Open-Source Linear Genetic Programming

Author
Jed Simson

Supervisor
Michael Mayo

Faculty of Computing and Mathematical Sciences
University of Waikato, Waikato, New Zealand

October 10, 2017

Abstract

Linear Genetic Programming (LGP) is a paradigm of genetic programming that employs a representation of linearly sequenced instructions in automatically generated programs. A linear approach lends itself to programs which have two unique attributes: a graph-based functional structure and the existence of non-effective instructions.

Motivated by the lack of existing implementations, an LGP system is developed and released into the open-source community. The system has a modern design with emphasis on correctness, ease of use, and extensibility. This work discusses LGP concepts and the implementation of a modern LGP system.

The system built is evaluated on a set of symbolic regression benchmark problems to ensure performance and correctness of the implementation. Three rounds of experiments demonstrate (1) the effects of different configurations of the system, (2) a comparison of different evolutionary algorithm performance within the system, and (3) the equivalence to a traditional tree-based GP approach and linear regression model.

Acknowledgements

Firstly, I would like to express my gratitude to Dr. Michael Mayo for his supervision throughout this work. His expertise, guidance, and willingness to answer all of my questions was deeply appreciated.

Secondly, for their continual support and encouragement during my studies — my parents and sister; my partner, Abby; and my good friend, Andrew.

Contents

List of Tables

List of Figures

1 Introduction

The desire for a system which can automatically craft computer programs has been known in the machine learning community for some time. Friedberg (1958) experimented with a system that solved problems by randomly changing instructions in a program and favouring those changes which most frequently achieved a positive result. Further work by Fogel, Owens, & Walsh (1966) applied simulated evolution to finite-state machines in a technique titled Evolutionary Programming.

Genetic Programming (GP) borrows concepts from evolutionary biology in order to optimize computer programs towards a particular goal. Individuals within a population are exposed to the processes of Darwinian natural selection, sexual recombination (genetic crossover), and mutation which lead to a change in characteristics over successive generations by exploiting differential fitness advantages in order to survive (Koza, 1994). GP as traditionally described utilizes a tree-based representation in which programs correspond to expressions in a functional programming language (Brameier & Banzhaf, 2007).

GP has been used successfully to solve machine learning problems across a wide range of domains including but not limited to robotics (Luke, 1998), image processing (Poli & Cagnoni, 1997), traditional software engineering (Langdon & Harman, 2015), and combination optimization problems (Jacobsen-Grocott, Mei, Chen, & Zhang, 2017).

Linear Genetic Programming (referred to as LGP from here) is a variant of GP where individuals in the population adopt an imperative program structure. The term "linear" refers specifically to the structure of the program and does not limit the type of problems that LGP can be used to solve (Brameier & Banzhaf, 2007).

There are two primary features which differentiate LGP from a traditional tree-based approach (Brameier & Banzhaf, 2007): first, LGP programs exhibit a unique graph-based data flow due to the way the contents of a particular register may be used multiple times during a programs execution. This leads to program graphs with higher variability thus enabling program solutions which are more compact in comparison to tree-based solutions to evolve.

Secondly, special non-effective code coexists with a program's effective code as a result of the imperative structure. Non-effective code refers to instructions within an LGP program which do not impact the program output. These non-effective instructions guard the effective instructions from disruption caused by the genetic operator application and allows variations to remain neutral in terms of a fitness change. There is evidence that such neutral variations are beneficial; Yu & Miller (2001) and Galvan-Lopez & Rodriguez-Vazquez (2006) show that there is a positive relationship between neutrality and evolvability and that neutrality tends to provide better evolutionary process performance, respectively.

Brameier & Banzhaf (2007) outline a method that can be used to efficiently and robustly detect and remove non-effective instructions.

Listing 1 illustrates an LGP program that operates on a set of 8 registers. Instructions marked as comments with the `//` prefix are not effective and have no effect on the programs output which is derived from the contents of register `r[0]` when the program terminates. These commented instructions are shown for the purposes of this example, however it should be noted that they are not part of the program representation. Furthermore, the graph-based data flow is evident in the way registers are reused through a program (e.g. registers `r[0]` and `r[4]`).

Effective Instruction An instruction in an LGP program is effective at its position if and only if it influences the output(s) of the program for one or more of the possible inputs.

Non-effective Instruction An instruction in an LGP program which has no influence on the output(s) for any of the possible inputs.

Listing 1 An example LGP program

```
1  void gp(double r[8]) {
2      ...
3      r[0] = r[5] + 71;
4      // r[7] = r[0] - 59;
5      if (r[1] > 0)
6          if (r[5] > 2)
7              r[4] = r[2] * r[1];
8      // r[2] = r[5] + r[4];
9      r[6] = r[4] * 13;
10     r[1] = r[3] / 2;
11     // if (r[0] > r[1])
12     //     r[3] = r[5] * r[5];
13     r[7] = r[6] - 2;
14     // r[5] = r[7] + 15;
15     if (r[1] <= r[6])
16         r[0] = sin(r[7]);
17 }
```

As an example, the first instance of `r[7]` being used as a destination register (line 4) is marked as non-effective. Let the set R_{eff} contain registers that are effective at the current program position. Starting at the last program instruction, it can be determined that `r[7]` is an effective register as it is used as the operand for the instruction, thus it is added to R_{eff}. The dependency analysis then involves moving backwards through the program to find the first instruction with $r_{dest} \in R_{eff}$; in this case the instruction on line 13 uses `r[7]` as its destination register and therefore `r[7]` is removed from R_{eff}. As the analysis through the program continues, there are no

more usages of `r[7]` as an operand register for an effective instruction. As a result, when the instruction on line 4 is encountered, `r[7]` is not a member of R_{eff} and hence the instruction is non-effective.

Despite the benefits of LGP there is no robust open-source system for utilizing its unique features in order to solve particular tasks. This is a stark contrast to the wide landscape of available tools for utilizing a more traditional tree-based GP representation. An open-source implementation which is flexible would fit this niche and provide a foundation to base further work using LGP upon, preventing others having to implement their own LGP system.

1.1 Motivation

The Open-Source Software (OSS) model describes a process that facilitates the development of software by contributors from varying backgrounds, whose source code can be modified and redistributed freely (Basili & Lonchamp, 2005).

With the rising popularity of open-source development (MacCormack, Rusnak, & Baldwin, 2006), it is somewhat surprising that there is an apparent lack of a robust, open-source solution for utilizing LGP – especially when considering the numerous open-source, tree-based GP systems that are available.

Sonnenburg et al. (2007) state the importance of OSS within the context of machine learning with regards to better reproducibility, innovative applications and the faster integration of these techniques and methods into other disciplines and industry.

The goal of this work is to design, implement and benchmark a completely open-source LGP system. The system attempts to satisfy the following attributes:

- Cross-platform support
- A flexible and adaptable architecture
- An API in a modern programming language
- Efficient implementations of the core LGP algorithms

The software produced has been released into the open-source community alongside API and usage documentation using the GitHub platform. Any development in the future will take place through the GitHub repository, allowing for others to contribute as they wish. In addition, the implementation is verified on a collection of standard benchmark problems to ensure its performance and correctness. It is intended that the software provides a useful contribution to those who wish to use LGP over the traditional tree-based approach.

1.2 Organization

The rest of the paper is organized as follows: Section 2 provides background information on LGP to create a foundation for the further sections, related work on open-source

GP systems, as well as a review of usages of LGP. Section 3 discusses how the problem is approached, including design decisions and their rationale as well as implementation details and how these relate to the overall goal. The benchmarks and their configurations are explained in Section 4 with the results and their corresponding discussion presented in Section 5. Section 6 concludes the paper and gives directions for any further work which could be done.

2 Background

This section concerns itself with the details of LGP in more depth, provides an overview of other open-source GP systems and reviews recent LGP applications and developments.

2.1 Symbolic Regression

Symbolic regression is the name given to the variant of regression analysis that involves automatically building mathematical models that describe relationships on numeric multivariate data sets (Vladislavleva, Smits, & Hertog, 2009). This is achieved by exploring the search space of such expressions with the goal of gaining insight into input variables that are related to changes in the output variables.

The initial expressions of a model are generated randomly from a user-defined specification of mathematical operators and new models are formed by combining or altering previous models (e.g. through genetic programming). The search continues until the perfect model is found or the allotted computation time is exceeded. Symbolic regression uses a fitness function to drive the evolution of models, typically optimising for measures of accuracy or simplicity.

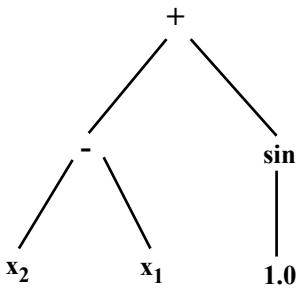
2.2 Overview of Linear Genetic Programming

Within the context of techniques for the evolution of computer programs there are two main approaches as acknowledged by Banzhaf (1993): The first operates by allowing programs to compete for computing resources such as CPU-time or memory and has the potential to create interesting phenomena, such as parasitism and symbiosis. Alternatively, the technique of *Genetic Programming* (sometimes referred to as Evolutionary Programming) concerns itself with evolving programs in accordance to behaviour described by a user's particular requirements.

Since the introduction of genetic algorithms as a tool for solving optimization problems, there has been a goal of evolving algorithms themselves using similar techniques. Cramer (1985) begun with the investigation into using linear bit sequences as a programming language that can be manipulated by genetic algorithms in order to generate simple functions from basic computational primitives. Banzhaf (1993) furthered this linear approach by using strings of linear op-codes that can be interpreted

as programs and evolved using traditional genetic algorithm techniques to evolve towards a certain goal.

Linear Genetic Programming (Brameier, 2004) is a variant of genetic programming in which sequences of instructions from an imperative programming language (e.g. C or machine) code are evolved. The linear structure does not limit the kind of problems that LGP can solve as it can find solutions to highly non-linear problems. For example the work of Guven (2009) utilized LGP to predict river daily flow rate data with a high level of accuracy.




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

Figure 1: Comparison of a tree-based program representation (left) with a linear program representation (right). The LGP program is represented in the C programming language as a function which takes a single array argument. The array supplied as an argument to the LGP program is given by $r = \{x_1, x_2, 1.0, 1.0, 0.0, 1.0\}$ such that `r[i]` in the program code corresponds to the ith value of r (e.g. `r[0]` $= r_0 = x_1$). Both programs compute the function $f(x_1, x_2) = (x_2 - x_1) + sin(1.0)$.

The more general phenomena associated with the linear GP approach are explored by Nordin & Banzhaf (1995) and Nordin, Francone, & Banzhaf (1996). Characteristics that only arise from a linear approach are detailed by Brameier & Banzhaf (2007) as well as implementation details regarding variation operators that take advantage of the linear representation in order to produce better, more compact solutions.

The subsequent sections detail the core concepts of LGP as outlined by Brameier & Banzhaf (2007) in order to provide a sufficient foundation for understanding the rationale presented when discussing the design and implementation of the system.

2.2.1 Representation

LGP operates with imperative programs that consist of a variable-length sequence of instructions which perform operations on the contents of a set of registers. These operations manipulate the contents of the registers to facilitate calculations and compute a result.

This approach is closely related to the underlying machine language, unlike tree-

based GP. LGP programs reflect the von Neumann architecture that describes a system of registers and basic instructions that effect those registers (Brameier & Banzhaf, 2007). One important implication of this particular representation is that programs of such a register machine describe a sequence of instructions whose order has to be observed during execution.

Each LGP program is provided a set of registers in order to facilitate its calculations and store input values. The register set is in essence made up of three distinct sections of registers.

Input Registers These hold a program's inputs before execution. These dictate the particular behaviour that the program is targeting. Typically these registers are referred to as x_1, x_2, etc.

Calculation Registers A variable number of additional registers used to aid in calculations performed as part of a program. Generally, these are initialized with a default value before a program is executed.

Constant Registers A pre-defined number of registers which are loaded with constant values and are write-protected.

One or more of the input or calculation registers are defined to be the *output register(s)*, allowing for the possibility of multiple program outputs (unlike a tree-based approach).

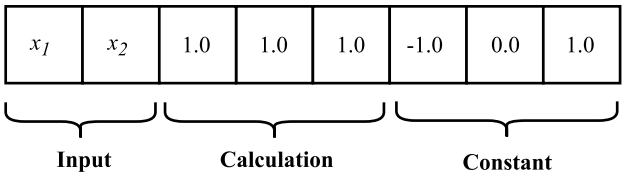


Figure 2: An illustration of a register set comprised of two input registers, three calculation registers, and three constant registers. The calculation registers are initialised with a default value of 1.0 while the constant registers provide the values $[-1.0, 0.0, 1.0]$ to aid in computation. In the context of a program in the C programming language, the contents would be accessed through array syntax (e.g. `r[0]` $= x_1$ and `r[5]` $= -1.0$)

Instructions in LGP are comprised of a single operation (i.e. function) and a set of operand registers. In all cases, the result of an instruction will be stored in one of the operand registers denoted as the *destination register*. The operation of a particular instruction will vary but generally, operations are performed on one or two operand registers called *source registers*. For example, the instruction $r_i := sin(r_j)$ has one source register (r_j) and one destination register (r_i) whereas the instruction $r_i := r_j + r_k$ has two source registers (r_j and r_k) and one destination register (r_i). It

is also worth mentioning that an instruction could have its destination register as one of its source registers, for example $r_i := r_i + r_j$.

Brameier & Banzhaf (2007) note that in general, instructions which perform multiple operations (e.g. $r_i := r_i + r_j + r_k$) do not necessarily increase the expressiveness and variability of programs, and such a higher dimensional program structure introduces an additional level of complexity with respect to the genetic operators.

Most commonly, LGP programs are represented as a sequence of instructions from the C programming language. This allows for solutions to be applied directly to a problem domain without the use of an interpreter for the genetic program solutions as they can be directly compiled (Brameier & Banzhaf, 2007) - clearly an advantage of LGP.

The instruction set given to LGP defines the particular programming language for programs evolved. Traditionally an LGP system will provide a selection of instructions such as basic arithmetic operations, exponential functions, trigonometric functions, boolean operations and conditional branches. It is possible however to use instructions tailored to the problem domain. For example, an LGP program could consist of commands for directing a robot within a certain environment (Brameier & Banzhaf, 2007). In general however, LGP employs instruction which are pure in the mathematical sense as they do not introduce side-effects.

There are two invariants that must be satisfied by LGP in order to guarantee that only valid programs are created, known as *syntactic* and *semantic* correctness. The former involves ensuring that the effects of genetic operators — whether it be mutation or recombination — maintain the correctness of the programs operated on. LGP achieves this by treating instructions as atomic units such that crossover cannot occur in the middle of an instruction, as well as restricting the parts of an instruction that mutation operators can effect. For example, the + operation in the instruction $r_i := r_j + r_k$ cannot be replaced with another register (e.g. $r_i := r_j r_i r_k$), as this would render the instruction invalid thus violating the program's syntactic correctness.

The latter — semantic correctness — demands that operations which may have undefined behaviour for certain inputs don't jeopardize a program's execution. This is typically attained by defining a high constant return value for undefined inputs in an attempt to penalize programs which make use of instructions with undefined inputs. Table 1 provides examples of instructions that are protected from undefined inputs in order to remain semantically valid in LGP.

Table 1: Examples of instructions that may be used by LGP with their corresponding semantically valid definition. $r_{i,j,k}$ refer to the registers of an LGP program and $C_{undefined}$ is generally a high constant value (e.g. 10^6).

Instruction	**Semantically Valid Definition**
$r_i := r_j \div r_k$	$if\ (r_k \neq 0) \quad r_i := r_j \div r_k \quad else \quad r_i := r_j + C_{undefined}$
$r_i := r_j^{r_k}$	$if\ (r_k \leq 10) \quad r_i := r_j^{r_k} \quad else \quad r_i := r_j + r_k + C_{undefined}$

Instruction	Semantically Valid Definition
$r_i := \sqrt{r_j}$	$r_i = \sqrt{\lvert r_j \rvert}$

2.2.2 Execution

The cost of computation when using LGP is dominated by the evaluation of programs, as is true with other GP systems. The reason for the high cost of evaluation is due to the fact that each program requires multiple executions for each specific case that the program is being tested against in order to evaluate its fitness.

Generally, execution relies on transforming a program's internal representation into some interpreted form that can be executed according to the semantics of that programming language. A tree-based program is interpreted through traversing the tree structure in a particular order, applying operators to values where appropriate.

LGP provides a method resulting from the structure of the linear programs that can be applied to accelerate the execution of programs evolved. LGP programs have the unique feature of containing instructions which are deemed *non-effective* (i.e. they have no influence on the programs output). Such instructions can be found efficiently and removed before a program is executed, decreasing the number of instructions that are executed for each program. This effect is particularly advantageous when there is a large number of cases in the training data used to tune the program's target behaviour.

2.2.3 Evolution

The algorithm detailed below forms the core evolutionary algorithm upon which LGP is built (Brameier & Banzhaf, 2007). In this particular steady-state EA, offspring replace existing individuals in the same population. This EA also utilises three separate sets of fitness cases: training, validation, and test cases. All individuals are trained on the training set to determine their fitness for the selection process of the algorithm. This is supplemented by a validation step that is performed on the best-fit program, in order to check the generalization ability of the solution. When the maximum number of generations are reached, the program with the minimum validation error is tested on the test data set.

1. *Initialize* a population of random programs.
2. Randomly *select* $2 \times n$ 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 with 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 with 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.

There are essentially four primary phases involved in such an EA - *Initialisation*, *Selection*, *Variation*, and *Evaluation*. These phases are described by Brameier & Banzhaf (2007) within the context of LGP as follows.

Initialisation

An evolutionary algorithm begins by building an initial population of genetic programs, normally achieved by randomly creating a number of valid programs. LGP defines an upper and lower bound on the initial program length such that the length of programs is randomly chosen from within these bounds with uniform probability.

It is important to find a balance between initial programs which are too short or too long. Programs that have a small initial program length lack sufficient genetic material to produce a diverse population, with this becoming particularly noticeable within small populations or when crossover is the dominant variation operator. On the contrary, initial programs that are too long may be more inflexible during evolutionary manipulations (Brameier & Banzhaf, 2007), and as a result it can be difficult for the EA to follow a search path from a complex region (e.g. a long program) to another complex region (with better programs).

Selection

The algorithm listed above applies tournament selection in order to randomly determine the individuals which will be subjected to the effects of the variation operators before being reintroduced into the population. Listing 2 illustrates the basic tournament selection algorithm as pseudo code for the case when fitness is being minimised. The parameter k controls the size of the tournament and thus the selection pressure that is imposed on the population individuals. A lower tournament size value corresponds to lower selection pressure and thus weak individuals have a greater chance of being selected. It is possible for a chosen individual to be removed from the population when selected, which prevents the same individual being selected multiple times for the next generation. LGP performs two tournaments in parallel in order to produce two parent individuals for crossover.

Listing 2 Tournament selection algorithm

```
def tournament_selection(population, k):
    N = length(population)
    winner = population[random(0, N - 1)]

    for i = 1 to k - 1:
        contender = population[random(0, N - 1)]

        if fitness(contender) < fitness(winner):
            winner = contender

    return winner
```

Variation

Variation operators are search operators that generate new programs from existing ones within a population. Two factors influence the variation of a population: firstly, reproduction between individuals in a population shares genetic material between solutions, with a bias towards genetic material that produces a gain in fitness. Secondly, mutation involves altering a single individual with the goal of diversifying the genetic material of the population.

Reproduction in LGP is achieved by combining two genetic programs using a linear crossover method as illustrated by Figure 3. This requires exchanging a segment of random position and length between two individuals. The crossover operator is considered to be highly disruptive in terms of LGP programs, as the removal of a subsequence of instructions could drastically alter the program's function.

LGP further categorizes the variation operators into two categories (aside from recombination operators): *macro* and *micro* operations respectively. Macro operations are performed at the program level and are one of the primary means for varying the length of genetic programs (alongside crossover). An example of such a macro operator is the macro-mutation operator (Brameier & Banzhaf, 2007) which either adds or removes instructions to a program based on a probabilistic model. The macro-mutation operator is less destructive than an operator which exchanges chunks of instructions, such as crossover, but can still be harmful to program function.

Micro operations are conducted at the instruction level and alter the particular parameters that effect an instruction, including its operation, register(s), or constants. An example of a micro operator is micro-mutation, which will randomly change either an instructions destination or operand register(s), operation, or add a small amount of noise to a constant register value. Micro-mutations are the least destructive mutation and encourage small step sizes in terms of program function change.

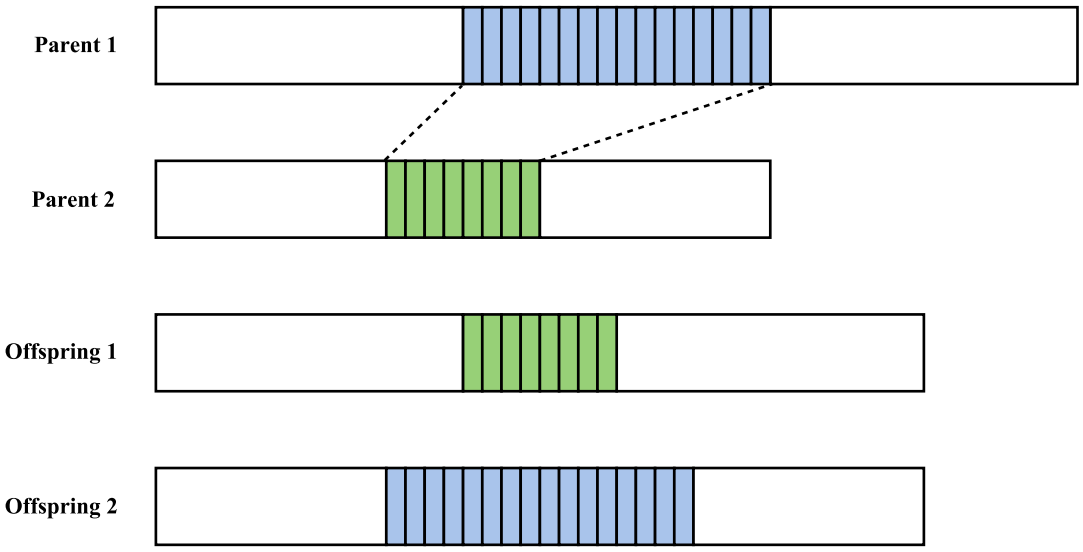


Figure 3: The method of crossover in LGP exchanges variable-length continuous sequences of instructions between parents.

