```

As output, we will be given the effective programs of 10 individuals which were the best as trained by each model. From this we could compute the average fitness of the best individuals trained by the model to gain a metric of how good the models solutions are on average.

Alternatively, the trainers offer an asynchronous interface which can be used to prevent blocking the main thread while the training process is completed.

```
// Build a trainer that will evaluate 10 parallel instances of the model.
val trainer = Trainers.DistributedTrainer(
    environment,
    model,
    runs = 10
)

// Gather the results asynchronously.
// Assuming we have a data set loader as in previous sections.
val job = trainer.trainAsync(trainingDatasetLoader.load())

// Training asynchronously allows us to subscribe to progress updates,
// allowing communication between the initiator and executing thread(s).
job.subscribeToUpdates { update ->
    println("training progress = ${update.progress}")
}

// Wait for the execution to complete
val result = job.result()

// Output the best (effective) program for each run.
result.evaluations.forEachIndexed { run, evaluation ->
    println("Run ${run + 1}")

    evaluation.best.effectiveInstructions.forEach(::println)

    println("\n(fitness = ${evaluation.best.fitness})")
}
```

The trainer will use the current co-routine context to spawn new co-routines.

Note: Both built-in trainers have guarantees in place to ensure that they can provide determinism. In the case of the `DistributedTrainer`, each thread has its own RNG instance, and thus the multi-threaded nature does not compromise the determinism guarantee.

2.4.2 API

See nz.co.jedsimson.lgp.core.evolution.training.

2.5 Modules

To make the LGP system malleable to different problem domains, it is designed with the concept of modules. Modules represent a core piece of functionality that the system can use to perform the various operations it needs to.

Any part of the system that is modular can be extended and custom implementations can be provided, as long as the system is made aware of these modules in some way.

There are some restrictions on how modules can be used and what parts of the system are modules, and the following sections of the guide will detail the operation of the different modules the system uses. Note that the evolutionary algorithm is a module, but as it has been previously described it will be omitted in the subsequent sections.

2.5.1 Operations

Operations define a way to map the value in a register to some other value based on the properties of that operation.

These operations change the values of the registers during the execution of an LGP program and directly influence the final result.

Built-In Operations

The lgp.lib.operations module defines a set of built-in operations that can be used in a few common situations, but the purpose of this document is to describe how the `Operation` API can be used to build custom operations suitable to a particular problem domain.

Basics

Operations are an abstract concept in the API, which at their core are composed of an *Arity* and a *Function*.

The execution of an operation means that an operations' function is applied to a set of n arguments, where n is equal to the operations' arity.

> **Warning:** An operation should validate that the number of arguments it is given to apply its function to matches its arity.

Arity

An `Arity` defines a way for an operation to specify how many arguments it expects.

Generally, the built-in `BaseArity` which provides values for unary and binary functions will be suitable to most problems, but to allow for further levels of arity to be handled, the `Arity` interface can be implemented.

Example

It is simple to define an arity suitable for operations with three operands:

```
enum class CustomArity : Arity {
    Ternary {
        // Each enum type needs to override the number property.
        override val number: Int = 3
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Function

The `Function<T>` type is really a type alias for `(Arguments<T>) -> T`.

That is to say, a function type takes a collection of arguments of some type `T` and maps those arguments to a value in the domain of `T`.

Because functions have a simple interface (they are really just a lambda function), it is straight-forward to define custom functions. The only *gotcha* with functions is that they are disjoint from the arity, so care must be taken to ensure that when executing the function at the operation level, the number of arguments is checked. This will be made clearer in the *Operation* section.

Example

For now, lets imagine a function that takes 3 arguments and computes the function (2.1).

$$f(x, y, z) = x + y - z \tag{2.1}$$

This function is going to operate on double values to keep it simple, but the function could operate on values of any type.

To translate this into a form the LGP system understands is straight-forward:

```
val ternaryFunc = { args: Arguments<Double> ->
    // Here we have access to the arguments. We can just assume
    // that 3 arguments have been given and let the consumer of
    // this function deal with any validation logic.

    args.get(0) + args.get(1) - args.get(2)
}
```

In our case, the arguments don't have names and are simply positional but the functionality is the same.

Clearly, this function is not one that would be particularly useful, but it is meant to demonstrate how easy it is to define a custom function to be used in the context of an operation.

Operation

As described earlier, operations are really just a composition of an `Arity` and a `Function<T>`.

To provide an implementation of an operation, an arity and function must be given as constructor parameters, but additional logic is required to complete the implementation.

The `information` field of an operation is essentially an object that provides some information about the operation, since operations are modules in the LGP system.

The `representation` field of an operation expects some string that describes the function, so that it can be printed. This is important when exporting an operation in the context of an instruction.

Furthermore, the function `execute(arguments: Arguments<T> ): T` must be overridden. This function is used to apply the operations function to the given arguments. This method is where any validation logic to ensure that the number of arguments given matches the arity should be done.

Example

Let's finish off the example by using our ternary arity and ternary function to define a ternary operation that operates on double values, and then building an instruction that uses that operation.

Starting with a class definition that provides the correct dependencies to the base `Operation` class:

```
class TernaryOperation : Operation<Double>(
    // Our arity for operations with 3 operands
    arity = CustomArity.Ternary,

    // Our function, x + y - z
    func = ternaryFunc
)
```

This means that a `TernaryOperation` is an `Operation` that applies `ternaryFunc` to a set of double values, with the number of elements in the set determined by `CustomArity.Ternary`.

Next, the actual implementation of the base class:

```
// Provide some description of this module.
override val information = ModuleInformation(
    description = "An operation for performing a " +
                  "custom ternary function."
)

// Provide a way to represent this operation.
// The way this representation is consumed should
// be defined by the type of instruction that uses
// this operation.
// In this case we provide a simple format string.
override val representation = "%s + %s - %s"

// The core method that applies this operations
// function to a set of arguments.
override fun execute(arguments: Arguments<Double>): Double {
    return when {
        // Short-circuit
        arguments.size() != this.arity.number -> {
            throw ArityException(
                "TernaryOperations takes 3 argument but " +
                "was given ${arguments.size()}."
            )
        }
        else -> this.func(arguments)
    }
}
```

Putting this all together gives us a custom operation that operates on 3 double values and performs the function (2.1):

```
class TernaryOperation : Operation<Double>(
    arity = CustomArity.Ternary,
    func = ternaryFunc
) {

    override val information = ModuleInformation(
        description = "An operation for performing a " +
                      "custom ternary function."
```

(continues on next page)

(continued from previous page)

```
    )

    override val representation = "%s + %s - %s"

    override fun execute(arguments: Arguments<Double>): Double {
        return when {
            arguments.size() != this.arity.number -> {
                throw ArityException(
                    "TernaryOperations takes 3 argument but " +
                    "was given ${arguments.size()}."
                )
            }
            else -> this.func(arguments)
        }
    }
}
```

API

See lgp.core.evolution.instructions.

2.5.2 Instructions

An Instruction is a vital component of an LGP program. They provide the functionality to allow programs to compute values and are executed in a sequential order when evaluating a program.

Overview

Instruction

An `Instruction` is composed of an `Operation` and information about the operands of the instruction. Whereas an `Operation` has a function that it uses to transform a given number of arguments, an `Instruction` has a destination register, a set of operand registers and an `Operation`. The instruction can be executed, which involves applying the operation to the operand registers and storing the result in the destination register.

The system provides a built-in instruction type (`BaseInstruction`) for instructions that have a single output register (the most common case of instruction). Where necessary, one can implement custom instructions that may need custom logic — for example, gathering sensory data from the environment.

The `BaseInstruction` type offers a C-style instruction based representation (suitable for export during the translation process). For example, an instruction that adds two operands together and stores the result in an output register would be output as `"r[1] = r[1] + r[2]"`.

Note: The `BaseInstruction` type is found in the `nz.co.jedsimson.lgp.lib` package which resides in the LGP-lib repository.

Instruction Generator

One of the important modules required to perform evolution is the `InstructionGenerator`. This module is responsible for providing the logic necessary to create instructions that build up valid programs in LGP. Like a tree-

based GP approach, there a multiple techniques for creating instructions and thus the implementation used needs to be modular to allow different schemes to be utilised.