Within these categories, there is the possibility for macro/micro mutations which only effect those instructions which have been marked as effective or mutations which effect any instruction.

```
void gp(double r[6]) {

    r[2] = r[1] - r[0];
    r[3] = sin(r[6]);
    r[2] = r[2] + r[3];

}
```

```
void gp(double r[6]) {

    r[2] = r[1] - r[0];
    r[3] = sin(r[6]);
    r[2] = r[2] + r[3];
    r[2] = r[2] * r[0];
}
```

```
void gp(double r[6]) {

    r[2] = r[1] - r[0];
    // r[3] = sin(r[6]);
    r[2] = r[2] + r[0];
}
```

Figure 4: The left program is an unmodified LGP individual that represents the function $f(x_1, x_2) = (x_2 - x_1) + sin(1.0)$. The middle program illustrates the effects of the macro-mutation operator, as an instruction has been added (shown by the green highlight). This causes the function of the program to change to $f(x_1, x_2) = ((x_2 - x_1) + sin(1.0)) \times x_1$. The right program, demonstrates the application of the micro-mutation operator which has changed one of the operand registers in the last instruction, changing the function to $f(x_1, x_2) = (x_2 - x_1) + x_1$. This has the effect of rendering the previous instruction non-effective as the contents of `r[3]` are no longer used. The values stored in the registers remain the same as in Figure 1 ($r = \{x_1, x_2, 1.0, 1.0, 0.0, 1.0\}$).

Evaluation

Evaluation is the most time consuming portion of an evolutionary algorithm as it involves executing each genetic program multiple times for each case provided as training data. This system focuses on symbolic regression problems (see Section 2.1) meaning that the evaluation process uses typical symbolic regression metrics. The fitness of a program is measured by an error function over a set of fitness cases, which define the problem that is desired to be solved or approximated by the programs.

The popular mean-squared error (MSE) function is given by Equation 1 and is an example of a fitness function commonly used by LGP. In this case, gp refers to the genetic program being measured, $(\vec{i_k}, o_k)$ is a set of n input-output fitness cases, and $gp(\vec{i_k})$ is the predicted output of gp on the inputs $\vec{i_k}$.

$$MSE(gp) = \frac{1}{n} \sum_{k=1}^{n} (gp(\vec{i_k}) - o_k)^2 \tag{1}$$

2.2.4 Characteristics

LGP originates somewhat from the desire to build a genetic programming system in which the genetic programs do not have to undergo an expensive interpretation phase, in an attempt to lessen the computational cost expended when evaluating the fitness of programs. However, as alluded to previously, LGP has other characteristics that arise as an artefact of the linear representation — these are explored here.

Introns are subsequences that exist within strings of DNA that hold information not expressed in the phenotype of an organism. As mentioned earlier, genetic programs consist of code that is either crucial in manipulating the program's output or is redundant and has no effect. Like their natural counterpart, these program introns may reduce the destructive influence of variations on the effective instructions of a program (Brameier & Banzhaf, 2007). LGP distinguishes between instructions which are effective and those are that are non-effective as follows.

Brameier & Banzhaf (2007) further categorize introns into *structural* and *semantic* introns. *Structural* introns are those described by non-effective instructions, where the code has no influence on the output of the program. In essence, these instructions are not connected to the output node of the graph that an LGP program represents. With a traditional tree-based GP approach, structural introns do not exist as instructions are always connected to the root of the tree. *Semantic* introns are instructions which do operate on registers affecting the output (so-called *effective registers*), but the operation which they perform is considered a *no-op* - in other words, the registers the instruction operates on remain the same before and after the instruction has been executed. For example, if r_0 is an effective register then the instruction $r_0 := r_0 \times 1$ is a *semantic* intron as it does not alter the value stored in r_0.

While Brameier & Banzhaf (2007) propose an algorithm that can efficiently and robustly detect structural introns, a similar algorithm for finding semantic introns has

high cost and relies on a probabilistic model. This algorithm requires a computation time of $O(m \cdot n^2)$ where m refers to the number of fitness cases and n is the length of the program, which is deemed too inefficient for removing introns at runtime and thus is not considered in this system.

A secondary characteristic of the linear representation as hinted earlier is that on the functional level, a LGP program defines a directed acyclic graph giving programs a graph-based data flow. These graphs can have a higher amount of variability than the tree structures used by a tree-based GP approach and the evolution of sub-graphs leads to a number of parallel calculation paths that is determined by the number of registers made available to an LGP program; a higher number of registers decreases the likeliness of a write conflict between sub-graphs allowing for a further degree of independent program sub-graphs (i.e. modularity; Brameier & Banzhaf (2007)).

2.3 Open-Source Genetic Programming

In the context of Genetic Programming, there is an array of different systems and implementations for applying GP to a problem. While the majority of these are built upon tree-based GP, there are a few offerings using techniques similar to LGP, and one LGP implementation. This section provides an overview of the different GP offerings and how they relate to the system built.

ECJ (Luke, 2010) is a system for evolutionary computation written in Java which focuses on flexibility and efficiency. ECJ is made available on GitHub[1] along with its source code, making it an entirely open-source system. ECJ focuses on tree-based GP, providing a number of different tree representation as well as a plethora of other features.

`gplearn`[2] offers an implementation of GP in Python with an API inspired by (and compatible with) the popular scikit-learn[3] machine learning framework. `gplearn` has the limitation of being restricted to solving symbolic regression problems, and like ECJ utilises a tree-based representation for evolved programs.

HeuristicLab (Wagner & Affenzeller, 2002) is a tool-kit for heuristic and evolutionary algorithms that is built on the .NET framework. The software offers a wide range of implementations of various different algorithms (not just Genetic Programming), but it is set apart in that it has a GUI designed to allow rapid prototyping of the various algorithms when applied to different problems. HeuristicLab does not offer LGP as one if its available algorithms, but it does provide information on how to model a system that is flexible in terms of the algorithm it uses to achieve its particular task.

Evoasm[4] is an open-source GP system that performs Automatic Induction of Machine code by Genetic Programming (AIMGP), a variant of linear GP which

[1] https://github.com/eclab/ecj
[2] https://github.com/trevorstephens/gplearn
[3] http://scikit-learn.org/stable/
[4] https://github.com/evoasm/evoasm/

represents and manipulates individuals as binary machine code. While AIMGP results in a significant speed up due to direct execution of the machine code, the dependence on a specific processor architecture (x86-64 in the case of Evoasm) restricts portability (Brameier & Banzhaf, 2007). Furthermore, often machine code based systems can be constrained by hardware limitations (e.g. the number of CPU registers). Evoasm does provide functionality for intron elimination in evolved programs as LGP offers.

VUWLGP (Fogelberg & Zhang, 2005) is an LGP system developed at the School of Statistics, Mathematics and Computer Science, Victoria University of Wellington, New Zealand. While VUWLGP is an open-source implementation of LGP, it was released before the work of (Brameier & Banzhaf, 2007) which consolidated and improved upon the previous work in the area of linear GP. As this implementation is guided by this text, it is expected that there is a considerable amount of difference between VUWLGP and this implementation. Moreover, the focus of this work is on providing a modern, cross-platform software package that is readily accessible and available such that the development process is streamlined.

2.4 LGP Usage

LGP has seen successful usage across a broad range of different domains as the GP representation of choice. This section (1) details a subset of the recent usages of LGP to provide further motive for the development of an open-source LGP system, and (2) highlights a selection of the recent developments to LGP.

2.4.1 Applications

Guven (2009) used LGP and Neural Networks as methods for predicting time series of river discharge data. River flow processes are generally accepted to be seasonal and non-linear, meaning the characteristics of stream flow generation are likely to be highly different during different periods. The models were evaluated using a combination of the coefficient of determination R^2, mean squared error (MSE), and mean absolute error (MAE). It was found that both techniques predicted the daily time series of discharge with a high degree of alignment with the observed data. LGP performed moderately better than the NN approach, however. The results support the use of LGP as a tool for predicting non-linear and dynamic river flow parameters.

Remaining in the context of modelling hydrological phenomena, Danandeh Mehr, Kahya, & Yerdelen (2014) investigated the use of LGP in predicting successive-station monthly stream-flow data. Artificial Neural Networks (ANN) have shown promising results in other applications in hydrology, but are often criticized as *black boxes* which are difficult to interpret. Danandeh Mehr et al. (2014) found that while ANNs and LGP both demonstrate an ability to handle the successive-station stream-flow process in general, the LGP model was superior in all of the scenarios examined. Furthermore, the LGP function set specified only the basic arithmetic operators $(+, -, \times, \div)$, meaning that the genetic programs can be represented by mathematical

equations which are preferential to ANNs due to their practical use and ability to be used as tools to identify knowledge from the training data.

The work of Ravansalar, Rajaee, & Kisi (2017) combined traditional LGP and a discrete wavelet transform (DWT) technique to create a hybrid model for monthly stream flow. DWT can be used to capture multi scale features of a signal by decomposing time series into several sub-series. These sub-series can be feed as inputs into the LGP model to predict the stream flow for one month ahead. This hybrid approach was found to perform better than standalone LGP with a superior ability to approximate the non-linear relationship between the inputs and outputs.

G. Wilson & Banzhaf (2010) apply LGP to trading on the foreign exchange market in attempt to evaluate LGP on a previously untested financial market domain. The LGP individuals were organised such that the output register stored the value corresponding to a trade recommendation. Here, a value of 0 in the output register indicated no trade is conducted. A value in the range $\pm[0, 1]$ is multiplied by the maximum dollar amount to be bought or sold per trade. The LGP function set consisted of standard mathematical and logical operators, in addition to traditional analysis metrics such as moving average, momentum, and channel breakout. It was found that the results gathered (and the overall final profits) were competitive with similar studies using other GP techniques, with 85% to 100% of buys being profitable.

Troiano, Birtolo, & Armenise (2016) build on recent investigation of a generative approach based on evolutionary algorithms in order to assist in the design of user interfaces. LGP is considered as a technique for optimizing the layout of a GUI menu system. Experimental results showed the ability of LGP to converge towards high fitness solutions, with solutions being comparable to those designed by humans. This outcome encourages the usage of LGP in such a context in order to aid designers in the process of designing menu systems, reducing human fatigue.

LGP has seen applications in traditional engineering contexts such as the work of Gandomi, Mohammadzadeh S., Pérez-Ordóñez, & Alavi (2014). The study was novel in its approach to applying LGP to a structural engineering problem to build a predictive model for the shear strength of RC beams without stirrups. The model was evaluated based on a multi-objective strategy optimising for model simplicity and high fitness on training and validation data. The LGP model was found to produce better outcomes than existing building codes and the resulting equation demonstrated the ability to capture the underlying physical behaviour. Furthermore, the simplicity of the model makes the integration into practical uses more straight-forward.

2.4.2 Developments

The recent development of so called *soft memory* (McPhee & Poli, 2008) for LGP involves altering assignment to a register such that it does not completely overwrite the value. Instead, the old and new values are combined using a weighted average:

$$v_{combined} = \gamma v_{new} + (1 - \gamma) v_{old}$$

Here, $v_{combined}$ is the final value stored in the register, v_{new} is the newly computed value and v_{old} is the original value in the register. γ is a constant that determines the amount of influence the previous value has on the new value; $\gamma = 1.0$ means that the old value does not affect the new value and the operation will be a *hard* assignment of the new value. Extensive empirical testing found that a soft memory approach performed comparatively to a hard memory implementation in some cases, while in certain instances a soft memory system had significant benefits, suggesting that further development of such a system may be worthwhile. The authors state that further work could refine the exact assignment implementation, with a moving average approach being proposed.

Sotto, Melo, & Basgalupp (2016) propose an improved version of LGP (labelled λ-LGP) which is based on an LGP implementation with only effective micro/macro-mutations. This base is supplemented by additional techniques for determining how individuals are chosen for reproduction. A number (λ) of mutations are applied to each individual in an attempt to encourage exploration of neighbouring zones in the fitness landscape. A set of different criteria determine whether a mutated individual will replace its parent. The implementation was evaluated on the Ant Trail problem where λ-LGP outperformed not only traditional LGP, but other state-of-the-art methods.

Hu, Payne, Banzhaf, & Moore (2011) discuss how the distribution of neutrality within programs affects the relationship between robustness and evolvability in LGP, particularly with respect to mutation-based search. Further to this, Hu, Banzhaf, & Moore (2013) show how the effects of recombination accelerate the evolution process and promote robust programs. It is proven that a combination of recombination and mutation operators is able to accelerate the evolutionary search process, with noticeable acceleration when the population is initialized from a robust phenotype.

Watchareeruetai, Takeuchi, Matsumoto, Kudo, & Ohnishi (2011) propose a technique for the identification and elimination of structural redundancies within a linear representation. This technique allows for the identification of genotypes (representations) which map to the same phenotype (program), as this is not a one-to-one relationship. The existence of such redundancies in a population increase the search space, which on one hand is argued to be a positive (as a means to escape local optima). On the other hand, Watchareeruetai et al. (2011) argue that the execution of such redundant programs should be avoided. It is shown that the avoidance of redundancy can improve the performance of LGP, as it encourages the exploration of the search space, at the phenotype level as opposed to the genotype level.

3 Approach

This implementation follows the fundamental representation and algorithms of LGP as outlined by Brameier & Banzhaf (2007). The system is designed in such a way that encourages modularity, similar to a package-based architecture which provides greater diversity and flexibility (Bouckaert et al., 2010).

The following sections detail the scope of the system, design and implementation details, and rationale for the various decisions made.

3.1 Design and Implementation

3.1.1 Principles

The feature set of the system is primarily focussed on the concepts outlined in the second chapter of the standard LGP reference (Brameier & Banzhaf, 2007) which details program representation, execution and evolution. The specific concepts are detailed here with regards to how they fit within the context of this system.

Program Representation

The program representation used in this implementation mainly follows that outlined by Brameier & Banzhaf (2007), with the addition of *generic registers*. Traditionally, LGP programs operate on a set of registers which contain floating-point values and the operations used must be appropriate for that data type. This system takes a more liberal approach, leaving the register data type generic with the goal of presenting a higher degree of flexibility to the user.

The implication of this is that the user will need to define custom operations to account for the data type of the registers. This is the intended outcome as it allows for greater scope in terms of what the genetic programs are able to accomplish. The system provides default implementations of the core components to be compatible with the traditional case of floating-point registers, but the flexibility exists to use any data type that a problem may need.

Program Execution

Program execution is achieved through a simple virtual machine built into the LGP system, that is able to understand and execute the instructions belonging to an evolved program. However, the organisation of the system allows for custom execution models to be defined where required.

The incorporation of algorithms for the elimination of non-effective code (Brameier & Banzhaf, 2007) ensures that the system has optimal performance and takes advantage of the linear representation.

Program Evolution

In addition to the traditional LGP approach, the system is supplemented with different models for evolution. The traditional LGP algorithm is available in addition to two parallelised algorithms described by Alba, Luque, & Nesmachnow (2013). The first parallelised algorithm utilises a master-slave parallel model where the computationally expensive portions of the EA are offloaded onto a set of threads, to be performed in parallel for increased performance characteristics.

The second model, distributed island migration, describes an algorithm that has a number of distinct populations which solutions can migrate between. Each population is evolved in parallel and the gene pool of the populations is distributed amongst the islands. This allows for greater diversity as exploration occurs between populations and exploitation occurs within populations.

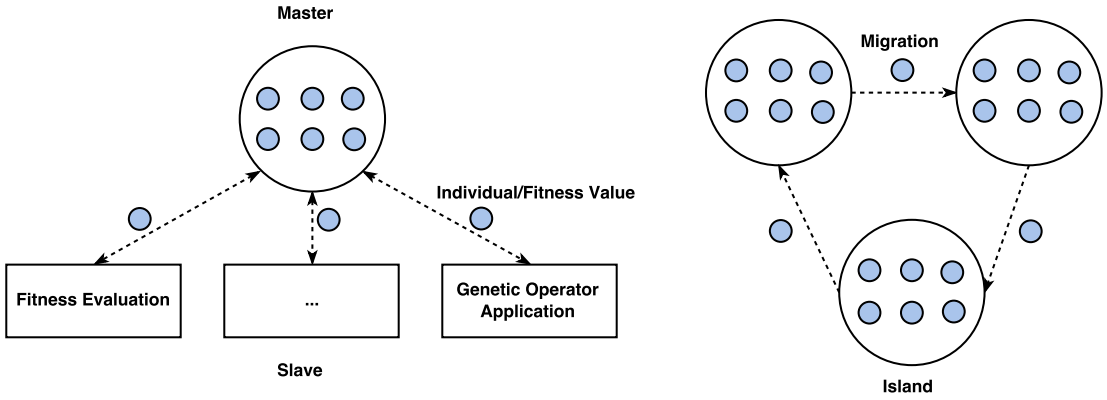


Figure 5: The two parallel models given as additions in the system. The master-slave model is given on the left, where slaves perform the computationally expensive tasks of fitness evaluation and even genetic operator application (for large populations). The right hand side model, island migration, shows distinct populations which solutions can migrate between to encourage diversity between the populations.

As well as these core concepts, a set of key design principles guides the direction of implementation with regards to design decisions:

Cross-platform

The system is able to be used on the major operating systems (MacOS, Windows, Linux) to allow for straight-forward integration into existing environments. Being a cross-platform system also widens the scope of potential users.

Extensible

A well-defined public API enables the system to be extended where necessary in order to adapt to a particular problem domain. The API grants access to creating a range

of custom modules, permitting customization of the following elements:

- Instruction set.
- Fitness function.
- Search operators (selection, recombination, mutation).
- Evolutionary algorithm.
- Program/instruction representations.

The design of a modular system is of great importance for this implementation. To ensure that the system's extensibility can be used to its fullest, extensive documentation is made available with the system. This includes documentation of the design and architecture, instructions for adapting the system to a new problem (e.g. defining new instructions). This core documentation is supplemented by well documented source code allowing for one to familiarise themselves with the code quickly.

Efficient and Modern

Modern language features are taken advantage of to provide a system that is efficient in terms of memory management and runtime (through parallelism). The system is built upon Kotlin, a modern programming language that is built on the Java Virtual Machine providing:

- Cross-platform support.
- Full access to the Java standard library in conjunction with an idiomatic standard library.
- Full Java interoperability.
- Support for modern functional interfaces and null-safety.
- Reified generics providing safer type-casting behaviour (important for a modular system).

3.1.2 System Architecture

Figure 6 provides a high-level overview of the different components which comprise the system and how they interact. Each of the components is detailed in the following sections as well as how they are used to perform the various LGP algorithms.

This section does not provide a usage guide for the software, as that role is fulfilled by the on-line documentation (http://lgp.readthedocs.io/en/latest/). Regardless, to supplement the concepts explained here, example code for setting up a problem within LGP system is given in Appendix A. This particular code sets up the Keijzer-6 benchmark as described in Section 4.1.1.

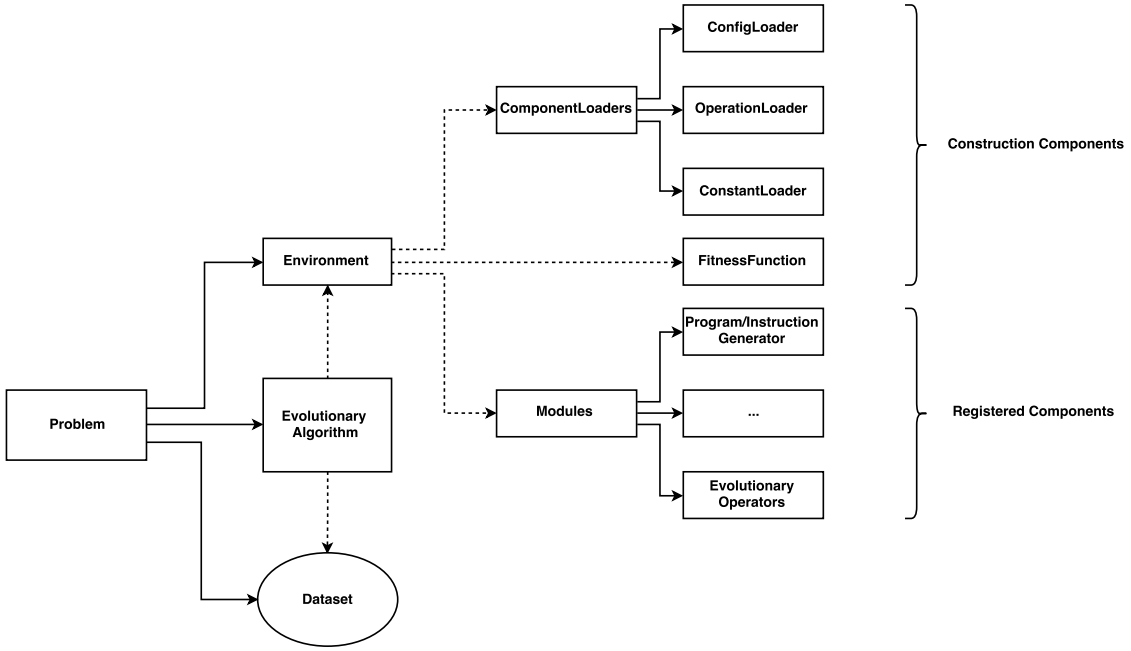


Figure 6: High-level overview of the system design.

Problem

At a high-level, a problem in the LGP system is essentially a wrapper that encapsulates the various details of a problem and the components that can be used to find solutions to it. Namely, a problem is comprised of data attributes (e.g. name, description of the problem, training/test datasets) and a collection of dependencies that can be used to initialise the system in order to find solutions for that particular problem.

There are three essential dependencies that must be resolved in order to build a problem: An environment in which the problem is defined, an evolutionary algorithm that is used to guide the search process, and a dataset that provides information for tailoring the genetic programs to the particular problem being targeted. Each of these is explained in the subsequent sections.

In terms of implementation, a problem is an abstract class with the structure given by Figure 7. The order in which the methods are called is important, as the environment must be initialised before the model, both of which must be initialised before the problem can be solved. As the class is abstract, it is up to the user defining the problem to provide a concrete implementation, leaving the problem definition wide open. This freedom is given as it is likely that each problem will have a wide range of requirements and possible configuration details that the user may which to tune.

`initialiseEnvironment()` gathers any dependencies that are required by the environment in order to perform the processes of LGP. The particular dependencies required are documented in the next section. As the parameters of the environment can vary dramatically, they are particularly influential on the result of the search

and the consolidation of these parameters in one central location encourages rapid experimentation.

`initialiseModel()` performs the tasks necessary to configure the evolutionary algorithm provided. As the particular implementation of the algorithm is given to the user, this may include custom set up logic as the user needs (e.g. initialising an external simulation system for evaluating the genetic program solutions).

Lastly, the `solve()` method begins the evolutionary search process. It is essential that the environment and evolutionary algorithm be initialised before a problem can be solved, as they provide the information needed to perform the search. The user can also define the shape of their solutions, depending on the information they want to be given at the completion of the process.

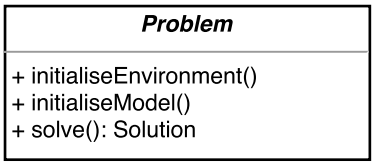


Figure 7: Problem class diagram. The class definition is abstract and it is expected that a concrete implementation be given that defines the particular problem to be solved.

Environment

As mentioned previously, the environment is an important piece of the puzzle when defining a problem to be solved using this system. The environment acts as a central repository for core components of the LGP system. 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: *Construction*, *Initialisation*, and *Registered*. These components and their order are described in the following sections.