The system offers a built-in `InstructionGenerator` (`RandomInstructionGenerator`) which is capable of producing a endless, random stream of new `BaseInstruction` instances.

Note: The `RandomInstructionGenerator` type is found in the `nz.co.jedsimson.lgp.lib` package which resides in the LGP-lib repository.

API

See nz.co.jedsimson.lgp.core.program.instructions for details on the `Instruction` and `InstructionGenerator` APIs.

For the API of the built-in modules — `BaseInstruction` and `RandomInstructionGenerator` — refer to nz.co.jedsimson.lgp.lib.

2.5.3 Programs

Overview

Genetic programs in the system are represented by the `Program` class. Programs are implemented as modules meaning that some of their logic is open to customisation.

Despite the fairly unrestricted interface of the program modules, LGP describes a particular form of program representation which this system adhere to. The invariants are that a program is comprised of a sequence of instructions and has a set of registers made available to it. The program interface also exposes a method to allow for its effective program to be found.

The `Program` interface is designed this way in the name of flexibility; instead of restricting the shape and operation of the component, the individual program logic can be customised to allow for situations where simply executing the instructions is not enough. For example, a control problem may involve moving a robot through an environment. In this case, instructions might be commands such as move left or rotate 90°, and the program would be responsible for gathering input data from the robot's environment and providing it to the instructions.

The built-in program implementation (`BaseProgram`), provides a simple program interface which can handle both single-output and multiple-output scenarios, depending on the output resolver used.

Instructions of the program are executed in sequential order and there is support for branching (i.e. conditional instructions). Furthermore, the `findEffectiveProgram()` method implements the intron elimination algorithm as outlined by Brameier, M. F., & Banzhaf, W. (2007)[1].

The interaction between instructions and programs can be leveraged through the modular interface to adapt the system to the needs of an individual problem.

Note: The `BaseProgram` type is found in the `nz.co.jedsimson.lgp.lib` package which resides in the LGP-lib repository.

---

[1] Brameier, M. F., & Banzhaf, W. (2007). Linear Genetic Programming. Springer Science & Business Media. https://doi.org/10.1007/978-0-387-31030-5

Program Generator

Program generation is facilitated by the `ProgramGenerator` class. Again, in the name of flexibility, this is a modular component as the generation scheme for initial programs may vary (random, effective, maximum-length, constant-length, and variable-length initialization techniques are acknowledge in the literature).

The program generator is expected to act as a stream of programs so that other components in the system can continuously generate new programs.

Two built-in program generators are offered:

1. `RandomProgramGenerator` creates a random, endless stream of programs.
2. `EffectiveProgramGenerator` creates a random, endless steam of effective programs (i.e. they are guaranteed to have an effect on the output register(s)).

Properties of the initial programs generated by each implementation can be tuned through the environment definition.

Note: The `BaseProgram` type is found in the `nz.co.jedsimson.lgp.lib` package which resides in the LGP-lib repository.

Program Translator

There are two options to facilitate the translation of genetic programs to representations which can be used outside of the context of the LGP system. Firstly, a `Program` implementation can directly produce an output through its `toString()` method.

Alternatively, the `ProgramTranslator` module, provides an interface for translating a program instance, that allows for custom output to be encapsulated for use in batch processing or from outside of the system.

The built in `BaseProgram` class has a corresponding `BaseProgramTranslator` which can convert the internal representation to a fully functional program in the C programming language. This translator provides two modes:

1. Main function included with model
2. Standalone function for model.

The first mode (when `includeMainFunction == true`), will translate to a C program that includes the model as a C function, alongside a main function that parses inputs for the model from the command-line. This is intended to be used when the model is executed from the command-line and is given inputs as arguments.

Secondly — if `includeMainFunction` is set to `false` — the translator will exclude the main function, instead including the model as a standalone function. This mode is better suited to situations where the model is integrated into an existing code base.

The output from `BaseProgramTranslator` should compile under most systems using the following command (assuming the output is saved in `model.c`):

```
gcc -o model model.c
```

API

See nz.co.jedsimson.lgp.core.program for information on the `Program`, `ProgramGenerator` and `ProgramTranslator` APIs.

API details for the built-in implementations are found in nz.co.jedsimson.lgp.lib.

2.5.4 Evolutionary Operators

Overview

Evolutionary Operators are a primary component of the system, as they provide the means for the evolutionary process to guide its search through the search space.

There are three main search operators implemented as part of the system — all of which are implementations of the module interface. What this means is that they particular implementation of search operator has a plug-in like interface, allowing different operators to be used where appropriate.

Implementing a custom operator is as simple as implementing the correct interface: if the operator is an extension of one of the built-in operators (as detailed below), the appropriate abstract class can be implemented. Otherwise, a custom module can be defined and used in an appropriate way as defined within the evolutionary algorithm.

Selection Operator

The `SelectionOperator` abstract class provides the basic outline of a selection operator as used by the system. This interface is used by the built-in evolutionary algorithms, and thus a concrete implementation of this abstract class will be able to provide custom functionality determined by the particular selection scheme to be used.

By default, the system offers implementations of tournament selection and binary tournament selection which will be suitable for a large number of cases.

Recombination Operator

`RecombinationOperator` abstracts away the details of an operator that is used to combine individuals. The interface is used in the same way as other operators, allowing a concrete implementation to dictate the particular behaviour employed by the operator.

By default, the system offers implementation of linear crossover as a `RecombinationOperator` which can be used in a range of circumstances.

Mutation Operator

A `MutationOperator` provides an abstract interface for mutating a given individual from a population. The details of this are up to the implementer, as for the other evolutionary operators — this means custom mutation operators can be implemented and used by the system where required.

LGP splits mutation operators into two categories: micro and macro mutation. Micro mutation performs mutation on the instruction level by changing properties of individual instructions such as register index, operation type, or constant value. In contrast, macro mutation operates at the program level and either adds or deletes entire instructions.

The `MicroMutationOperator` and `MacroMutationOperator` classes provide a implementations of the `MutationOperator` abstract class that can be used to facilitate both types of mutation. These built-in operators perform effective micro and macro mutations as given by Brameier, M. F., & Banzhaf, W. (2007)[1].

API

See nz.co.jedsimson.lgp.core.evolution.operators.

[1] Brameier, M. F., & Banzhaf, W. (2007). Linear Genetic Programming. Springer Science & Business Media. https://doi.org/10.1007/978-0-387-31030-5

2.5.5 Fitness

Fitness evaluation is an important part of the evolutionary search process, and is the most time consuming portion of the algorithm — requiring multiple program evaluations on a set of fitness cases. The resulting fitness evaluations help to guide the evolutionary search towards better performing solutions.

There are three components in the system that are responsible for performing and controlling fitness evaluation: (1) fitness function, (2) fitness context, and (3) fitness evaluator. The following sections will detail each component, as well as the interaction between them.

Fitness Function

A fitness function (`FitnessFunction`) is really just an ordinary function that has two arguments — a list of program outputs and a list of fitness cases — and produces a single real-valued output.

A fitness function is used to measure the error between the expected output values (as determined by the fitness cases) and predicted output values of the model (i.e. an LGP program). An example fitness function, mean-squared error, is given by (2.2), where Y is a vector of n expected outputs and $\hat{Y}$ is a vector of n predicted outputs.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{2.2}$$

The API provides a collection of built-in functions, found in the `FitnessFunctions` object. These can be selected as construction components when building an LGP environment.

Alternatively, one can simply write a function with the signature `(List<TOutput>, List<FitnessCase<TData>>) -> Double` to implement a custom fitness function for the particular problem being solved (where `TOutput` is type of output (e.g. `Outputs.Single<TData>`) and `TData` is the type of data in the registers (e.g. `Double`)). The functionality is encapsulated in the `FitnessFunction` abstract class.

Fitness Context

A fitness context (`FitnessContext`) essentially maps a program to a set of input-output examples in order to produce a fitness value. This is done by executing the program on each of the input vectors and aggregating the output results. The error can then be measured using a `FitnessFunction`.

The reason this is encapsulated in its own class is to allow for modularity. The particular `FitnessContext` implementation is chosen by the user, meaning that custom logic for aggregating and evaluating results can be implemented — e.g. for multiple-output programs or weighted outputs.