Construction Components

Construction components are those that are required when building an environment instance and are passed to the class constructor (hence construction components). These components provide the base information necessary for resolving further components. The set of components that fall under the construction label are: *Component Loaders*, *Default Value Provider*, *Fitness Function*.

Like most parts of the system, *Component Loaders* are a module that provide the ability to load components. They are not strictly a component themselves, but they represent a promise to load a component when instructed to through the interface given by Figure 8.

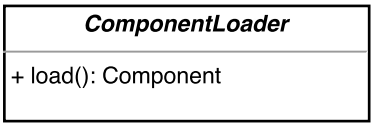


Figure 8: A `ComponentLoader` provides a simple interface that is a promise to load a component when requested.

An environment requires three component loaders: `ConfigLoader`, `ConstantLoader`, and `OperationLoader`.

The `ConfigLoader` is as the name suggests, responsible for loading configuration information (a component in the system). A configuration loader does not in any way restrict the process with which configuration is loaded, it simply defines an interface that promises to provide configuration in the appropriate form when requested. This open interface means that configuration could be loaded from a file stored on a machine's local hard disk, a database, or over a network connection — the environment is deliberately oblivious to the details of the implementation.

A `ConstantLoader` works in a similar fashion to the `ConfigLoader`. It provides constant values when requested, which can be used by LGP programs. The particular method used is not important, as constants may be provided in various ways: they could be hard-coded values or they could be randomly generated.

An `OperationLoader` provides the operations (functions) that the system is able to make use of when evolving programs. Operations in this system are modular components, allowing for the possibility of the definition of custom operations for the particular problem being solved. In some cases, the built-in operations will be suited to the problem and the default operation loader provided can be used. In cases where custom operations are used however, there may be some custom logic required to load the operations appropriately.

The implementation provides a set of default component loaders for the common scenarios.

The *Default Value Provider* represents a generator which can produce the values that are used to populate the calculation registers of the register set. Generally, the built in constant value provider will be used so that all calculation registers have the same initial value, but flexibility is built into the API to allow for custom logic (e.g. random calculation register values).

The *Fitness Function* is a crucial component for the LGP system as it is the metric used to determine which solutions are better than others, in order to guide the evolutionary search. In this system, a fitness function has a simple definition: it is a function which takes a list of predicted target values and a list of expected target values and returns a simple floating-point measure that determines how well the predictions fit the target values.

Initialisation Components

Initialisation components operate at a level below construction components and are somewhat concealed from the user. They are automatically loaded by an environment when a set of suitable construction components have been given. Generally, an initialisation component will be some component that has been loaded from a component loader and acts as a kind of *global state* that isn't affected by the LGP system (e.g. configuration information loaded from a `ConfigLoader`).

The most important initialisation component is the register set which is created during the initialisation of the environment. The configuration information given when constructing the environment will detail the number of input registers and the number of calculation registers, while the `ConstantLoader` will provide a set of constants. This allows for the register set to be initialised with the correct initial values and stored as part of the environment, where it is kept as a *global reference* register set. Each time a program individual is created it receives a new copy of this reference register set.

Registered Components

Registered components are those which the system regards as modular. The system is built upon the concept of *modules* which describe self-contained components that are used to fulfil certain roles within the system (e.g. search operators, program generation schemes). A modular design allows for the system to be malleable to different problems as custom functionality is easily defined.

The rationale behind this design is to allow for a high degree of flexibility regarding the particular arrangement and configuration of the various components made available to the system. Furthermore, users can implement their own modules and register their custom behaviour as necessary.

A registered component requires a reference to the environment in which it operates as it may be useful for a custom component to make use of information that is stored within the environment (e.g. configuration details, the register set).

Registering a component involves associating a module type (i.e. the type of the component) with a particular instance of the module. There is no restriction on the type of components allowed, meaning that the system is *open* and custom functionality can be inserted where necessary (a plug-in approach). The environment stores a mapping of module type to the implementation of that module and allows access to that implementation from within the system.

Modules

The hierarchy that the system uses to organise its modules is illustrated by Figure 9. These modules are situated at the core of the system and provide the means for performing LGP. An overview of the modules that comprise this hierarchy and their implementation details is provided here.

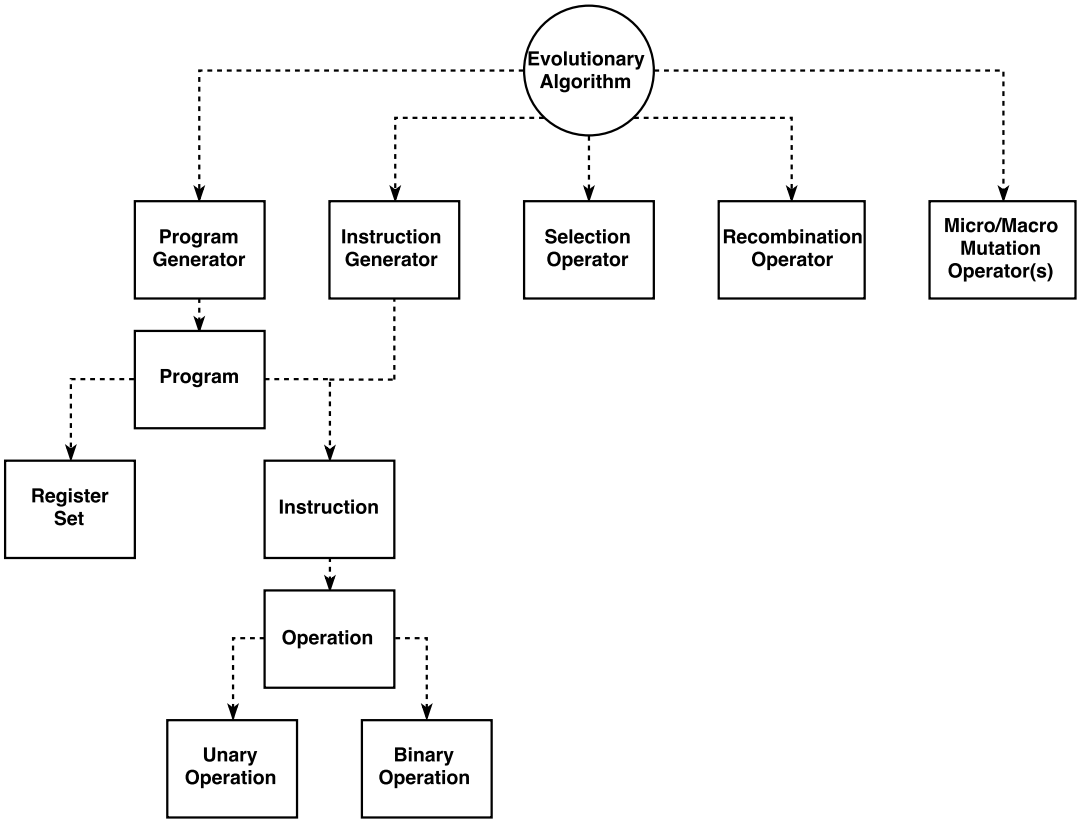


Figure 9: Hierarchy of the modules that comprise the core of this LGP system.

Evolutionary Algorithm

In this system, the evolutionary algorithm (represented by the `EvolutionModel` class) is an abstract concept used to describe the core evolution process. In its most basic form, the EA provides a way for itself to be trained on a data set and tested on a data set. Each of these operations will produce a result that describes the state of the model.

Training an EA performs the process of evolution to build a population of solutions and find a best solution. Testing the EA generally involves using the best solution to form an evaluation for an unknown set of data (i.e. not the data used for training). In this case, the `EvolutionModel` is being used as a predictor.

Brameier & Banzhaf (2007) describe a simple Steady-State EA (as detailed in Section 2.2.3) that forms the basis of their LGP system. A modified version of this algorithm that does not perform validation steps is implemented in the system as an `EvolutionModel` that can be used to perform LGP with. In addition to this, the system is left open to the possibility of alternative EAs that may make sense for particular problems.

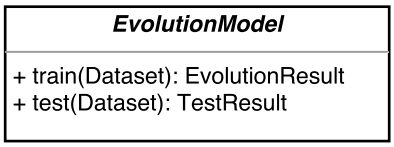


Figure 10: `EvolutionModel` class diagram

Program Generator

The system represents genetic programs with the `Program` class (see Figure 11). A program in the LGP system is defined as a module meaning that the exact details of how the program operates is left up to the user of the system. This open interface is different from other systems which often impose their own structure and logic on the representation of programs.

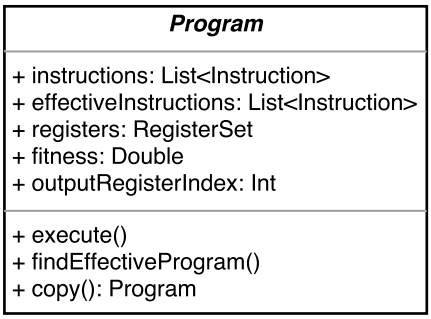


Figure 11: `Program` class diagram

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.

Similarly, the instructions that a program is comprised of are also modules to the system, meaning that the exact effect of an instruction can be left up to the user. In a typical LGP system, instructions would represent pure mathematical functions which don't introduce side-effects (Brameier & Banzhaf, 2007), however the system allows for the ability for instructions to perform any operation they need — provided that the instruction receives its inputs through a set of registers.

The rationale behind designing the system in this way as opposed to restricting the program/instruction representation is to allow further flexibility with regards to applying LGP to problems that don't necessarily have a strict mathematical representation (e.g. control problems). In such a context, instructions might be

commands such as move left or rotate 90° and inputs would come from the environment within which those actions occur.

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 (Brameier & Banzhaf (2007) acknowledge random, effective, maximum-length, constant-length, and variable-length initialization techniques). The program generator is expected to act as a stream of programs so that other components in the system can continuously generate new programs.

Instruction Generator

An instruction is represented by the modular `Instruction` class which itself is built upon the `Operation` module. An `InstructionGenerator` must be defined to create instructions that are valid (or the default implementation can be used). An operation in this system represents the combination of a function (e.g. +) and an arity (i.e. the number of arguments the function expects). A function encapsulates the behaviour of a transition from an argument or set of arguments to an output value (e.g. `Function<T> = (Arguments<T>) -> T` where `T` refers to the type of the registers). The operation level also defines the representation of the function, allowing for custom translation behaviour when instructions are exported (e.g. instead of the C programming language, the system may wish to target the JVM).

The Unary and Binary operation classes are provided as default operation types that expect 1 or 2 arguments respectively. As the majority of operations used by the LGP system will fit under these two operation types, it is beneficial for them to be included as defaults.

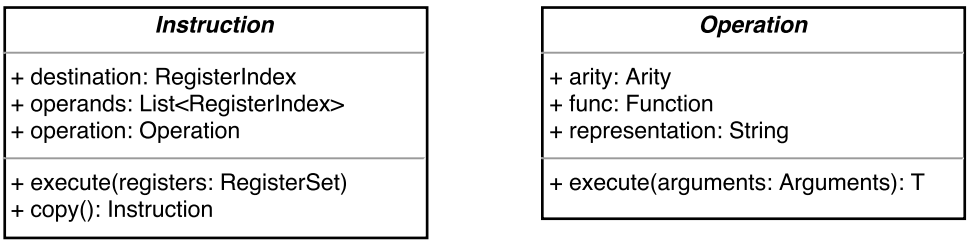


Figure 12: `Instruction` and `Operation` class diagrams

Selection Operator

As stated in Section 2.2.3, LGP as described by Brameier & Banzhaf (2007) traditionally uses tournament selection as the means of deciding the individuals which will be exposed to the effects of the variation operators. While the system implements tournament selection, the interface is also left open so that other selection algorithms (e.g. fitness proportionate selection or reward-based selection) can be implemented and utilised as part of the evolutionary algorithm. The selection operator interface is provided by the `SelectionOperator` abstract class and is given in Figure 13.

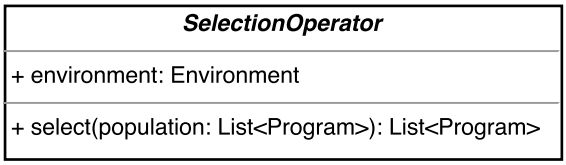


Figure 13: `SelectionOperator` class diagram

Recombination Operator

The traditional EA given in Section 2.2.3 performs two-point linear crossover (see Figure 3). While this method is generally used, Brameier & Banzhaf (2007) do detail various other recombination operators for linear programs (e.g. one-point crossover, one-segment recombination). The system implements two-point linear crossover as described by Brameier & Banzhaf (2007) but any recombination operator could be used by the system if implemented using the interface given in Figure 14. The API for two-point linear crossover as it is implemented is also included to illustrate how the base interface can be extended to provide further functionality (e.g. the addition of algorithm parameters). It should be noted that the implementation performs the recombination in place, so that the individuals are directly modified.

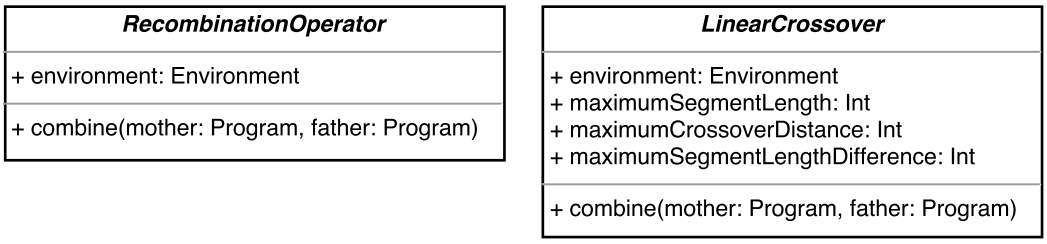


Figure 14: `RecombinationOperator` class diagram

Micro/Macro Mutation Operators

In its basic form, a mutation operator has the simple interface given by Figure 15. However, the API provides two implementations of this interface: `MicroMutationOperator` and `MacroMutationOperator`. These implement the effective micro and macro mutation algorithms as given below. Where external parameters are needed (e.g. maximum/minimum program length, rate of constants), the environment facilitates access so that the algorithms can gather the information needed. The micro mutation operator also takes a custom constant mutation function as a parameter, which accounts for the case where registers don't contain simple floating-point values.

As with the other modular components of the system, a user has the ability to build their own mutation operators as is required.

Effective Macro Mutation

Parameters: insertion rate p_{ins}; deletion rate p_{del}; maximum program length l_{max}; minimum program length l_{min}.

1. Randomly select macro mutation type *insertion* | *deletion* for probability p_{ins} | p_{del} and $p_{ins} + p_{del} = 1$.
2. Randomly select an instruction at a program position i (mutation point).
3. If $l(gp) < l_{max}$ and (*insertion* or $l(gp) = l_{min}$) then:
   (a) Insert a random instruction at position i.
   (b) If effective mutation then:
      i. If instruction i is a branch go to the next non-branch instruction at a position $i := i + k$ $(k > 0)$.
      ii. Run non-effective code elimination until program position i.
      iii. Randomly select an effective destination register $r_{dest}(i) \in R_{eff}$.
4. If $l(gp) > l_{min}$ and (*deletion* or $l(gp) = l_{max}$) then:
   (a) If effective mutation then select an effective instruction i if existent.
   (b) Delete instruction i.

Effective Micro Mutation

Parameters: mutation rates for registers p_{regmut}, operators $p_{opermut}$, and constants $p_{constmut}$; rate of instructions with constant p_{const}; mutation step size for constants d_{const}.

1. Randomly select an *effective* instruction.
2. Randomly select mutation type *register* | *operator* | *constant* for probability p_{regmut} | $p_{opermut}$ | $p_{constmut}$ and $p_{regmut} + p_{opermut} + p_{constmut} = 1$.
3. If *register* mutation then:
   (a) Randomly select a register position *destination* | *operand*.
   (b) If *destination* register then select a different *effective* destination register the non-effective code elimination algorithm.
   (c) If *operand* register then select a different *constant* | *register* for probability p_{const} | $1 - p_{const}$.
4. If *operator* mutation then select a different instruction operator randomly.
5. If *constant* mutation then:
   (a) Randomly select an *effective* instruction with a constant c.
   (b) Change constant c through a standard deviation d_{const} from the current value: $c := c + \mathcal{N}(0, d_{const})$.

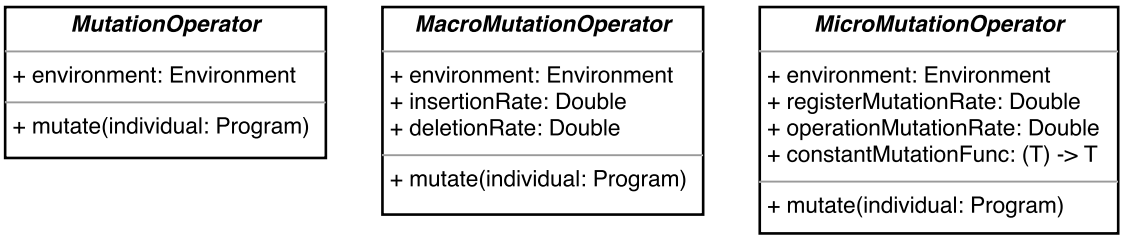


Figure 15: `MutationOperator` and implementations class diagrams

Extensions

When describing the environment component of this system, it was mentioned that modules are registered by mapping a module type to a particular implementation of module. The modules that have been included in the core API of the system have been detailed, but the way in which the system is designed allows for the registration of any type of module.

This leaves the system open for future developments to LGP and GP in general, as the system can be easily extended to make use of novel search operators or evolutionary algorithms. Alternatively, a simpler module could be included to perform tasks such as logging of the evolutionary process or real-time aggregation of the results.

4 Evaluation

To evaluate the system, a set of standard benchmark problems were implemented within the confines of the system. The benchmarks (which are detailed in the subsequent sections) include a combination of synthetic and real-world symbolic regression problems. Varying configurations will be used to show the effects different parameters have on the results of the system. These benchmarks intend to test the system's capabilities and gather insight about the influence of different configurations on the system's performance.

As a secondary stage of evaluation, the best performing configurations are utilised to test the abilities of the different parallelised evolutionary algorithms implemented as described in Section 3.1. Here, the techniques are compared based on measurements of three statistics created by the evolutionary process: fitness, runtime, and population diversity.

Finally, a comparison of LGP with results gathered from the execution of a tree-based genetic programming solution and a linear regression model on a subset of the benchmarks is made. These demonstrate problems in which a linear representation has an advantage as well as highlighting comparable performance between techniques.

The following sections detail the benchmark problems, as well as outlining the experimental methods and configurations.

4.1 Benchmarks

4.1.1 Synthetic Symbolic Regression

White et al. (2013) propose a set of benchmark problems for evaluating GP systems. These problems are chosen to address issues identified with typical GP benchmarks, such as the relative ease with which they can be solved. This subsection outlines the symbolic regression problems taken from White et al. (2013).

Keijzer-6

The Keijzer-6 problem is derived from Keijzer (2003) where a training and test range are proposed. The problem requires extrapolation, not just interpolation (White et al., 2013) due to the usage of the sum operation. In comparison to the other benchmark problems, it is relatively simple due to its low dimensionality.

$$f_1(x) = \sum_{i=1}^{x} \frac{1}{i} \tag{2}$$

Korns-12

Korns-12 remains unsolved in Korns (2011). The problem is interesting in that the dataset contains 5 input variables, but only 2 have an affect on the output of the function. The problem focuses on testing the system's ability to determine unimportant input variables and avoid their use in approximating the function.

$$f_2(x_0, x_1, x_2, x_3, x_4) = 2.0 - (2.1 \times (cos(9.8 \times x_0) \times sin(1.3 \times x_4))) \tag{3}$$

Vladislavleva-4

This problem originates from Vladislavleva et al. (2009) and is described by the authors as their "favourite problem". The problem is dubbed *UBall5D*[5] (see Figure 16 for an illustration) and the original authors state their GP system has difficulty finding the simple and harmonious input–output relationship. Like the Keijzer-6 problem, Vladislavleva-4 requires extrapolation.

$$f_3(x_0, x_1, x_2, x_3, x_4) = \frac{10}{5 + \sum_{i=1}^{5}(x_i - 3)^2} \tag{4}$$

Nguyen-7

The other benchmark problems primarily consist of relatively basic arithmetic and trigonometric operations. Nguyen-7 (Uy, Hoai, O'Neill, McKay, & Galván-López, 2011) is a logarithmic function and thus requires a different operation set from the

[5]Five-dimensional unwrapped ball.

other problems. Like the Keijzer-6 problem, Nguyen-7 has a lower dimensionality than the other problems.

$$f_4(x) = ln(x+1) + ln(x^2+1) \tag{5}$$

Pagie-1

Pagie & Hogeweg (1997) define the Pagie-1 as a simple 2-D numerical function which can be scaled in terms of difficulty by introducing more variables. It has a reputation for difficulty using standard GP. Harper (2012) state that across all the literature, standard GP failed to solve the problem and in their own paper, approximately 12% of the runs were successful. The reason for such difficulty is that the GP population becomes bloated very quickly, which halts the system's ability to produce fitness improvements.

$$f_5(x,y) = \frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}} \tag{6}$$

Table 2: A summary of the synthetic symbolic regression benchmarks used to evaluate the system. In the training and testing sets, $U[a,b,c]$ refers to c uniform random samples drawn from a to b, inclusive. $E[a,b,c]$ is a grid of points evenly spaced with an interval of c, from a to b inclusive.

Name	Equation	Training Set Testing Set
Keijzer-6	$\sum_{i=1}^{x} \frac{1}{i}$	$E[1,50,1]$
		$E[1,120,1]$
Korns-12	$2.0 - (2.1 \times (cos(9.8 \times x_0) \times sin(1.3 \times x_4)))$	$U[-50,50,10,000]$
		$U[-50,50,10,000]$
Vladislavleva-4	$\frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$	$U[0.05,6.05,1,024]$
		$U[-0.25,6.35,5,000]$
Nguyen-7	$ln(x+1) + ln(x^2+1)$	$U[0,2,20]$
		$U[0,2,100]$
Pagie-1	$\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$	$E[-5,5,0.4]$
		$U[-5,5,1000]$

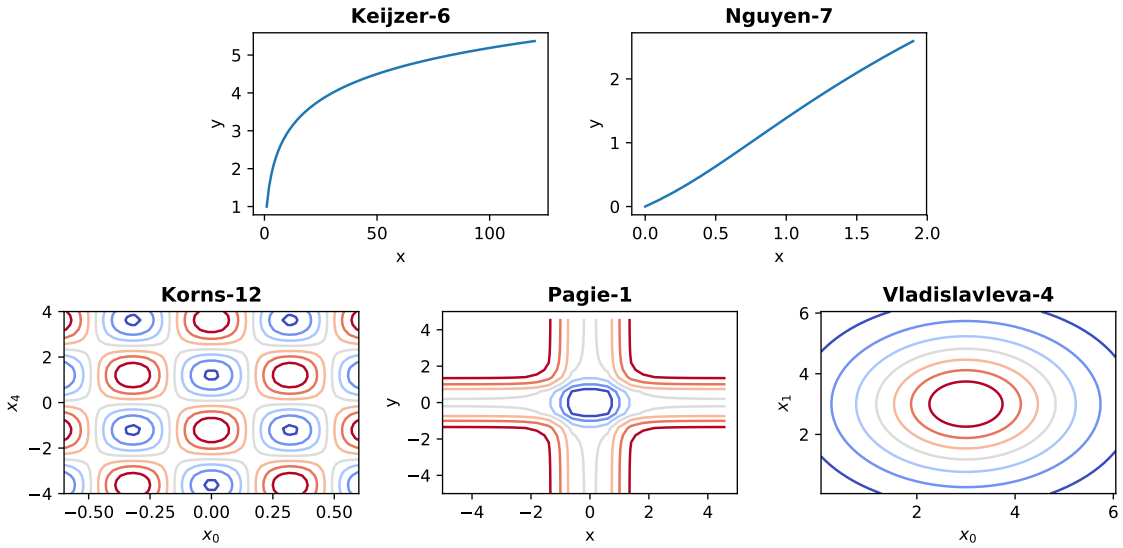


Figure 16: Illustrations of the synthetic symbolic regression problems. The higher dimensional problems have been represented as projections onto 2-D intervals. The Pagie-1 problem is represented as a contour plot of the 3 dimensions; Korns-12 is plotted using the 2 dimensions that the function actually utilises; Vladislavleva-4 is shown with respect to x_0 and x_1, with $x_2 = x_3 = x_4 = 0$.