By default, the API provides the `SingleOutputFitnessContext` which will simply gather a single program output from program's specified output register. Each output will used to evaluate the programs fitness using the specified `FitnessFunction`.

There is also a `MultipleOutputFitnessContext` which can be used in the cases where a program has multiple outputs. This will require a fitness function that can handle programs with multiple outputs to be used. There are no such functions built into the API, as they will be problem-specific.

Note: The `Output` type across the problem implementation should be consistent with the context and fitness function used:

- Use `Outputs.Single` when the `SingleOutputFitnessContext` and `SingleOutputFitnessFunction` will be used.

- Use `Outputs.Multiple` when the `MultipleOutputFitnessContext` and `MultipleOutputFitnessFunction` will be used.

Fitness Evaluator

The fitness evaluator (`FitnessEvaluator`) is responsible for combining all the other pieces together. The `FitnessEvaluator` simply takes a program and a data set and uses a `FitnessContext` to evaluate the fitness. The `FitnessEvaluator` does not know about the details contained within the `FitnessContext` as that is decided by the user — it simply adheres to the interface of the other components.

API

See nz.co.jedsimson.lgp.core.evolution.fitness.

2.5.6 Extensions

Overview

As somewhat alluded to when describing the environment component of the system, the system allows custom registered components to be registered to allow for custom functionality where necessary. This is done by mapping a module implementation to a particular module type.

What wasn't mentioned is that there is no set enumeration of module types, meaning that you can create your own module type and own custom module and the system will be able to use it once registered.

This has been done to leave the system open for future developments to LGP and GP in general — namely, the creation and integration of novel search operators or evolutionary algorithms to extend the system. Alternatively, a less complex module could be included to perform tasks such as logging of the evolutionary process or real-time aggregation of the results.

Example

For the sake of completeness, we will create a custom module implementation and register it to a custom module type to show how the process works. It should be noted that this module will not be used by the system as none of the built-in components are made to use this custom module, but the principle is what is important.

The custom module we'll be creating is a logger which can be injected into the system, to be made available anywhere the environment can be queried.

To start, we create our own module type to which the module implementation will be registered:

```
// Using an enum class allows for further module types to
// be added later.
enum class CustomModuleType : RegisteredModuleType {
    Logger
}
```

Next, we need to build a module implementation which can be registered to this module type. This could be anything as long as it implements the `Module` interface, but for our purposes we are building a logger:

```
import java.time.LocalDateTime

enum class LoggerLevel {
    Error,
    Warning,
    Info,
    Debug
}

class Logger(
    val name: String,
    val level: LoggerLevel = LoggerLevel.Info
) : Module {

    override val information = ModuleInformation(
        description = "A custom logger that will be injected" +
                      "into the evolutionary process."
    )

    fun log(level: LoggerLevel, message: String) {
        if (level <= this.level) {
            val now = LocalDateTime.now()
            val levelName = level.name.toUpperCase()

            println("[$now] ($name-${levelName}): $message")
        }
    }
}
```

Our logger offers a fairly basic set of functionalities. It can have a name and a level set and will log messages that are suitable for the level set (if the message request level falls below the logger's set level, the message will be printed).

Finally, when building an `Environment`, we can simply register an implementation of this module with our new module type and the system will become aware of that module and how to access it. Because we want the logger to be a singleton instance (i.e. the same logger should be returned each time it is requested), we need to make sure the builder returns a specific instance:

```
...

// Our environment - initialisation details are omitted.
val env = Environment<Double, Outputs.Single>(
    configLoader,
    constantLoader,
    operationLoader,
    defaultValueProvider,
    fitnessFunction = { mse }
)

val logger = Logger(
    name = "EvolutionaryLogger",
    level = LoggerLevel.Info
)

// Build up a container for any modules that need to be registered.
// This is where we'll register our custom logger module as specified above.
val container = ModuleContainer(
    modules = mapOf(
```

(continues on next page)

(continued from previous page)