4.1.2 Real-world Regression

In combination with the synthetic symbolic regression problems, three real-world symbolic regression problems are employed to demonstrate the system's performance in a more practical setting. These problems are chosen for their varying degrees of difficulty and linearity.

Wine Quality Data Set

The first problem involves modelling wine quality based on physico-chemical tests (Cortez, Cerdeira, Almeida, Matos, & Reis, 2009). There are two datasets relevant to red and white variants of the Portuguese "Vinho Verde" wine, each comprised of twelve features; the first eleven features are real-valued input variables based on physico-chemical tests. The twelfth feature, wine quality, is the output variable and is based on sensory data, scored between 0 and 10.

Parkinson's Data Set

The second problem, from Tsanas, Little, McSharry, & Ramig (2010) requires predicting a clinician's Parkinson's disease symptom score on the UPDRS scale from a set of 16 voice measure inputs. The dataset consists of a range of biomedical voice measurements from 42 people with early-stage Parkinson's disease recruited to a six-month trial of a tele-monitoring device for remote symptom progression monitoring.

There are two outputs that are targeted, the motor UPDRS and the total UPDRS. For the purposes of these experiments, only the total UPDRS measure is targeted.

Gene Expression Survival Data Set

The third and final real-world regression problem requires using a set of gene expression data to predict survival time, as described in Lenz et al. (2008). The problem displays a high level of dimensionality, consisting of 100 features which have a highly non-linear relationship. This problem will only be used during the comparison to linear regression to measure the performance of LGP with regards to a high-dimensional, non-linear problem.

4.2 Experiments

Details of the experiments used to evaluate the system are detailed in the following sections. The first goal is to evaluate the effects of different configurations on the system. From this, a baseline set of configurations that work well across problems is investigated alongside configurations which are particularly suited to individual problems. It is expected that these configurations do not necessarily provide the absolute best result in terms of fitness, as the goal is to evaluate the different effects of the various parameters the configurations target. Secondly, a demonstration of the performance of the parallelised evolutionary algorithms within the context of an LGP system is given. Finally, a comparison to both tree-based GP and a tradition linear regression model is made.

4.2.1 Effects of Parameter Combinations

The first round of experiments is designed to determine a combination of parameters that produces the best results on average across the benchmarks problems, as well as identify the combinations which perform best on a particular benchmark. From this, relationships between parameter combinations can be investigated to determine the cause behind differing performance characteristics.

Parameters specific to LGP that are believed to have the most significant effects on the results have been chosen, namely: (1) initial and overall program lengths, (2) number of calculation registers, (3) operation set, and (4) balance of micro and macro-mutation rates.

Initial and overall program lengths determine the amount of genetic material that exists within a population and sets a level of diversity. The number of calculation registers greatly effects the variability within programs, as it dictates the number of sub-graphs within a program graph. Genetic program expressiveness is controlled through the operation set provided, by defining the programming language that programs are built with. Finally, the balance of micro and macro-mutation has an effect on the amount of exploitation and exploration the system employs.

Round 1 measures the effects on fitness performance of 81 different combinations of parameters across the set of benchmark problems. Each benchmark problem is configured and executed with each of the 81 different parameter combinations. 10 independent runs occur per benchmark, resulting in 810 total runs per benchmark. Each parameter has 3 settings from which it can draw: a restrictive value, a conservative value, and a generous value. The exact values for these settings have been separated into 4 tables (one for each parameter) and are described in the following sections.

Program Length

The system splits the configuration of program length into four separate, but related parameters: initial minimum program length (I_{min}), initial maximum program length (I_{max}), minimum program length (O_{min}), and maximum program length (O_{max}). The initial minimum and maximum program length parameters determine the length of programs which are randomly generated at the start of the evolutionary process. The minimum and maximum program length parameters set a limit on the size of programs as the evolutionary process is carried out — a programs length will always be within the minimum and maximum program length settings.

A restrictive setting dictates that initial programs will have between 10 and 20 instructions with an upper bound of 30 instructions during the evolutionary process. This setting has been chosen as it introduces a limit on the amount of diversity within a population. A conservative, middle-of-the-road setting is chosen to allow programs to grow to a relatively complex size whilst allowing for initial programs that have a significant amount of genetic material. Lastly, the generous setting is designed to provide a large amount of genetic material and encourage complicated programs. While this generally proves useful, there are cases where too longer programs can cause the trajectory of the evolutionary process to follow a narrow path where it can not escape to reach better solutions.

Table 3: Values for the initial and overall program lengths. The values refer to the number of overall instructions in a program. When generating an initial program, its length will be chosen randomly from the range given by (I_{min}, I_{max}).

	Initial (I_{min}, I_{max})	**Overall** (O_{min}, O_{max})
Restrictive	(10, 20)	(10, 30)
Conservative	(30, 60)	(30, 100)
Generous	(50, 100)	(50, 200)

Calculation Registers

The number of calculation registers is important in LGP as it can determine the amount of variability with regards to sub-graphs in programs. A restrictive value

limits the number of sub-graphs and encourages overwriting of the input registers. On the other side of the spectrum, too many calculation registers can support independent calculations which do not make use of previously calculated values. The conservative value attempts to provide a balance between the two extremes.

Table 4: Values for the number of calculation registers made available to programs in the LGP system.

	# Calculation Registers
Restrictive	2
Conservative	6
Generous	10

Operation Set

In LGP, the operation set defines the programming language of the genetic programs. A restrictive operation set limits the ability of the programs to model complex functions whereas a generous operation set can dramatically increase the size of the search space, making it more difficult to discover the optimum solution. As the benchmarks all involve mathematical modelling in the form of symbolic regression, the operation sets consist of varying degrees of arithmetic, trigonometric, power and logarithmic functions. The most generous operation set also allows for conditional expressions to be utilised.

Table 5: Values for configuring the operation set available to programs in the system.

	Operation Set
Restrictive	$+, -, \times, \div$
Conservative	$+, -, \times, \div, sin, x^y, sqrt, ln$
Generous	$+, -, \times, \div, sin, x^y, sqrt, ln, x^2, if >, if \leq$

Micro/Macro Mutation Rates

The system allows the mutation rate to be configured in terms of the balance between micro and macro mutations. A high level of macro mutation encourages exploration of the solution space as opposed to a high level of micro mutation which promotes exploitation of initial solutions. The type of these parameters is described as restrictive, conservative, and generous as with the other parameter combinations, but it may be more accurate to describe them in terms of how they navigate the search space — exploration (restrictive), exploitation (generous), or a balance (conservative).

Table 6: Values for micro and macro mutation rates in the system.

	Micro Mutation Rate	**Macro Mutation Rate**
Exploration	25%	75%
Balanced	50%	50%
Exploitation	75%	25%

Other Parameters

To ensure the effects of the various combinations are able to be analysed consistently, the other parameters that the system offers will remain constant throughout the series of experiments. The parameters have been chosen to be relatively conservative based on experiments outlined by Brameier & Banzhaf (2007).

Table 7: General parameters that remain constant in the system. A relatively balanced set of parameters are chosen to ensure consistency as the aforementioned parameter combinations are tested.

Parameter	**Value**
Population Size	500
Generations	500
Constant Rate	50%
Constants	$\{-1, 0, 1\}$
Number of Offspring	2
Tournament Size	4
Crossover Rate	50%
Recombination Operator	Linear Crossover
Maximum Segment Length	6
Maximum Crossover Distance	5
Maximum Segment Length Difference	3
Macro Mutation Operator	Effective Macro Mutation
Insertion Rate	50%
Deletion Rate	50%
Micro Mutation Operator	Effective Micro Mutation

Register Mutation Rate	33.3%
Operator Mutation Rate	33.3%
Constant Mutation Rate	33.3%
Constant Mutation Function	Random Gaussian Noise

As the benchmarks come from a wide variety of sources, they do not all use the same fitness function. The particular objective function targeted is sourced from the original literature in which the problem is described and are repeated below.

Table 8: Fitness functions used by each individual benchmark.

Benchmark	**Fitness Function**
Keijzer-6	Mean-Squared Error
Korns-12	Mean-Squared Error
Nguyen-7	Mean-Squared Error
Pagie-1	Mean-Absolute Error
Vladislavleva-4	Mean-Squared Error
Red Wine Quality	Mean-Absolute Error
White Wine Quality	Mean-Absolute Error
Parkinsons Total	Mean-Absolute Error
Gene Expression Survival	Mean-Absolute Error

4.2.2 Evolutionary Algorithm Comparison

In order to draw comparisons between the different evolutionary algorithms the system offers, combinations identified from the first round of experiments are used to collect a set of statistics from the EAs. These will capture the advantages and disadvantages of each EA from the point-of-view of the benchmark problems.

Three evolutionary algorithms are provided by the system — the traditional steady-state LGP EA (Brameier & Banzhaf, 2007), as well as two parallelised algorithms: master-slave and island migration (Alba et al., 2013). To compare these techniques, three measures are utilised: (1) fitness of the best solution found, (2) runtime of the evolutionary process, and (3) diversity within the population over the course of the EA. Subsequent sections detail the applied measurement methods further.

Each benchmark is run 30 times using each EA to measure the statistics listed above. Additional settings required for the island-migration EA are specified in Table 9.

The best parameter combinations established for each benchmark in Round 1 will be used to configure the system, with all other parameters remaining unchanged from those specified for first round of experiments (see Table 7).

Fitness

Fitness is measured by the typically employed method — application of a genetic program solution to a set of input-output examples where the error is measured by an objective function. The particular fitness functions used by the benchmark problems are given below, where Y is a vector of n expected outputs and $\hat{Y}$ is a vector of predicted outputs:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{7}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{Y}_i - Y_i| \tag{8}$$

Runtime

Runtime is measured in milliseconds from the start of the evolutionary process until the termination. A measurement in milliseconds is used to provide a fine-grained level of precision.

Diversity

Brameier & Banzhaf (2007) outline a technique for measuring the distance between two linear genetic programs using string edit distance. This process involves representing the program's effective code as a string of operations (e.g. (-, +, /, +, *, -)). The edit distance between two programs operation sequence can be computed using an edit distance algorithm (in this case, Levenshtein distance) to determine the structural difference between the effective parts of the programs. A difference in effective code is more likely to be directly related to a difference in program behaviour.

To measure diversity within an EAs population, a random sample of 500 pairs of programs are chosen every 50 generations to have their edit distance computed and aggregated. This aggregation can be used to determine the average level of diversity over the course of the EA.

Table 9: Parameters for the island migration evolutionary algorithm.

Parameter	**Value**
Number of Islands	6
Migration Interval	Every 100 Generations
Migration Size	5 Individuals

4.2.3 Comparison to TGP and Linear Regression

As a final method of evaluation, a subset of the benchmark problems are tested against a tree-based GP (TGP) system (`gplearn`) and a Linear Regression model (Weka). The synthetic problems are compared using the tree-based system, while the linear regression model is used to draw comparisons for the real-world problems. The raw fitness performance is collected and compared to the results from the LGP system to quantify the comparability between the two techniques. Furthermore, the average program length from both LGP and TGP will be collected and compared.

Tables 10 and 11 detail the parameters used in the comparison of the LGP and TGP systems. The parameters are intended to be fair between the two systems, and are not optimised for any individual benchmark problem. Both systems are configured to have the same population size, number of generations, constant values and function set. Although the individual mutation operators differ, both systems are configured with a bias towards less destructive mutations; this is displayed by a favouring of micro-mutations for LGP and a disfavouring of sub-tree mutations for TGP.

Table 12 lists the parameters used to configure LGP when compared against the linear regression model. Linear regression is performed using the `weka.classifiers.functions.LinearRegression` classifier from Weka (M. Hall et al., 2009). The linear regression model offers little in the way of configuration, so the default built-in configuration was used. Parameters used to configure the LGP system are relatively conservative and aim to be benchmark agnostic.

The rationale behind different parameters in each comparison is that the synthetic regression problems require slightly less extreme parameters than the real-world problems.

Table 10: Configuration of the LGP system for the comparison of LGP and TGP.

Parameter	**Value**
Initial Min. Program Length	30
Initial Max. Program Length	60
Min. Program Length	30
Max. Program Length	200
Constants	$\{-1, 0, 1\}$
Constant Rate	40%
# Calculation Registers	8
Population Size	500
Generations	100
Micro-Mutation Rate	75%

Macro-Mutation Rate	25%
Number of Offspring	4
Tournament Size	4
Crossover Rate	50%
Macro-Mutation Insertion Rate	50%
Macro-Mutation Deletion Rate	50%
Register Mutation Rate	33.3%
Operation Mutation Rate	33.3%
Constant Mutation Rate	33.3%
Operation Set	$\{+, -, \times, \div, sin, \sqrt{}, ln, \frac{1}{x}\}$

Table 11: Configuration of the TGP system for the comparison of LGP and TGP.

Parameter	**Value**
Initial Tree Depth	7 – 14
Initialisation Method	Ramped Half and Half
Constants	$\{-1, 0, 1\}$
Constant Terminals	40%
Population Size	500
Generations	100
Tournament Size	4
Crossover Rate	50%
Sub-tree Mutation Rate	25%
Hoist Mutation Rate	37.5%
Point Mutation Rate	37.5%
Function Set	$\{+, -, \times, \div, sin, \sqrt{}, ln, \frac{1}{x}\}$

Table 12: Configuration of the LGP system for the comparison of LGP and Linear Regression.

Parameter	**Value**
Initial Min. Program Length	30
Initial Max. Program Length	100
Min. Program Length	50
Max. Program Length	200
Constants	$[0-9]$
Constant Rate	40%
# Calculation Registers	8
Population Size	1000
Generations	500
Micro-Mutation Rate	65%
Macro-Mutation Rate	35%
Number of Offspring	6
Tournament Size	6
Crossover Rate	50%
Micro-Mutation Insertion Rate	50%
Macro-Mutation Insertion Rate	50%
Register Mutation Rate	33.3%
Operation Mutation Rate	33.3%
Constant Mutation Rate	33.3%
Operation Set	$\{+, -, \times, \div, sin, x^y, \sqrt{}, ln, \frac{1}{x}, if \leq, if >\}$

5 Results

5.1 Effects of Parameter Combinations

5.1.1 Overall Best Performing Combinations

Figure 17 depicts the average fitness of the ten combinations which performed best across all of the benchmarks. The best performing overall combinations are determined based on their total average fitness. Of these combinations, the combination with the lowest *average* fitness was combination 50 which has been highlighted with a dashed black line. Diamond markers show the combination with minimum average fitness for each particular benchmark, with the colour of the marker indicating the combination as described by the legend.

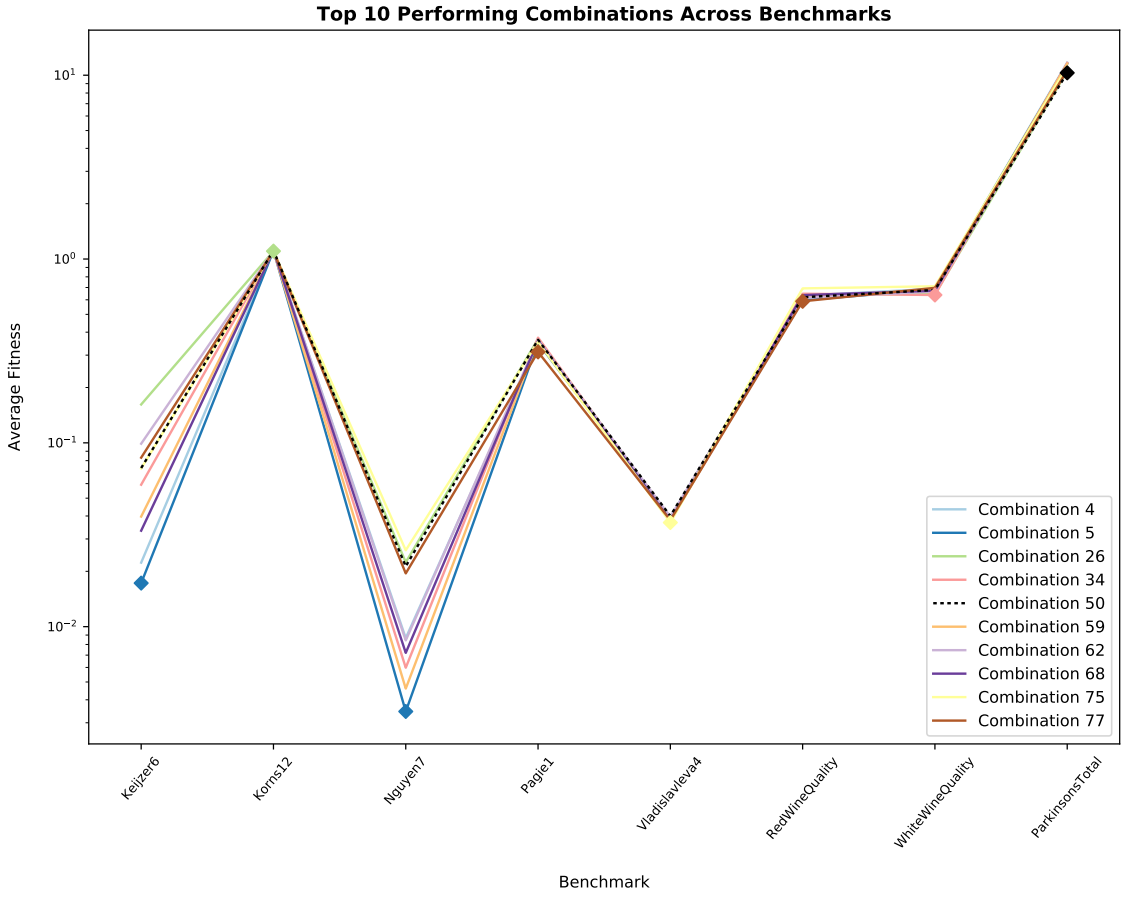


Figure 17: Illustration of combinations that performed best on average across the benchmarks. Here, individual lines represent the average fitness of a top-performing combination on a per-benchmark basis. The top-performing combinations are determined based on their total average fitness. Combination 50 is identified as the combination that performs best *on average* across the benchmarks. Markers indicate the combination with minimum average fitness per benchmark.

The results for the Keijzer-6 and Nguyen-7 problems interestingly show completely different characteristics than those of the other problems. A greater amount of variation in the average fitness between the combinations for these problems is apparent. This phenomena most likely arises due to the lower complexity of these problems, resulting in a favouring of more restrictive/conservative combinations of parameters (combination 5 for example). Additionally, over the course of the EA, it appears that the harder problems all begin to converge on an optimum solution and as a result, the differences between combinations have less of an effect on performance, in terms of fitness.

A summary of the individual parameter settings for the overall top ten performing combinations as illustrated above is given in Table 13. Amongst these combinations, it can be noted that no combination makes use of a restrictive operation set configuration. The implication is that for the particular benchmark problems tested, the operation set is vital, and as such an operation set consisting only of simple arithmetic operations negatively impacts the performance of the system.

Similarly, it appears the problems have a preference towards combinations which favour exploitation over exploration. Only one of the top performing combinations utilised a more exploratory search, providing the best result for the Vladislavleva-4 problem. This could be due to the system attempting to avoid the destructive nature of the macro-mutation operator (as used by an exploratory search).

Table 13: Summary of the parameter settings used by the top-performing combinations across the benchmarks as identified in Figure 17.

Combination	Program Length	# Calculation Registers	Operation Set	Micro/Macro Mutation Rate
4	Restrictive	Restrictive	Conservative	Balanced
5	Restrictive	Restrictive	Conservative	Exploitation
26	Restrictive	Generous	Generous	Exploitation
34	Conservative	Restrictive	Generous	Balanced
50	**Conservative**	**Generous**	**Conservative**	**Exploitation**
59	Generous	Restrictive	Conservative	Exploitation
62	Generous	Restrictive	Generous	Exploitation
68	Generous	Conservative	Conservative	Exploitation
75	Generous	Generous	Conservative	Exploration
77	Generous	Generous	Conservative	Exploitation

5.1.2 Combination Frequencies

Figures 18 and 19 illustrate the frequency with which a particular combination occurred in the top and bottom twenty performing combinations, respectively. Any trends to the combinations that perform best or worst are difficult to interpret in this visualisation (Section 5.1.3 investigates per-benchmark trends to form a deeper understanding of these results). Broadly speaking though, it can be gathered that there is a definite relationship between fitness performance and configuration. Notably, certain combinations consistently perform satisfactorily or poorly — for example, combinations 0, 1, 3 frequently occur in the bottom performing set. Conversely, combinations 5, 26, 77 are regularly present in the top performing group. This suggests that some preference towards particular combinations is evident.

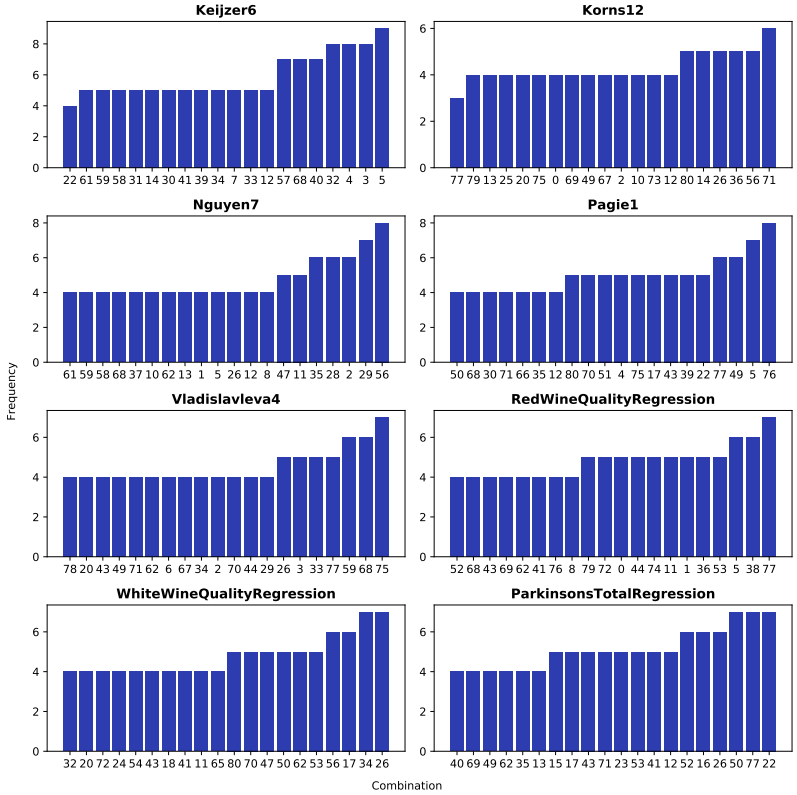


Figure 18: Frequencies of combinations in the top twenty results. Combinations are organised in sequential order, numbered from 0 to 80, with 0 being the most restrictive and 80 the most generous.

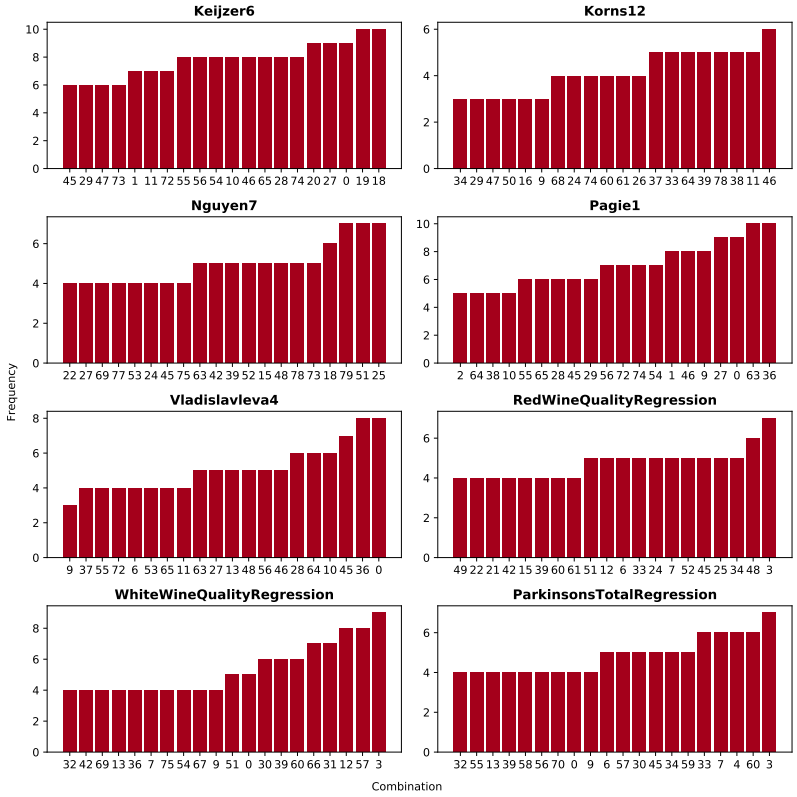


Figure 19: Frequencies of combinations in the bottom twenty results. The organisation is the same as for Figure 18.

5.1.3 Combination Frequency Trends

In this section, the results for each benchmark are investigated individually to complement the broader, high-level analysis given in Sections 5.1.1 and 5.1.2. Three main visualisation techniques are used to examine the results: (1) one-dimensional heat maps to show the relationship between fitness and a particular parameter setting where appropriate, (2) box-plots to measure the distribution of average fitness with respect to parameter settings, and (3) scatter plots to illustrate the position of combinations in terms of the combination space.

Keijzer-6

Figure 20 shows the combinations in sequential order against their average fitness value. When arranged in a sequential order, the combinations represent a spectrum where the left-most combinations (starting at 0) are the most restrictive and the right-most combinations (starting at 80) are the most generous.

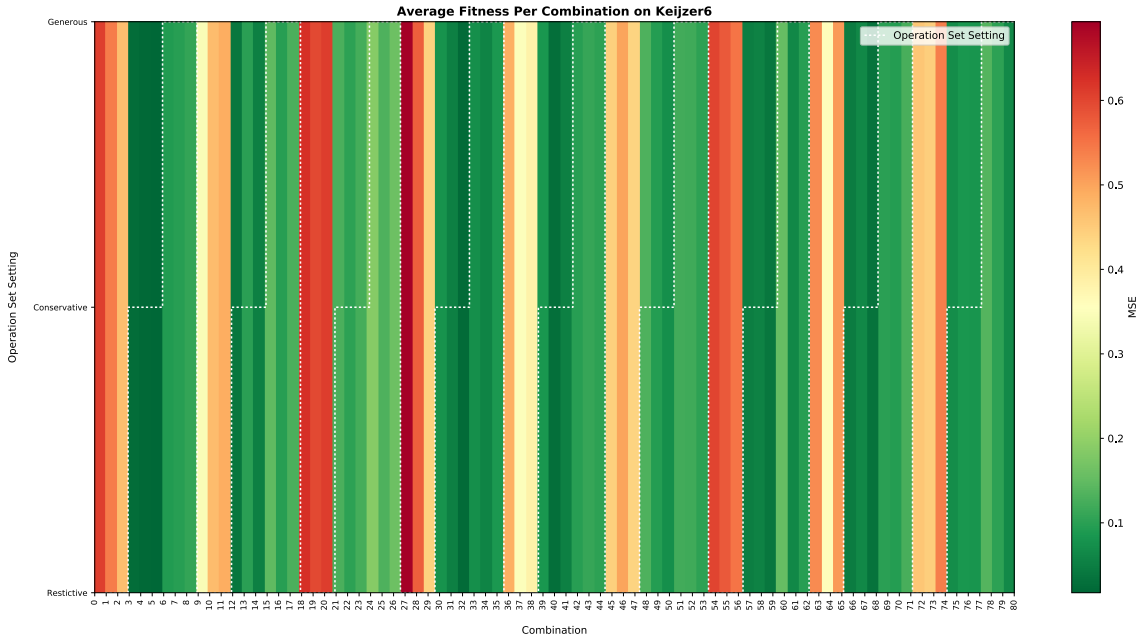


Figure 20: Individual combination performance on the Keijzer-6 benchmark. The colour of the bars is a function of average fitness. A white overlay line highlights the interaction between the operation set setting and the combination performance.

The problem's heat map demonstrates a distinct pattern: a sequence of three poorly performing combinations, followed by six well performing combinations. This pattern occurs as a direct function of the operation set setting, given on the y-axis. A reasonably clear relationship is observed — where a restrictive operation set is used as part of a combination, the average fitness is considerably worse than when a conservative or generous operation set is applied.

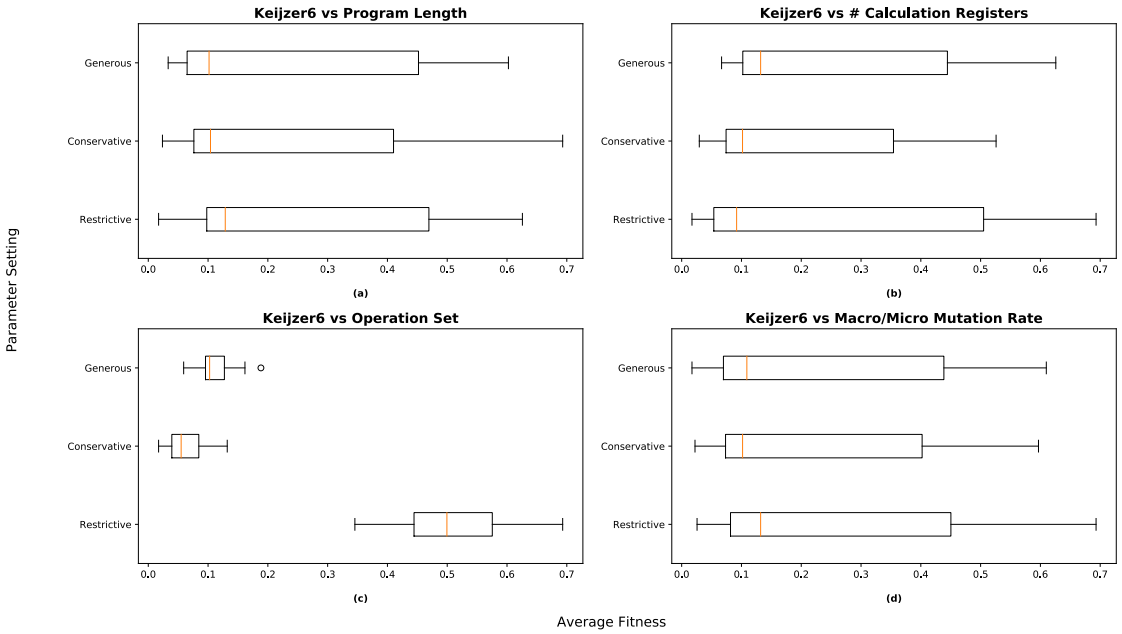


Figure 21: An illustration of the distribution of average fitness as a function of combination parameter setting. Each plot focuses on a a particular parameter of the combinations and highlights the distribution of average fitness for each setting that parameter takes.

As a secondary assessment, Figure 21 explores the distribution of average fitness with respect to particular parameter settings for the Keijzer-6 problem. The range of average fitness with respect to program length, number of calculation registers, and mutation rates is large with a clear positive skew (see Figure 21a-c). While this demonstrates relatively good performance with a high concentration of results around the median, it also illustrates a large amount of variance above the median. With respect to the operation set however, there is a clear relationship between fitness and the particular operation set value (e.g. Figure 21d). The conservative and generous settings produce a lower average fitness, with the entire range being tightly concentrated around a comparatively low median value, confirming the interpretation of the pattern observed in Figure 20.

Korns-12

The results for Korns-12 are more difficult to interpret through a 1-D heat map in contrast to the Keijzer-6 problem, and has been omitted. This is most likely due to different combinations showing less variation in fitness performance. Figure 22b shows that a conservative calculation registers setting produces solutions with fitness below the lower quartile of other settings. Similarly, a generous program length or operation set promotes the discovery of better solutions (shown in Figure 22a,c).

Figure 23 positions the parameter combinations in terms of their level of restrictiveness or generosity, with the size of the points indicating the error level (i.e. smaller is better). It becomes quite clear that the distribution of well-performing combinations is fairly evenly spread between the restrictive and generous sides of the spectrum.

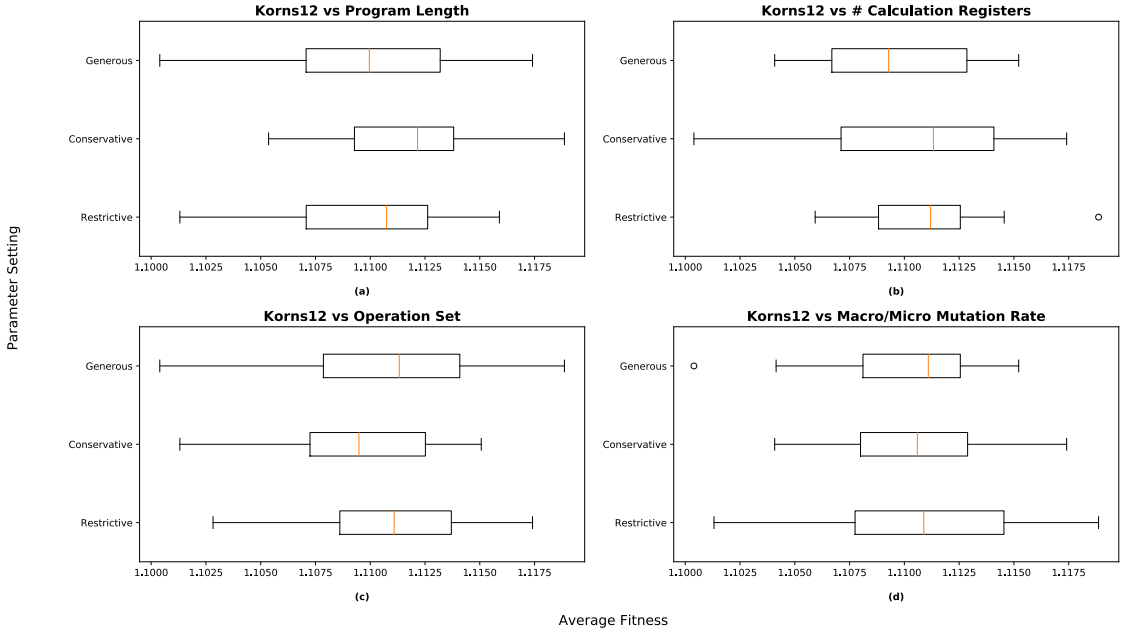


Figure 22: Distribution of average fitness as a function of combination parameter type. There is a less obvious trend in these results in comparison to the Keijzer-6 problem, which could be a result of the difficulty difference between the two problems. This implies that difficult problems require more fine-grained tuning of parameters than is encouraged by the combinations used here.

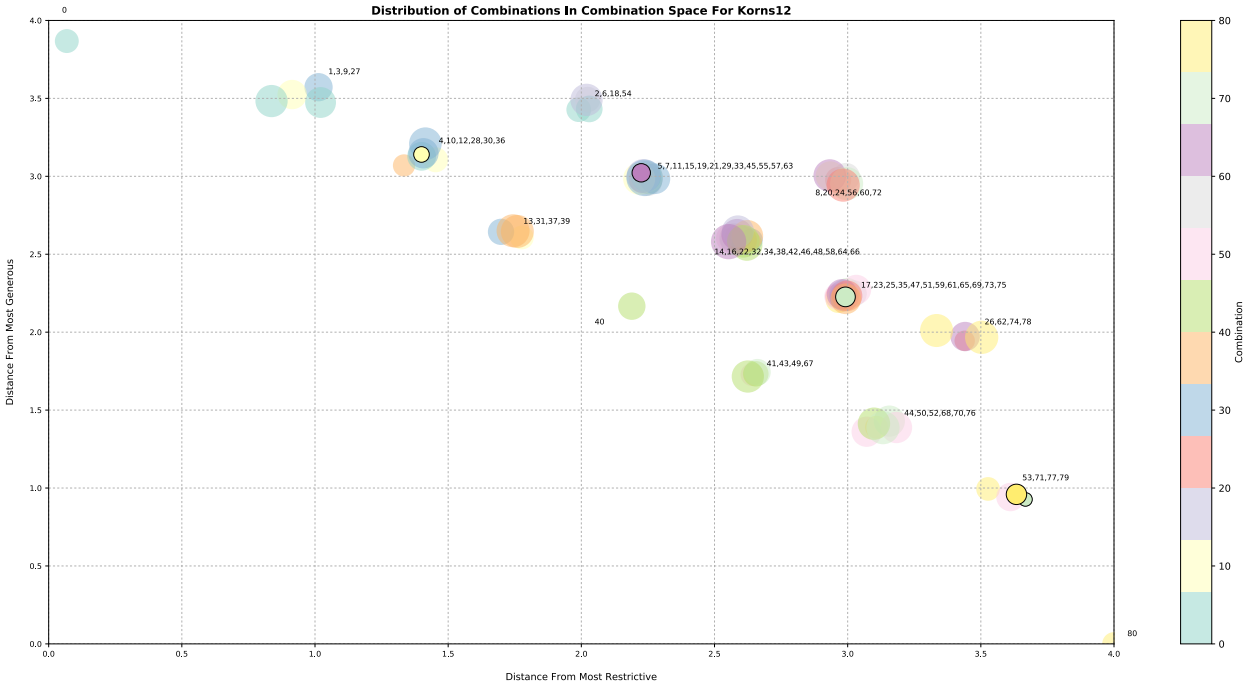


Figure 23: Illustration of the distribution of the different combinations in combination space. The combinations are positioned with respect to their distance from the most restrictive and most generous combinations. The size of individual points is based on the level of error that combination produced. The top five combinations are emphasised with a black outline.

A summary of the individual parameter configurations is provided in Table 14, for the top five performing combinations (indicated by points with black outlines in Figure 23). Here, some trends become more obvious.

Combinations 12, 63, 69

Combinations 12 and 63 are located closer to the restrictive portion of the spectrum, while combination 69 is a relatively conservative setting. These combinations all favour an exploratory search which may allow them to find more complex solutions and better performing solutions, despite their restrictions.

Combinations 71, 79

These combinations are situated in the most generous section of the spectrum, and interestingly utilise a more reserved balanced/exploitative search. This suggests that initial solutions from these combinations perform quite well and allow the system to easily hone in on a well-performing solution.

Of these five combinations, combination 71 produced the best result perhaps acting as a middle-ground between combinations 12, 63, 69 and combination 79.

Table 14: Top five performing combinations for the Korns-12 benchmark.

Combination	Program Length	# Calculation Registers	Operation Set	Micro/Macro Mutation Rate
12	Restrictive	Conservative	Conservative	Exploration
63	Generous	Conservative	Restrictive	Exploration
69	Generous	Conservative	Generous	Exploration
71	Generous	Conservative	Generous	Exploitation
79	Generous	Generous	Generous	Balanced

Nguyen-7

Nguyen-7 performs optimally in cases where a restrictive number of calculation registers and a restrictive/conservative operation set are applied. Combined with an exploitative search, the top five combinations employ configurations with a tendency towards restrictive parameters. In contrast, combination 61 provides programs with greater complexity due to a generous program length and operation set, but appears to be countered by a balanced mutation rate. The rest of the top performing combinations are situated in close relational proximity to each other (see Figure 25). It should be noted that a small amount of jitter is added to the points to prevent overlap.

Figure 24 further illustrates the bias of well-performing combinations towards restrictive combinations, especially with regards to number of calculation registers and operation set. Combinations with a restrictive number of calculation registers and operation set showed less range in terms of fitness, with a majority of solutions situated close to the left-skewed median value (see Figure 24b,c).

Furthermore, better overall performance is given by the generous macro/micro-mutation rate setting (exploitative search) in comparison to a conservative/restrictive setting. A majority of solutions are situated around the median, which is comparatively low (given in Figure 24d).

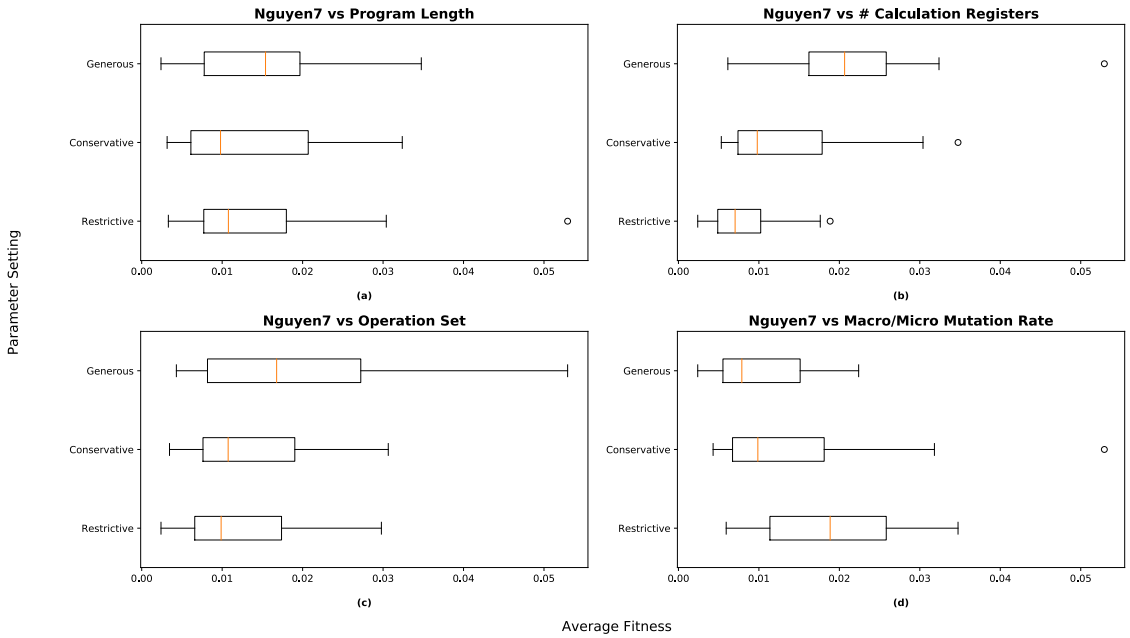


Figure 24: Distribution of average fitness as a function of combination parameter type. Nguyen-7 shows an inclination towards restrictive combinations — notably those with a restrictive number of calculation registers and operation set, especially in conjunction with an exploitative search.

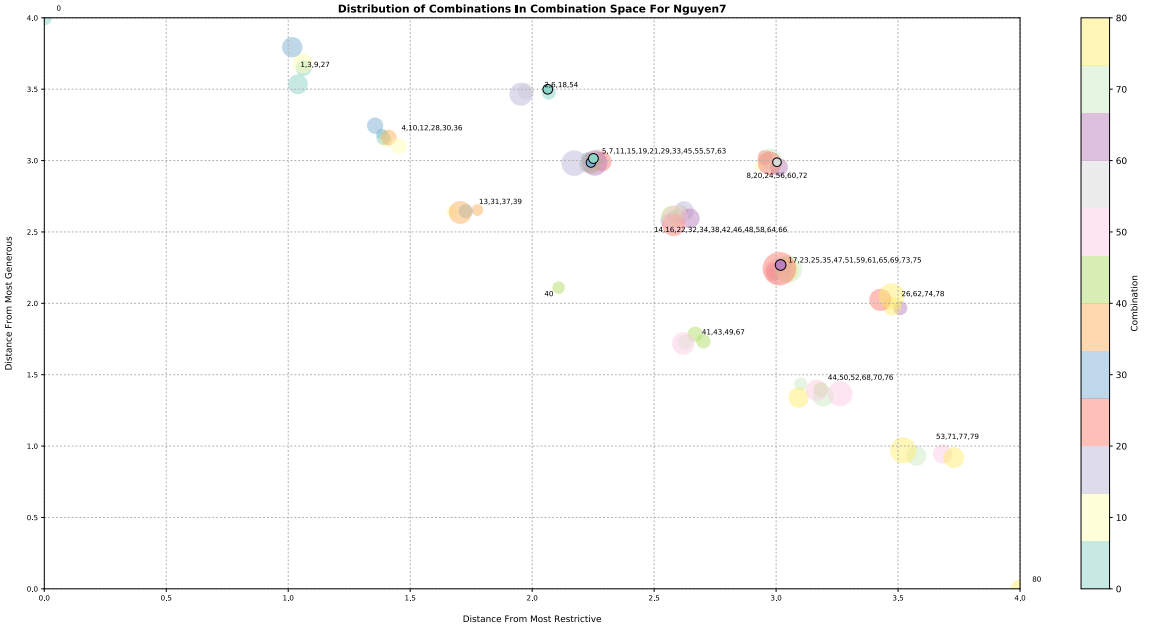


Figure 25: Illustration of the distribution of the different combinations in combination space for the Nguyen-7 problem. The top five performing combinations are highlighted and mainly reside in the conservative area (centre) of the spectrum.