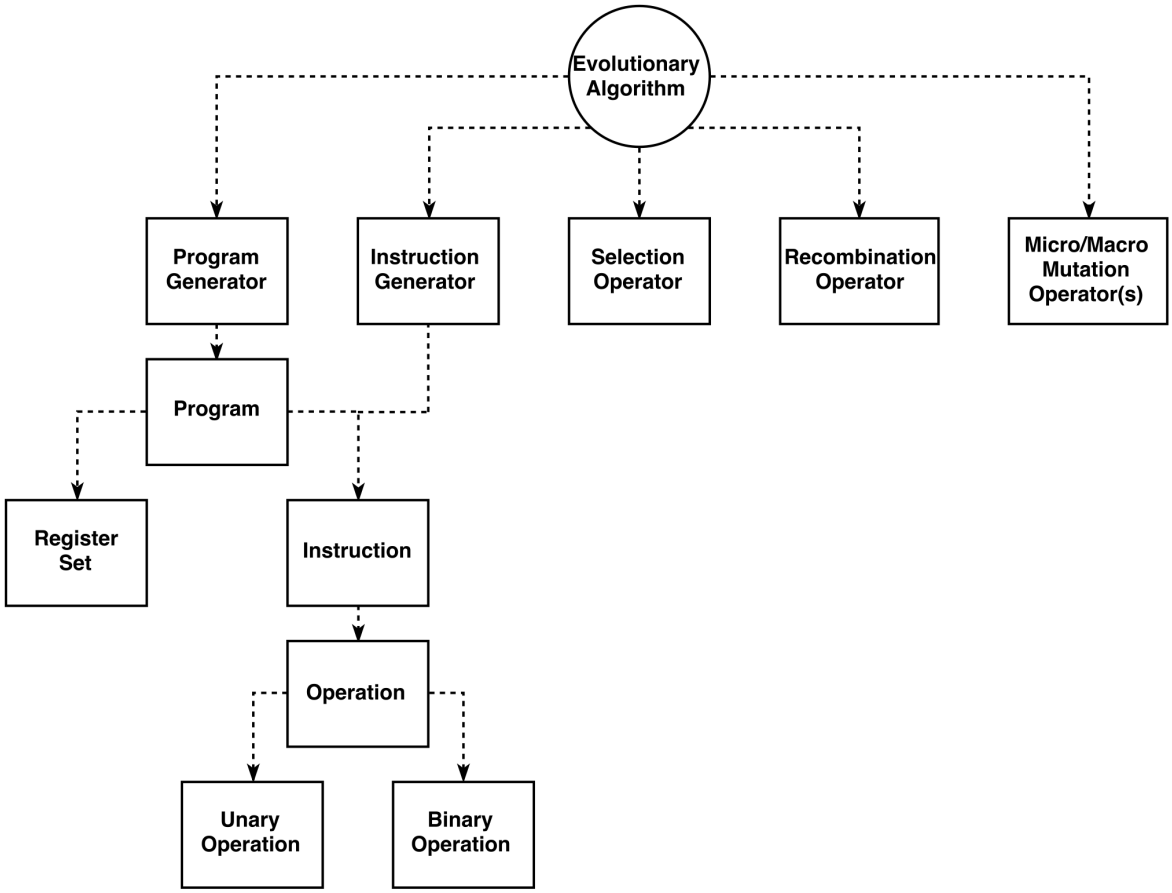
```
        CoreModuleType.InstructionGenerator to
        { BaseInstructionGenerator(env) },

        CoreModuleType.ProgramGenerator to
        { BaseProgramGenerator(env) },

        // Our custom logger instance.
        CustomModuleType.Logger to
        { logger }
    )
)

// Inform the environment of these modules.
env.registerModules(container)
```

Now the logger instance can be accessed from anywhere the environment is visible within the system. None of the built-in modules are set up to use this custom module, but they could be adapted to use this functionality — meaning the system is extremely malleable to different extensions.

API

There are a few relevant APIs for creating custom modules. Firstly, the nz.co.jedsimson.lgp.core.modules package provides the definition of the `Module` interface which must be implemented in order to create custom modules.

Furthermore, to create a custom module type to allow for a custom module to be registered, the nz.co.jedsimson.lgp.core.environment package defines the `RegisteredModuleType` interface which must be implemented to create a new module type that is able to be registered within the environment.