Table 15: Top five performing combinations for the Nguyen-7 benchmark.

Combination	Program Length	# Calculation Registers	Operation Set	Micro/Macro Mutation Rate
2	Restrictive	Restrictive	Restrictive	Exploitation
5	Restrictive	Restrictive	Conservative	Exploitation
29	Conservative	Restrictive	Restrictive	Exploitation
56	Generous	Restrictive	Restrictive	Exploitation
61	Generous	Restrictive	Generous	Balanced

Pagie-1

Figure 26 shows the relationship between fitness and the four different parameters targeted. There are two clear trends to notice in this visualisation. Firstly, a generous setting of program length and calculation registers enables the system to find better solutions (e.g. Figure 26a,b). While the median is roughly equivalent between settings, the lower quartile and minimum of the generous setting for program length and calculation registers are considerably lower than for other settings. Figure 27 supports these observations as the top five performing combinations fall on the generous side of the spectrum.

Secondly, in terms of operation set there is an indication that a conservative/generous setting performs best, whereas a restrictive setting drastically hinders the fitness performance of the system. In Figure 28, this relationship between operation set and average fitness becomes more apparent; average fitness deteriorates significantly where a restrictive setting is utilised. Also of considerable interest is how a conservative setting tends to have a slight edge over a generous setting, perhaps relating to the expansion of the search space imposed by a larger operation set.

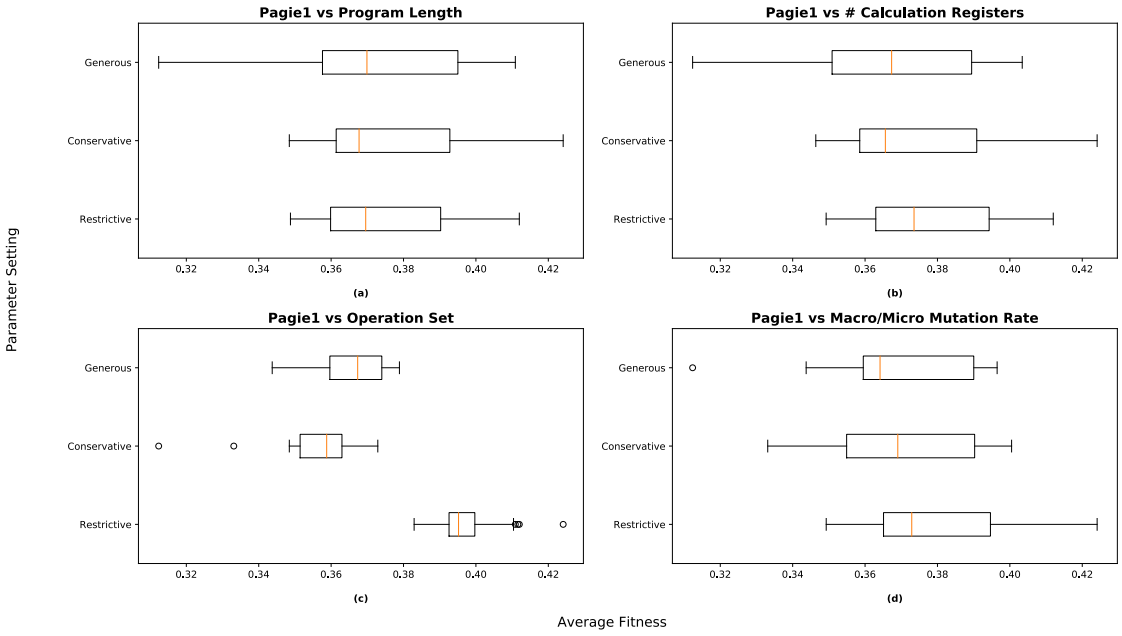


Figure 26: Distribution of average fitness as a function of combination parameter type. Notably, a generous program length and calculation register setting allows for better solutions to be found during the evolutionary process.

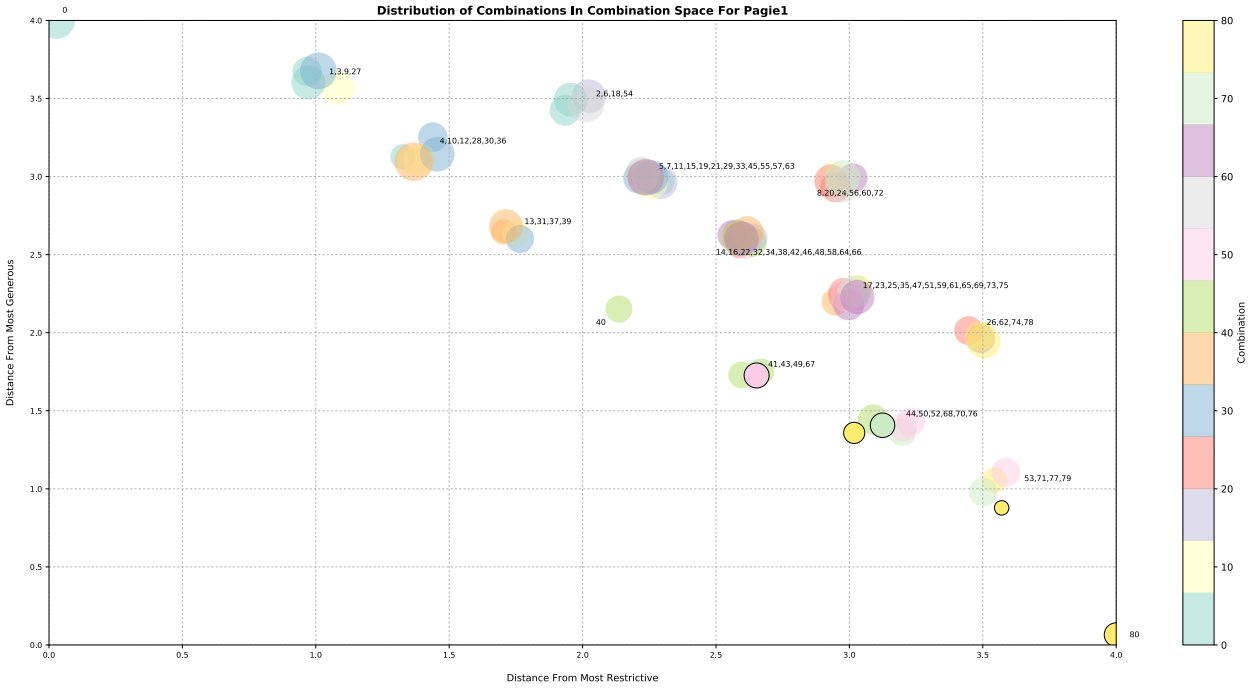


Figure 27: Illustration of the distribution of the different combinations in combination space for the Pagie-1 problem. The top five performing combinations are highlighted and gravitate towards the generous side of the spectrum, which may be a result of the problems relative difficulty.

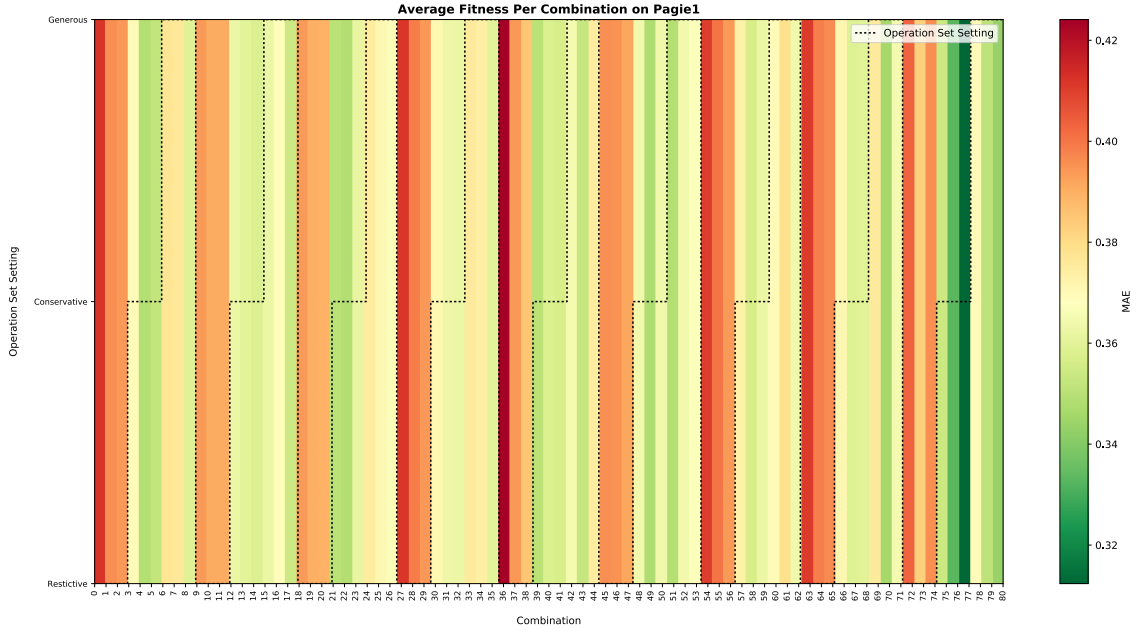


Figure 28: Individual combination performance on the Pagie-1 benchmark. The colour of the bars is a function of the fitness with the particular range defined by the colour bar on the left of the plot. A black dashed overlay line highlights the interaction between the operation set setting and the combination performance.

Vladislavleva-4

Vladislavleva-4 shows less variation in terms of average fitness suggesting that the system narrows down on a set of relatively well-performing solutions with relative ease, which may be due to the settings causing a focus on local optima.

Generally, Vladislavleva-4 is able to find slightly better solutions with a restrictive mutation rate setting (shown in Figure 29d); a restrictive setting specifies a more exploratory search through a bias towards macro-mutations. In contrast to other benchmarks, Vladislavleva-4 is the only problem to favour the more destructive macro-mutations, possibly influenced by the system's aforementioned tendency to converge on local optima. Macro-mutations drastically alter the function of the program and a higher probability of such operations could potentially aid in escaping localised areas of the solution space.

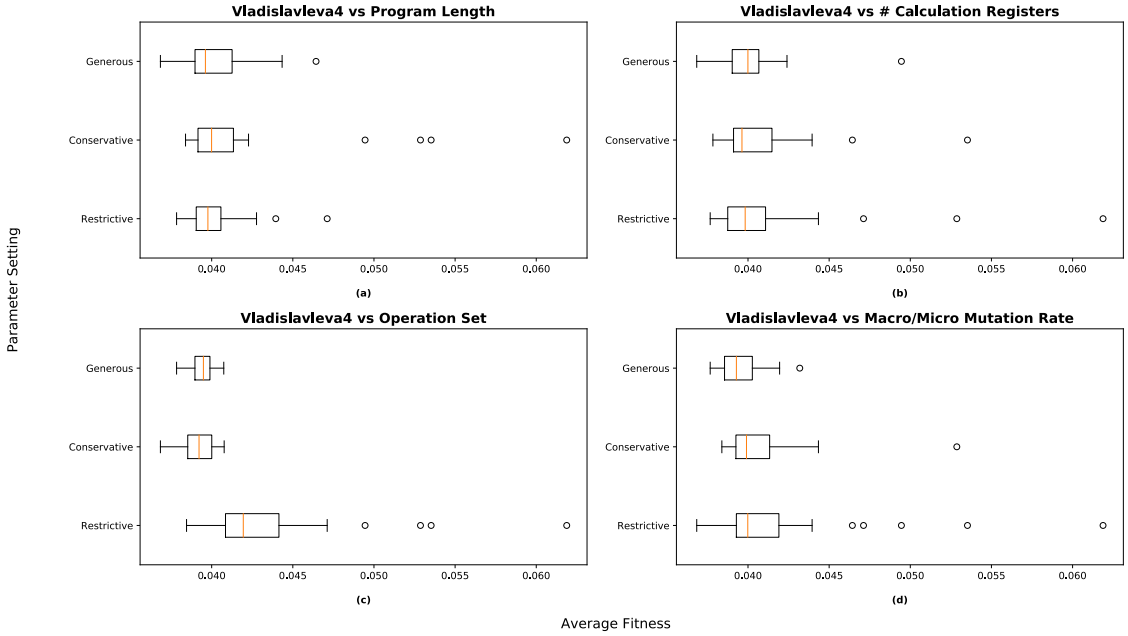


Figure 29: Distribution of average fitness as a function of combination parameter type. Vladislavleva-4 shows less variation as a result of the effects of program length and number of calculation registers.

Other parameters demonstrate a similar trend to that of previous benchmarks: a generous program length and number of calculation registers has a minor advantage over other settings, allowing for some degree of lower fitness solutions to be unearthed (as per Figure 29a,b). A conservative operation set performs slightly better than a generous setting, but perhaps more interesting is the way in which a restrictive operation set is detrimental in terms of fitness (see Figure 29c). This observation aligns with those for some of the other benchmarks, further emphasising the importance of an operation set in allowing better solutions to be found.

Figure 30 illustrates a majority of well-performing combinations on the generous side of the spectrum; likewise, a collection of combinations that produce a high level of error are evident on the restrictive side.

Figure 30: Illustration of the distribution of the different combinations in combination space for the Vladislavleva-4 problem. The top five performing combinations are highlighted and sit somewhat further into the conservative/generous portion.

Red Wine Quality

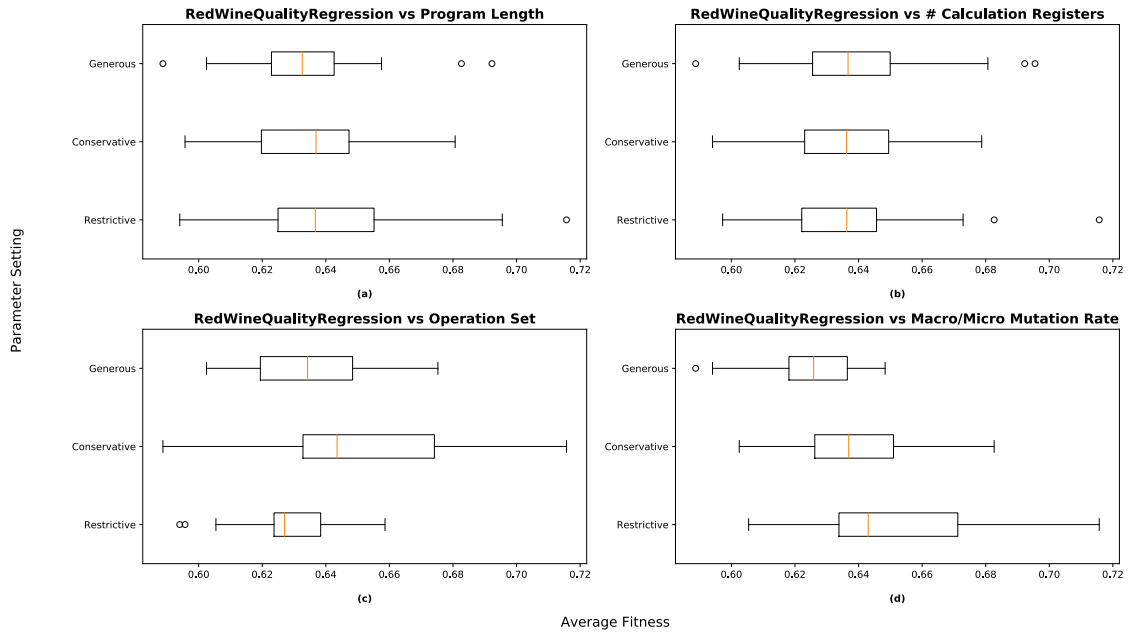


Figure 31: Distribution of average fitness as a function of combination parameter type. The results for the Red Wine Quality benchmark are less obvious and more difficult to interpret. A more generous program length appears to demonstrate less variation in average fitness.

The results for this benchmark are more difficult to interpret, as there is a higher degree of alignment between the various settings (restrictive, conservative, generous). This suggests that the difference between combinations has little effect on overall performance, potentially alluding to a necessity for low-level tuning of the parameters.

Despite this, there are some small observations to be made. As is true for the majority of the other benchmarks, there is an overall preference towards an exploitative search (refer to Figure 31d). Additionally, there is little evidence of an effect of parameter setting for program length or number of calculation registers on average fitness; however, as the program length becomes more generous, the range of average fitness decreases (Figure 31a,b).

An interesting results is present in terms of operation set (see Figure 31c). A conservative operation set allows for the discovery of some more ideal solutions, but also demonstrates worse average fitness performance due to an increased discovery rate for poorly performing solutions too.

White Wine Quality

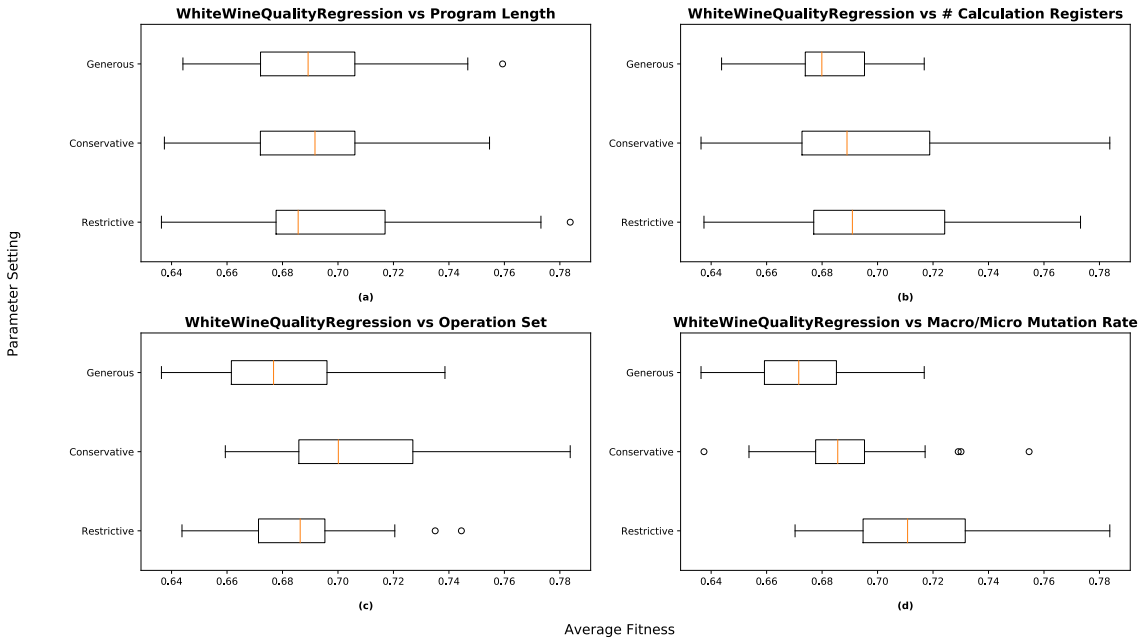


Figure 32: Distribution of average fitness as a function of combination parameter type. The White Wine Quality benchmark has a story story similar to the Red Wine Quality benchmarks, with the exception of a conservative operation set providing worse performance.

A similar story can be told for the white wine quality benchmark, but some trends are clearer. Like most other benchmarks, an exploitative search provides the best results (see Figure 32d). The number of calculation registers creates a more pronounced effect on average fitness, with a generous setting providing slightly better average

fitness. That being said, other settings allow for some lower fitness solutions to be found. In contrast to the red wine quality benchmark, a conservative operation set does not produce any benefits, instead having an overall negative effect on fitness.

Parkinson's Total

The most interesting trend shown by the Parkinson's total regression problem is in relation to the number of calculation registers (shown in Figure 33b). As the number of calculation registers increases, the overall range of average fitness decreases. This kind of relationship is displayed somewhat by other benchmarks, but not in such a clear way. A continuation from this could involve investigating if there is a point of diminishing returns as the number of calculation registers increases.

In comparison to the other benchmarks, the results show some parallels; an exploitative search shows superior results on average and a restrictive operation set has a negative impact on the overall fitness of solutions (Figure 33c,d).

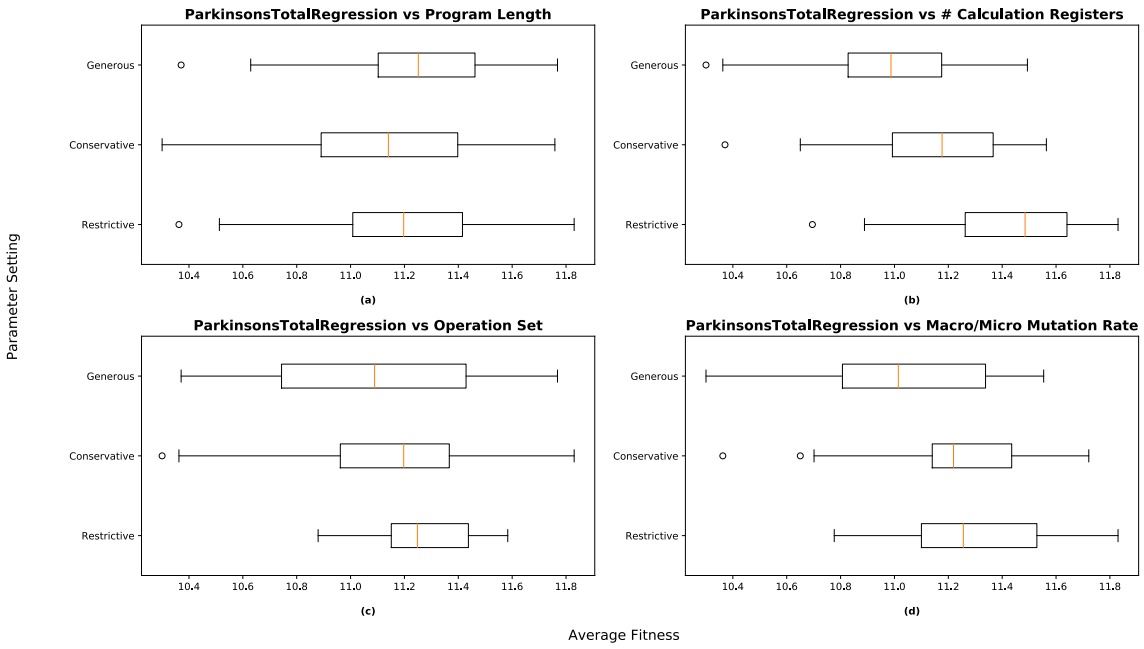


Figure 33: Parkinson's Total regression shows an interesting relationship between the number of calculation registers and overall fitness. As the number of calculation registers increases, the average fitness decreases.

Figure 33a shows some evidence that a conservative program length has a minor advantage over other settings. It is likely that this is to account for the preferred complexity of a more generous number of calculation registers; that is, a too generous program length may begin to negate the benefits of additional calculation registers.

It should be noted that scatter plots for the last three benchmarks have been omitted as they don't provide any unique insight into particular preferences.

5.2 Evolutionary Algorithm Comparison

5.2.1 Parameters

Parameter combinations used to configure the system for each benchmark are taken from the best performing combinations identified in the first round of experiments. It should be noted that these combinations do not necessarily provide the best achievable results, only the best results amongst the combinations evaluated in the first round. For clarity, the parameters used are summarised in Table 16.

Table 16: Parameter combinations used to configure the system for the EA comparison. The combinations chosen are those that performed best on each problem in the first round of experiments.

Problem	Program Length	# Calculation Registers	Operation Set	Micro/Macro Mutation Rate
Keijzer-6	Restrictive	Restrictive	Conservative	Exploitation
Korns-12	Generous	Conservative	Generous	Exploitation
Nguyen-7	Generous	Restrictive	Restrictive	Exploitation
Pagie-1	Generous	Generous	Conservative	Exploitation
Vladislavleva-4	Generous	Generous	Conservative	Exploration
Red Wine Quality	Generous	Generous	Conservative	Exploitation
White Wine Quality	Restrictive	Conservative	Generous	Exploitation
Parkinsons Total	Conservative	Generous	Conservative	Exploitation

5.2.2 Fitness Comparison

Figure 34 shows the average fitness result of each EA per benchmark. Here, the island-migration EA outperformed the other techniques in the majority of cases. The exceptions are the Keijzer-6 and Nguyen-7 benchmarks, which show better results under the steady-state and master-slave algorithms than the island-migration technique. This is likely an artefact of the simplicity of these problems in comparison to the other benchmarks, as was noted in Section 4.1.1.

The island-migration approach is typically utilised for its ability to increase the diversity level of an EAs population (Alba et al., 2013). In the case of the Keijzer-6 and Nguyen-7 problems, it appears that the system's ability to narrow its search towards a particular solution is damaged by the increase in diversity. This suggests that the island-migration technique should be utilised in situations where the EA tends to converge early on solutions which may not necessarily be the optimum.

Pagie-1 gains the most notable fitness performance increase from the island-migration implementation with a decrease of approximately 25% in average fitness in comparison to the other techniques. This is likely attributed to the increased diversity level as described later in Section 5.2.4.

The steady-state and master-slave techniques perform comparably which is an expected result, as the core of these two algorithms is essentially the same — the only difference being the addition of parallel processing which does not appear to have any impact on fitness performance.

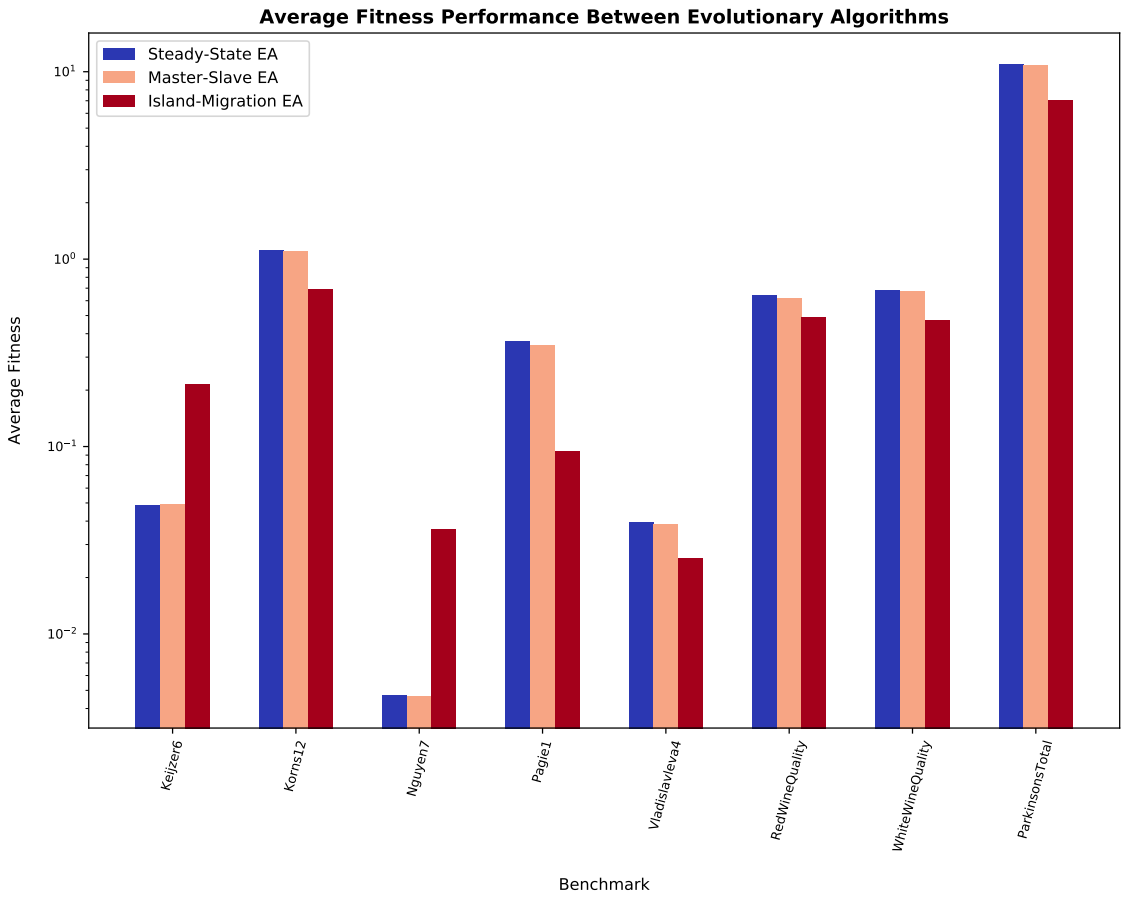


Figure 34: Average Fitness Performance of the three EAs. Fitness is averaged over 30 independent runs for each benchmark and categorised by the particular EA implementation used. For all benchmarks except the Keijzer-6 and Nguyen-7 problems, the island-migration EA has a slight advantage over the other implementations.

5.2.3 Runtime Comparison

Runtime is an important consideration when choosing an EA, due to the tendency of EAs to run for long periods of time — particularly if the problem requires a large amount of fitness evaluation. Figure 35 suggests that in cases where time is critical or a large amount of fitness evaluations is required, the island-migration technique should be avoided, as it has drastically greater runtime than other techniques. Of course the particular problem may gain other benefits from the island-migration technique, and thus the choice is problem dependent.

The master-slave technique provides a significant decrease in average runtime, likely due to the way it performs time-consuming tasks (e.g. fitness evaluation) in parallel. This benefit is most noticeable in cases where problem difficulty is relatively low, as the speed increase tends to diminish on problems where the average runtime is quite high (i.e. those that are more difficult).

A possible cause for this effect could be that difficult problems tend to require more training cases, causing the overhead of creating and aggregating threads to partially outweigh the benefits of parallel processing. It should be noted however that the master-slave technique still provides better runtime performance than the other techniques across all the benchmarks tested against here.

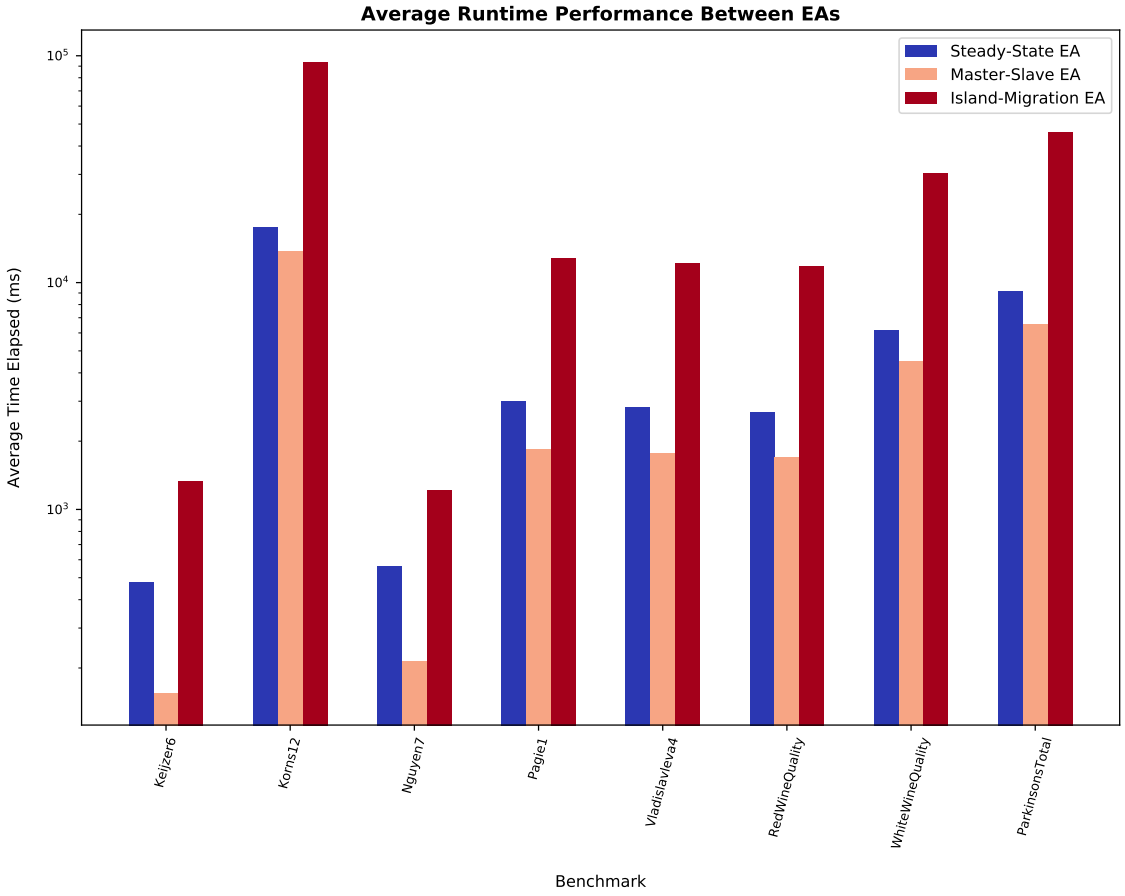


Figure 35: Average Runtime Performance of the three EAs. Runtime is measured in milliseconds and averaged over 30 independent runs for each benchmark. The master-slave EA provides a significant decrease in runtime across all benchmarks.

5.2.4 Diversity Comparison

The average diversity level for each benchmark is illustrated in Figure 36. The steady-state and master-slave techniques perform comparably with respect to diversity — an unsurprising result when considering that the core algorithm is fundamentally the same between the two techniques.

On the other hand, the island-migration technique demonstrates an increased level of diversity across all of the benchmarks tested. The Pagie-1 problem displays a significantly higher level of diversity in comparison to the other benchmark problems,

which is believed to be due to the usage of a particularly generous combination of parameters to configure the system.

The combination that performed best for Pagie-1 in the initial round of experiments specified a generous program length and number of calculation registers, with a conservative operation set. This seems to encourage the island-migration technique to continue growing programs with a high level of variability. This is reflected in the fitness performance as described previously, with Pagie-1 performing considerably better under the island-migration EA implementation.

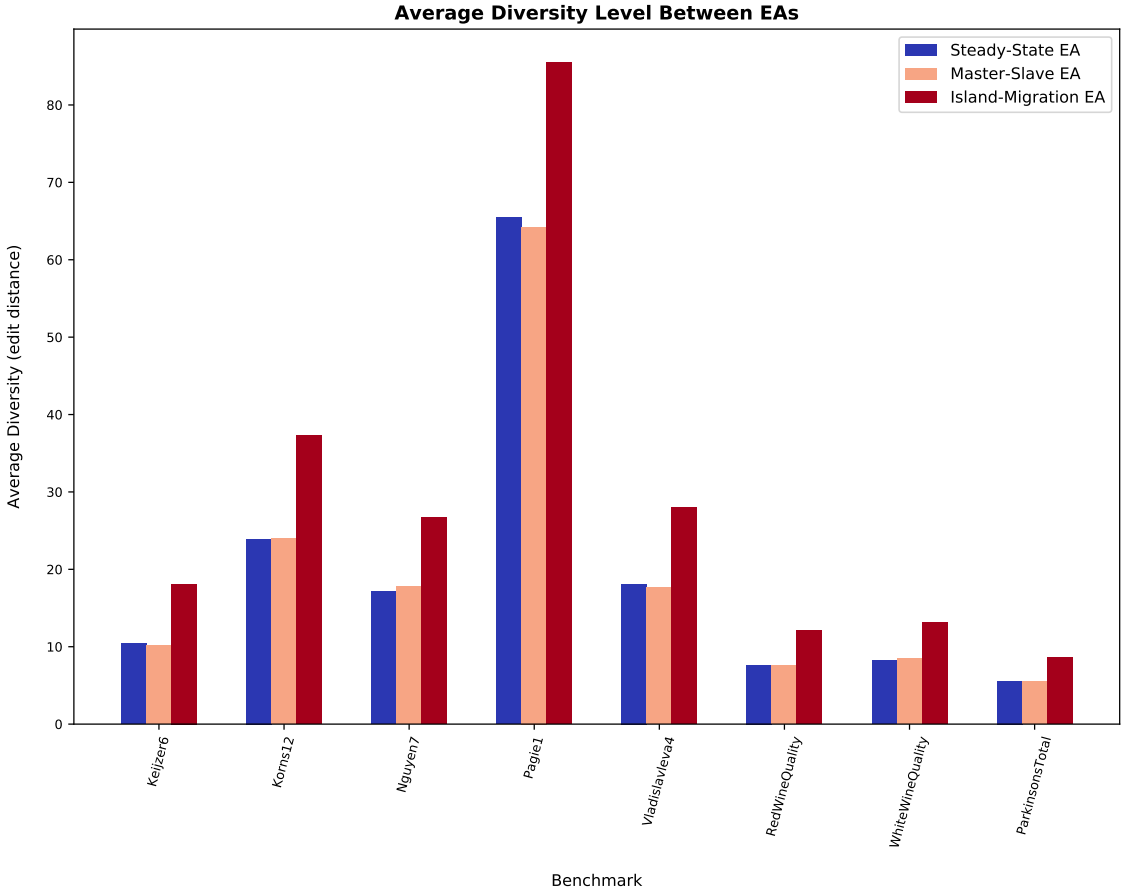


Figure 36: Average Diversity Level of the three EAs. Diversity is measured in terms of edit distance to produce an average level of diversity. Edit distance is measured in units of number of instructions. Here, the island-migration EA results in the most diverse populations.

5.3 Comparison to Tree-based GP and Linear Regression

5.3.1 Tree-based GP

Figure 37 provides a comparison of the fitness between the LGP implementation and a tree-based GP (TGP) solution. Overall, the two systems perform comparatively; likely a result of similar configurations. LGP slightly outperforms TGP on the Korns-12 benchmark. There is no clear reason as to why this may be, and as the difference in fitness is minor there is no warrant for further investigation.

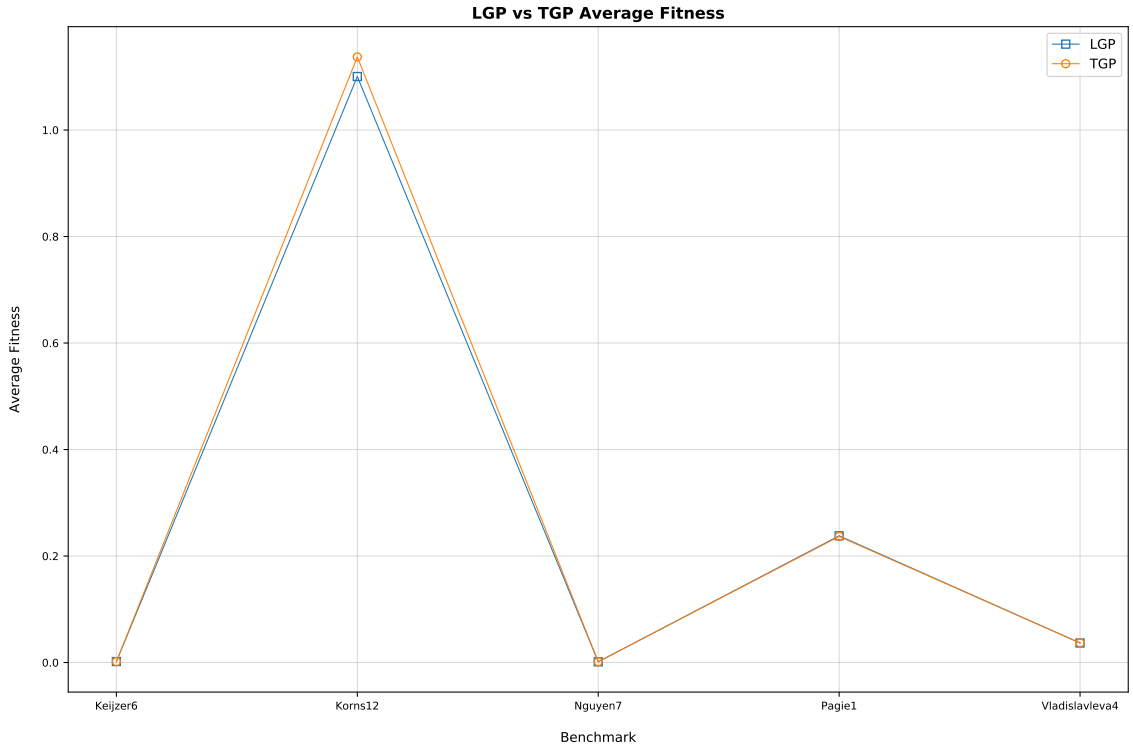


Figure 37: Average fitness of LGP and TGP on the five synthetic benchmark problems. Performance is essentially comparable between the two techniques with regards to fitness.

There are no inferences that can be made regarding whether LGP performs better on any of the synthetic benchmark problems as the performance between the techniques is so closely related. Despite this, commentary can be made about the LGP implementation. The results demonstrate that the system's implementation is at least as correct as the TGP based system and can perform comparatively when utilised in similar contexts.

A more enlightening implication can be gathered from Figure 38 which compares the average program length between LGP and TGP. LGP demonstrates an ability to find more succinct solutions in most cases. As described previously, the two systems perform comparably in terms of fitness, meaning that TGP requires more complex

programs to produce similar quality results. Generally, a simpler solution is preferred as it facilitates easier interpretation of the underlying model and integration into other contexts (Gandomi et al., 2014).

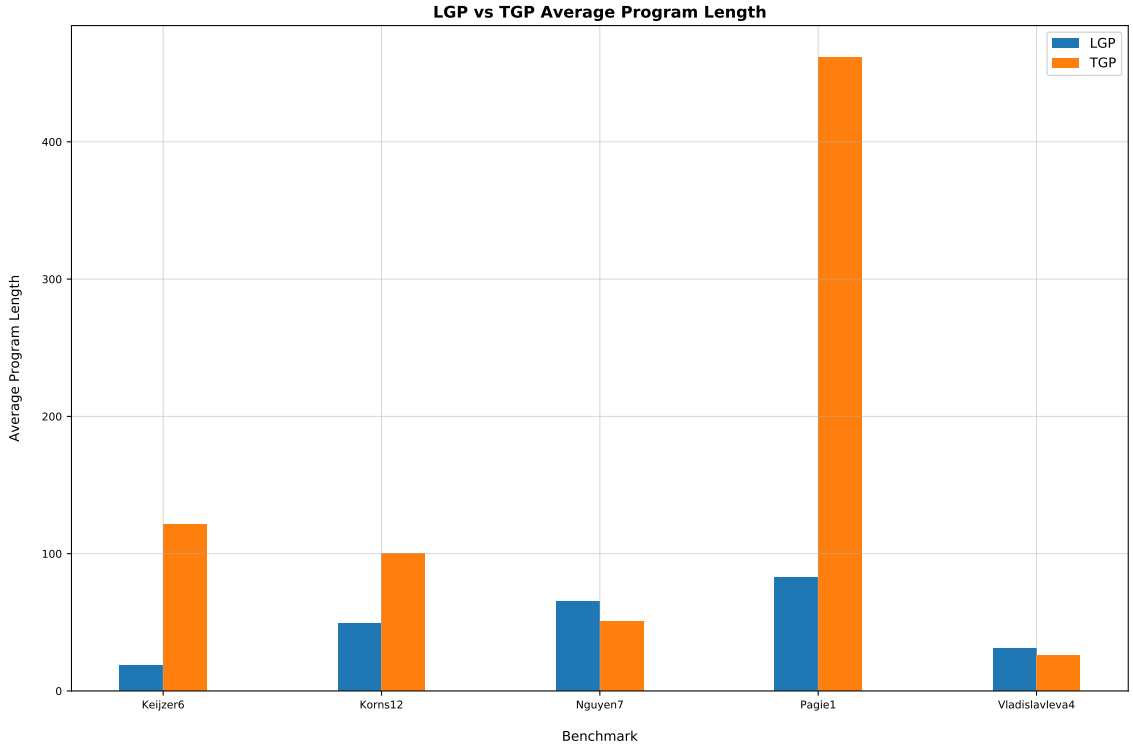


Figure 38: Comparison of average program length between LGP and TGP. Here, LGP shows a relatively clear advantage; TGP generally has a much greater average program length than LGP, despite performing comparably in terms of fitness.

TGP shows lower average program length on the Nguyen7 and Vladislavleva4 problems, but no identifiable cause for this behaviour could be determined. One suggestion is that TGP manages to converge on a good solution more rapidly than LGP; such a solution's genetic material could spread through the population and bring the average program length down. Alternatively, these results could correspond to the inherent randomness of the tests, as the difference is rather minor.

Also of note is the disparity in average program length between LGP and TGP on the Pagie1 problem. The average program length produced by TGP is roughly five times greater than that of LGP. Pagie & Hogeweg (1997) mention the problems susceptibility to bloat when using standard GP, however, this does not appear to hold true for LGP. Brameier (2004) argue that the compactness of linear programs may be associated to the multiple usage of registers and an implicit parsimony pressure caused by structurally non-effective code. Although the solutions generated by LGP are more compact, quality of the solutions in terms of fitness is equivalent to TGP, suggesting that LGP still encounters difficulty in approximating the function.

5.3.2 Linear Regression

A summary of the results is depicted in Figure 39. Broadly speaking, the techniques are comparable on these particular problems. LGP performs slightly better on the white wine quality and parkinson's total problems, but is generally comparable. LGP performs ideally on problems where the target is a non-linear function of the inputs (i.e. linear regression performs poorly), as demonstrated by the results on the gene expression survival problem. This problem is highly non-linear, and as a result LGP produces significantly more ideal solutions. These findings suggest that while LGP can perform comparably to linear regression in the context of certain problems, its advantages are more pronounced with non-linear problems.