2.6 Usage

The system is built using Kotlin and the easiest way to use it is through the Kotlin API. Instructions for installation and usage of the Kotlin compiler, `kotlinc`, can be found for the Command Line or IntelliJ IDEA.

2.6.1 Installation

> **Warning:** The LGP framework requires JDK 8 (Java 1.8).

A JAR containing the core API can be downloaded from the GitHub releases page. The command below can be used to download the JAR from a terminal so that development against the API can begin:

```
curl -L https://github.com/JedS6391/LGP/releases/download/4.2/LGP-core-4.2-2019-02-09.
↪jar > LGP-core.jar
```

We also need the latest copy of the base LGP implementations, provided in the LGP-lib repository. The command below can be used to download the JAR from a terminal:

```
curl -L https://github.com/JedS6391/LGP-lib/releases/download/1.1/LGP-lib-1.1-2019-02-
↪09.jar > LGP-lib.jar
```

Note: These command will download the most up-to-date releases as of publishing this guide. For other releases, please see the GitHub releases pages (LGP-core and LGP-lib).

2.6.2 With Kotlin

Here, we'll focus on how to use the system through Kotlin (particularly from the command line) but documentation is provided for using the API through Java.

Assuming that `kotlinc` is installed and available at the command line, the first step is to download the core API JAR file as described in the *Installation* section.

Next, create a blank Kotlin file that will contain the problem definition — typically this would have a filename matching that of the problem:

```
touch MyProblem.kt
```

We're not going to fully define the problem as that would be a needlessly extensive exercise, so we'll simply show how to import classes from the API and build against the imported classes.

In `MyProblem.kt`, enter the following content:

```
import nz.co.jedsimson.lgp.core.environment.config.Configuration
import nz.co.jedsimson.lgp.core.evolution.Description
import nz.co.jedsimson.lgp.lib.base.BaseProblem
import nz.co.jedsimson.lgp.lib.base.BaseProblemParameters

fun main(args: Array<String>) {
    val parameters = BaseProblemParameters(
        name = "My Problem",
        description = Description(
            "A simple example problem definition"
        ),
        // A problem will generally need custom configuration
        config = Configuration()
    )

    val problem = BaseProblem(parameters)

    println(problem.name)
    println(problem.description)
}
```

Here, we use the `BaseProblem` implementation to use a default set of parameters that we can quickly test against using a data set (which is omitted here).

To compile, we use `kotlinc`:

```
kotlinc -cp LGP-core.jar:LGP-lib.jar -no-jdk -no-stdlib MyProblem.kt
```

This will generate a class file in the directory called `MyProblemKt.class`. To interpret the class file using the Kotlin interpreter is simple:

```
kotlin -cp LGP-core.jar:LGP-lib.jar:. MyProblemKt
```

You should see the following output:

```
My Problem
Description(description=A simple example problem definition)
```

Alternatively, the same result can be achieved by setting the destination to another JAR file and executing using the Java interpreter:

```
# Compile to a JAR using kotlinc
kotlinc -cp LGP-core.jar:LGP-lib.jar -no-jdk -no-stdlib -d MyProblem.jar MyProblem.kt

# Use the Kotlin interpreter to execute the main function
kotlin -cp LGP-core.jar:LGP-lib.jar:MyProblem.jar:. MyProblemKt
```

2.6.3 With Java

The same functionality as above from the perspective of Java is not quite as elegant, but still fully possible. Because Java doesn't offer optional parameters, it makes the Kotlin API slightly harder to use as we have to provide values for any optional parameters.

To start, a new Java file should be created with the name of the main class as per the usual Java specification:

```
touch MyProblem.java
```

Next, the file can be filled with the following:

```
import kotlin.jvm.functions.Function2;
import nz.co.jedsimson.lgp.core.environment.config.Configuration;
import nz.co.jedsimson.lgp.core.evolution.Description;
import nz.co.jedsimson.lgp.core.evolution.fitness.FitnessCase;
import nz.co.jedsimson.lgp.core.evolution.fitness.FitnessFunctions;
import nz.co.jedsimson.lgp.core.evolution.fitness.FitnessFunction;
import nz.co.jedsimson.lgp.core.program.Outputs;
import nz.co.jedsimson.lgp.lib.base.BaseProblem;
import nz.co.jedsimson.lgp.lib.base.BaseProblemParameters;

import java.util.Arrays;
import java.util.List;

public class MyProblem {

    static String name = "My Problem";
    static Description description = new Description(
        "A simple example problem definition"
    );
    static String configFilename = null;
    static Configuration config = new Configuration();
    static Double[] constants = { -1.0, 0.0, 1.0 };
    static String[] operationClassNames = {
        "lgp.lib.operations.Addition",
        "lgp.lib.operations.Subtraction",
        "lgp.lib.operations.Multiplication",
        "lgp.lib.operations.Division"
    };
```

(continues on next page)

(continued from previous page)

```
    static double defaultRegisterValue = 1.0;
    static FitnessFunction<Double, Outputs.Single<Double>> mse = FitnessFunctions.
↪getMSE();
    static int tournamentSize = 20;
    static int maximumSegmentLength = 6;
    static int maximumCrossoverDistance = 5;
    static int maximumSegmentLengthDifference = 3;
    static double macroMutationInsertionRate = 0.67;
    static double macroMutationDeletionRate = 0.33;
    static double microRegisterMutationRate = 0.4;
    static double microOperationMutationRate = 0.4;
    static Long randomStateSeed = null;
    static int runs = 10;

    public static void main(String[] args) {
        BaseProblemParameters parameters = new BaseProblemParameters(
            name,
            description,
            configFilename,
            config,
            Arrays.asList(constants),
            Arrays.asList(operationClassNames),
            defaultRegisterValue,
            mse,
            tournamentSize,
            maximumSegmentLength,
            maximumCrossoverDistance,
            maximumSegmentLengthDifference,
            macroMutationInsertionRate,
            macroMutationDeletionRate,
            microRegisterMutationRate,
            microOperationMutationRate,
            randomStateSeed,
            runs
        );

        BaseProblem problem = new BaseProblem(parameters);

        System.out.println(problem.getName());
        System.out.println(problem.getDescription());
    }
}
```

This set-up is the same as for the Kotlin API usage example, but is slightly more verbose due to Java's omission of optional parameters as mentioned previously.

To compile and run however, is still fairly straight-forward:

```
# First, compile the code against the LGP API
javac -cp LGP-core.jar:LGP-lib.jar MyProblem.java

# Secondly, run the resulting class on the JVM
java -cp LGP-core.jar:LGP-lib.jar:. MyProblem
```

If everything went as expected, then the same output should be produced as for the Kotlin example:

```
My Problem
Description(description=A simple example problem definition)
```

2.7 Java Interoperability

For the most part, using the system's API from the context of a Java program will be very similar to that of a Kotlin program. However, because the API is written in and designed for Kotlin, there are certain situations where the Java API is somewhat of a second-class citizen.

2.7.1 Documentation (Javadoc)

Generally, one can get by referencing the dokka API documentation when using Java. For the cases where a more Java-centric view is required, the system has Javadoc available too.

2.7.2 Peculiarities

Because of the API being design first and foremost for interaction within Kotlin, there are a few pain points that have been aggregated and documented here as a reference:

Unsafe Implementations Due to Java's deficiency when it comes to reified generics, the `Environment::registeredModule` and `ModuleContainer::instance` functions require workarounds.

The workaround implementations (`Environment::registeredModuleUnsafe` and `ModuleContainer::instanceUnsafe`) cannot guarantee type safety in the same way their Kotlin counterparts can, and thus should be used with caution. The documentation for both methods provides further detail on the exact problems that arise.

Lambda Functions Wherever the Kotlin API requires a lambda function, the Java equivalent can make use of Java lambda functions.

The API however will specify Kotlin types (e.g. `kotlin.jvm.functions.Function1`) as opposed to the Java equivalent. This should not impact usage in any way.

Optional Parameters Any API functions which expose optional parameters in Kotlin will require that a parameter is given explicitly in Java, due to the lack of optional parameters in Java.

The best thing to do is to simply find out what the default parameter used is and make use of that value.