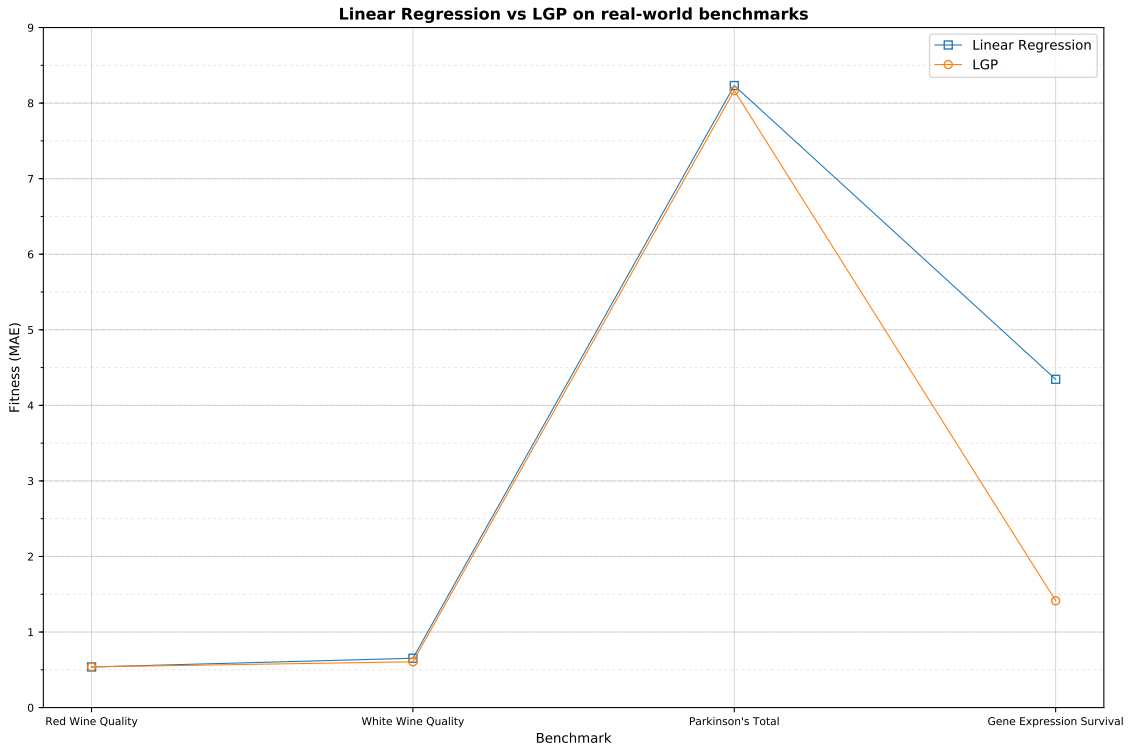


Figure 39: Comparison of mean-absolute error between the LGP and Linear Regression techniques, with respect to the real-world regression benchmarks. Performance is comparable between techniques on the wine quality benchmarks, but linear regression shows an advantage on the parkinson's data set. LGP significantly outperforms linear regression on the gene expression survival data set.

5.4 Summary

From the three experiments conducted and their respective results, a set of conclusions have been previously described. Here, those conclusions are consolidated and summarised.

Firstly, the system produced competitive results for all benchmarks tested against, as well as showing comparable performance to both TGP and a standard linear regression model. This validation indicates the system's implementation is correct, robust and well-suited for solving these sorts of problems.

Secondly, a demonstration of varying system configurations highlights implications as to the reconfigurability of the system and the effect this has on results. Overall, most of the benchmark problems had a preference towards more generous combinations of parameters. Interestingly, lower dimensional problems signal favouring of more restrictive parameters, as generous settings generally expand the size of the search space — potentially negatively impacting the quality of solutions discovered. Further work may involve implementation of a set of standard parameters that can be applied where there is knowledge with respect to the difficulty of the problem (i.e. presets).

Lastly, the application of different EAs within the context of LGP showed promising results. The traditional steady-state EA is outperformed by the master-slave EA in terms of runtime, while the island-migration technique exhibits an ability to increase the variability within the LGP population. These results suggest that some problems may benefit from non-traditional EAs, particularly if runtime or population diversity are a consideration. Furthermore, the implementation of two custom EAs shows the flexibility and extensible nature of the software — a core design principle of the system.

6 Conclusion

Within the wide landscape of open-source tools available for GP, an open-source LGP implementation is surprisingly lacking. In contrast to other techniques, LGP has two unique characteristics arising from its linear program representation: first, a graph-based functional structure resulting from the way register contents are used multiple times during computation. This structure leads to program graphs with higher variability and the development of more compact solutions. Second, the existence of non-effective instructions which do not have an influence on program output. Non-effective instructions help to guard other instructions from disruption caused by genetic operator application and enables the occurrence of neutral variations — variations that don't change the fitness of a program.

Motivated by the deficiency of an existing open-source implementation, the goal of this work was to design, implement and benchmark a completely open-source LGP system. The primary contribution is in the form of the software and is relevant to the open-source and AI communities.

The system offers cross-platform support and a modern API through the usage of the Kotlin programming language. Kotlin is built on the Java Virtual Machine meaning the system is inherently cross-platform, with the addition of modern programming language constructs. A flexible and adaptable architecture is achieved through a modular system design, allowing custom functionality to be added where required.

Essentially every component of the system has a modular definition, allowing for a great degree of flexibility.

Evaluation of the system on a set of benchmark problems demonstrated the system's correctness and performance. A mix of synthetic and real-world symbolic regression problems were chosen to provide a range of difficulty in terms of benchmarks.

Notably, the LGP system showed comparable performance to both TGP and linear regression. Interestingly, LGP showed the ability to produce more succinct solutions than TGP without a compromise in fitness performance, which is particularly important when problems are prone to bloat. Compared to linear regression, LGP performed equivalently on problems with a high degree of linearity but showed significant benefits on problems where the target is a non-linear function of the inputs.

Moreover, the system's extensibility permitted the implementation and integration of two parallelised evolutionary algorithms which showed promising performance characteristics: a master-slave technique produced significant average runtime benefits while an island-migration technique promoted a greater diversity level. The evaluation of the system exemplified its viability as a foundation to prevent others having to implement their own LGP system.

From the platform of an open-source LGP system, the options for further work are plentiful. For example, the implementation and validation of further LGP components within the context of the system, such as custom search operators or evolutionary algorithms may further advocate the usage of LGP.

Alternatively, this work did not investigate the utilisation of the system for classification problems which further work could address. Overall, the system hopes to see usage in place of other GP systems so that the LGP technique can be applied to a wider range of problem domains.

The source code and binaries have been made publicly available on GitHub[6]. It is expected that any future development of the system will occur through this repository. To supplement this, API documentation[7] and a usage guide[8] is provided for those who want to use the system. The API documentation is focused on the implementation details whereas the usage guide provides a tutorial of the concepts and how they relate to the implementation.

[6]https://github.com/JedS6391/LGP

[7]https://jeds6391.github.io/LGP/api/html/index.html

[8]http://lgp.readthedocs.io/en/latest/

7 References

Alba, E., Luque, G., & Nesmachnow, S. (2013). Parallel metaheuristics: Recent advances and new trends. *International Transactions in Operational Research*, *20*(1), 1–48. https://doi.org/10.1111/j.1475-3995.2012.00862.x

Banzhaf, W. (1993). Genetic Programming for Pedestrians. *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, 628. Retrieved from http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/GenProg_forPed.ps.Z

Basili, V. R., & Lonchamp, J. (2005). Open Source Software Development Process Modeling. *Software Process Modeling*, *10*, 29–64. https://doi.org/10.1007/0-387-24262-7_2

Bouckaert, R. R., Frank, E., Hall, M. a., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2010). WEKA—Experiences with a Java Open-Source Project. *The Journal of Machine Learning Research*, *11*, 2533–2541. Retrieved from http://dl.acm.org/citation.cfm?id=1756006.1953016

Brameier, M. (2004). *On Linear Genetic Programming* (PhD thesis). https://doi.org/10.1007/s10710-007-9036-8

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

Cortez, P., Cerdeira, A., Almeida, F., Matos, T., & Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, *47*(4), 547–553. https://doi.org/10.1016/j.dss.2009.05.016

Cramer, N. L. (1985). A representation for the Adaptive Generation of Simple Sequential Programs. In *Proceedings of an international conference on genetic algorithms and the applications* (pp. 183–187). Retrieved from http://www.cs.bham.ac.uk/$/sim$wbl/ftp/biblio/gp-html/icga85_cramer.html

Danandeh Mehr, A., Kahya, E., & Yerdelen, C. (2014). Linear genetic programming application for successive-station monthly streamflow prediction. *Computers and Geosciences*, *70*, 63–72. https://doi.org/10.1016/j.cageo.2014.04.015

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. New York, USA: John Wiley.

Fogelberg, C., & Zhang, M. (2005). VUWLGP — An ANSI C++ Linear Genetic Programming Package.

Friedberg, R. M. (1958). A learning machine: Part I. *IBM Journal of Research and*

Development, *2*(1), 2–13. https://doi.org/10.1147/rd.21.0002

Galvan-Lopez, E., & Rodriguez-Vazquez, K. (2006). The Importance of Neutral Mutations in GP. *Parallel Problem Solving from Nature - PPSN IX*, *4193*, 870–879. https://doi.org/doi:10.1007/11844297_88

Gandomi, A. H., Mohammadzadeh S., D., Pérez-Ordóñez, J. L., & Alavi, A. H. (2014). Linear genetic programming for shear strength prediction of reinforced concrete beams without stirrups. *Applied Soft Computing*, *19*, 112–120. https://doi.org/10.1016/j.asoc.2014.02.007

Guven, A. (2009). Linear genetic programming for time-series modelling of daily flow rate. *Journal of Earth System Science*, *118*(2), 137–146. https://doi.org/10.1007/s12040-009-0022-9

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software. *ACM SIGKDD Explorations Newsletter*, *11*(1), 10. https://doi.org/10.1145/1656274.1656278

Harper, R. (2012). Spatial co-evolution: quicker, fitter and less bloated. *Gecco*, 759–766. https://doi.org/doi:10.1145/2330163.2330269

Hu, T., Banzhaf, W., & Moore, J. H. (2013). Robustness and evolvability of recombination in linear genetic programming. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 7831 LNCS, pp. 97–108). https://doi.org/10.1007/978-3-642-37207-0_9

Hu, T., Payne, J. L., Banzhaf, W., & Moore, J. H. (2011). Robustness, evolvability, and accessibility in linear genetic programming. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 6621 LNCS, pp. 13–24). https://doi.org/10.1007/978-3-642-20407-4_2

Jacobsen-Grocott, J., Mei, Y., Chen, G., & Zhang, M. (2017). Evolving heuristics for Dynamic Vehicle Routing with Time Windows using genetic programming. In *Evolutionary computation (cec), 2017 ieee congress on* (pp. 1948–1955). IEEE.

Keijzer, M. (2003). Improving Symbolic Regression with Interval Arithmetic and Linear Scaling. *Genetic Programming Proceedings of EuroGP2003*, *2610*, 70–82. https://doi.org/10.1007/3-540-36599-0_7

Korns, M. F. (2011). Accuracy in Symbolic Regression. In *Genetic programming theory and practice ix* (pp. 129–151). https://doi.org/doi:10.1007/978-1-4614-1770-5_8

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, *4*(2), 87–112. https://doi.org/10.1007/

BF00175355

Langdon, W. B., & Harman, M. (2015). Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, *19*(1), 118–135. https://doi.org/10.1109/TEVC.2013.2281544

Lenz, G., Wright, G., Dave, S., Xiao, W., Powell, J., Zhao, H., ... Staudt, L. (2008). Stromal Gene Signatures in Large-B-Cell Lymphomas. *New England Journal of Medicine*, *359*(22), 2313–2323. https://doi.org/10.1056/NEJMoa0802885

Luke, S. (1998). Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 214–222.

Luke, S. (2010). The ECJ Owner's Manual. *San Francisco, California, A User Manual for the ECJ ...*. Retrieved from http://ecj.googlecode.com/svn-history/r362/trunk/ecj/docs/manual/manual.pdf

MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*, *52*(7), 1015–1030. https://doi.org/10.1287/mnsc.1060.0552

McPhee, N. F., & Poli, R. (2008). Memory with Memory: Soft Assignment in Genetic Programming. In *GECCO'08* (pp. 1235–1242). https://doi.org/10.1145/1389095.1389336

Nordin, P., & Banzhaf, W. (1995). Complexity Compression and Evolution. In *Genetic algorithms: Proceedings of the sixth international conference (icga95)* (pp. 310–317). Retrieved from http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.2133

Nordin, P., Francone, F., & Banzhaf, W. (1996). Explicitly defined introns and destructive crossover in genetic programming. *Advances in Genetic Programming*, *2*, 111——134. Retrieved from http://www.cs.mun.ca/$/sim$banzhaf/papers/ML95.pdf

Pagie, L., & Hogeweg, P. (1997). Evolutionary consequences of coevolving targets. *Evolutionary Computation*, *5*(4), 401–18. https://doi.org/10.1.1.36.7856

Poli, R., & Cagnoni, S. (1997). Genetic Programming with User-Driven Selection: Experiments on the Evolution of Algorithms for Image Enhancement. In *Genetic programming 1997: Proceedings of the 2nd annual conference* (pp. 269–277).

Ravansalar, M., Rajaee, T., & Kisi, O. (2017). Wavelet-linear genetic programming: A new approach for modeling monthly streamflow. *Journal of Hydrology*, *549*, 461–475. https://doi.org/10.1016/j.jhydrol.2017.04.018

Sonnenburg, S., Braun, M. L., Ong, C. S., Bengio, S., Bottou, L., Holmes, G., ... Williamson, R. (2007). The Need for Open Source Software in Machine Learning. *J.*

Mach. Learn. Res., *8*, 2443–2466. Retrieved from http://dl.acm.org/citation.cfm?id=1314498.1314577

Sotto, L. F. D. P., Melo, V. V. de, & Basgalupp, M. P. (2016). An improved λ-linear genetic programming evaluated in solving the Santa Fe ant trail problem. In *Proceedings of the 31st annual ACM symposium on applied computing* (pp. 103–108). https://doi.org/doi:10.1145/2851613.2851669

Troiano, L., Birtolo, C., & Armenise, R. (2016). Searching optimal menu layouts by linear genetic programming. *Journal of Ambient Intelligence and Humanized Computing*, *7*(2), 239–256. https://doi.org/10.1007/s12652-015-0322-7

Tsanas, A., Little, M. A., McSharry, P. E., & Ramig, L. O. (2010). Accurate telemonitoring of parkinsons disease progression by noninvasive speech tests. *IEEE Transactions on Biomedical Engineering*, *57*(4), 884–893. https://doi.org/10.1109/TBME.2009.2036000

Uy, N. Q., Hoai, N. X., O'Neill, M., McKay, R. I., & Galván-López, E. (2011). Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, *12*(2), 91–119. https://doi.org/10.1007/s10710-010-9121-2

Vladislavleva, E. J., Smits, G. F., & Hertog, D. den. (2009). Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, *13*(2), 333–349. https://doi.org/10.1109/TEVC.2008.926486

Wagner, S., & Affenzeller, M. (2002). The HeuristicLab Optimization Environment. *Evolutionary Computation*, 1–15.

Watchareeruetai, U., Takeuchi, Y., Matsumoto, T., Kudo, H., & Ohnishi, N. (2011). Redundancies in linear GP, canonical transformation, and its exploitation: A demonstration on image feature synthesis. *Genetic Programming and Evolvable Machines*, *12*(1), 49–77. https://doi.org/10.1007/s10710-010-9118-x

White, D. R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B. W., Kronberger, G., ... Luke, S. (2013). Better GP benchmarks: Community survey results and proposals. https://doi.org/10.1007/s10710-012-9177-2

Wilson, G., & Banzhaf, W. (2010). Interday foreign exchange trading using linear genetic programming. *GECCO 10 Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, *3*, 191–212. https://doi.org/10.1007/978-1-4419-1626-6

Yu, T., & Miller, J. (2001). Neutrality and the Evolvability of Boolean Function Landscape. In *Genetic programming* (pp. 204–217). https://doi.org/10.1007/

8 Appendix

Appendix A

Problem Example Code Definition

```
package lgp.benchmark.problems

import lgp.benchmark.*
import lgp.benchmark.operations.CustomOperationLoader
import lgp.core.environment.*
import lgp.core.environment.config.Config
import lgp.core.environment.config.ConfigLoader
import lgp.core.environment.constants.DoubleConstantLoader
import lgp.core.environment.dataset.*
import lgp.core.evolution.*
import lgp.core.evolution.fitness.FitnessCase
import lgp.core.evolution.fitness.FitnessFunction
import lgp.core.evolution.fitness.FitnessFunctions
import lgp.core.evolution.population.*
import lgp.core.modules.ModuleInformation
import lgp.lib.BaseInstructionGenerator
import lgp.lib.BaseProgramGenerator
import java.util.*

class Keijzer6Solution(
        problem: String,
        trainingResult: TrainingResult<Double>,
        testResult: TestResult<Double>,
        testingFitness: Double
) : BenchmarkSolution<Double>(problem, trainingResult, testResult, testingFitness)

class Keijzer6Problem(options: BenchmarkOptions) : BenchmarkProblem<Double>(options) {

    override val name = "Keijzer-6"

    override val description = BenchmarkProblemDescription(
            details = mapOf(
            "Name" to this.name,
            "Variables" to "1 (x)",
            "Equation" to "f(x) = sum(1 / i) for i in [1, x]",
            "Operations" to "+, -, *, /, sin, sqrt, ln, inverse",
            "Training Set" to "E[1, 50, 1]",
            "Testing Set" to  "E[1, 120, 1]",
            "Fitness Function" to "MSE",
            "Author" to "Keijzer, M.",
            "See" to "http://dl.acm.org/citation.cfm?id=1762676"
        )
    ).format()

    override val configLoader = object : ConfigLoader {
        override val information = ModuleInformation(
                "Custom configuration for the Keijzer6 problem."
        )

        override fun load(): Config {
            val config =  Config().apply {
                initialMinimumProgramLength = 30
                initialMaximumProgramLength = 60
                minimumProgramLength = 30
                maximumProgramLength = 200
                numFeatures = 1
                constantsRate = 0.4
                numCalculationRegisters = 8
                populationSize = 500
                generations = 100
                microMutationRate = 0.75
                macroMutationRate = 0.25
                numOffspring = 4
                crossoverRate = 0.5
            }

            return config
        }
    }

```

```
    override val constantLoader = DoubleConstantLoader(
            constants = listOf("-1.0", "0.0", "1.0")
    )

     val trainingDatasetLoader = object : DatasetLoader<Double> {
         val func = { x: Double ->
             (1..x.toInt()).map { i ->
                 1.0 / i
             }.sum()
         }

         val gen = SequenceGenerator()

        override val information = ModuleInformation(
                "Generates a data set for the Keijzer6 problem."
        )

        override fun load(): Dataset<Double> {
            val xs = gen.generate(
                start = 1.0,
                end = 50.0,
                step = 1.0,
                inclusive = true
            ).map { x ->
                Sample(
                    listOf(Feature(name = "x", value = x))
                )
            }.toList()

            val ys = xs.map { x ->
                this.func(x.feature("x").value)
            }

            return Dataset(xs.toList(), ys.toList())
        }
    }

    val testDatasetLoader = object : DatasetLoader<Double> {
        val func = { x: Double ->
            (1..x.toInt()).map { i ->
                1.0 / i
            }.sum()
        }

        val gen = SequenceGenerator()

        override val information = ModuleInformation(
                "Generates a data set for the Keijzer6 problem."
        )

        override fun load(): Dataset<Double> {
            val xs = gen.generate(
                start = 1.0,
                end = 120.0,
                step = 1.0,
                inclusive = true
            ).map { x ->
                Sample(
                    listOf(Feature(name = "x", value = x))
                )
            }.toList()

            val ys = xs.map { x ->
                this.func(x.feature("x").value)
            }

            return Dataset(xs, ys.toList())
        }
    }

    override val defaultValueProvider = DefaultValueProviders.constantValueProvider(1.0)

    override val fitnessFunction: FitnessFunction<Double> = { outputs, cases ->
        val mse = (FitnessFunctions.MSE())(outputs, cases)

        mse
    }

    override val operationLoader = CustomOperationLoader(
        listOf(
            lgp.lib.operations.Addition::class.java,
            lgp.lib.operations.Subtraction::class.java,
            lgp.lib.operations.Multiplication::class.java,
            lgp.lib.operations.Division::class.java,
            lgp.lib.operations.Sine::class.java,
            lgp.benchmark.operations.SquareRoot::class.java,
            lgp.benchmark.operations.NaturalLog::class.java,
            lgp.benchmark.operations.Inverse::class.java
        )
    )

    override val registeredModules = ModuleContainer(
```

```
163             modules = mutableMapOf(
164                 CoreModuleType.InstructionGenerator to {
165                     BaseInstructionGenerator(environment)
166                 },
167                 CoreModuleType.ProgramGenerator to {
168                     BaseProgramGenerator(environment, sentinelTrueValue = 1.0)
169                 },
170                 CoreModuleType.SelectionOperator to {
171                     TournamentSelection(environment, tournamentSize = 4)
172                 },
173                 CoreModuleType.RecombinationOperator to {
174                     LinearCrossover(
175                             environment,
176                             maximumSegmentLength = 6,
177                             maximumCrossoverDistance = 5,
178                             maximumSegmentLengthDifference = 3
179                     )
180                 },
181                 CoreModuleType.MacroMutationOperator to {
182                     MacroMutationOperator(
183                             environment,
184                             insertionRate = 0.5,
185                             deletionRate = 0.5
186                     )
187                 },
188                 CoreModuleType.MicroMutationOperator to {
189                     MicroMutationOperator(
190                             environment,
191                             registerMutationRate = 0.33,
192                             operatorMutationRate = 0.33,
193                             constantMutationFunc = {
194                                 v -> v + (Random().nextGaussian() * 1.0)
195                             }
196                     )
197                 }
198             )
199         )
200 
201         override fun initialiseEnvironment() {
202             this.environment = Environment(
203                 this.configLoader,
204                 this.constantLoader,
205                 this.operationLoader,
206                 this.defaultValueProvider,
207                 this.fitnessFunction
208             )
209 
210             this.environment.registerModules(this.registeredModules)
211         }
212 
213         override fun initialiseModel() {
214             this.model = Models.SteadyState(this.environment)
215         }
216 
217         override fun solve(): Keijzer6Solution {
218             try {
219                 val trainer = Trainers.DistributedTrainer(
220                     environment,
221                     model,
222                     runs = this.options.runs
223                 )
224                 val trainingDataset = this.trainingDatasetLoader.load()
225                 val trainingResult = trainer.train(trainingDataset)
226 
227                 val bestModel = trainingResult.evaluations.zip(trainingResult.models)
228                         .sortedBy { (evaluation, _) -> evaluation.best.fitness }
229                         .map      { (_, model)      -> model }
230                         .first()
231 
232                 val testDataset = this.testDatasetLoader.load()
233 
234                 val testResult = bestModel.test(testDataset)
235 
236                 val testFitness = this.fitnessFunction(
237                     testResult.predicted,
238                     testDataset.inputs.zip(testDataset.outputs).map { (features, target) ->
239                         FitnessCase(features, target)
240                     }
241                 )
242 
243                 return Keijzer6Solution(this.name, trainingResult, testResult, testFitness)
244             } catch (ex: UninitializedPropertyAccessException) {
245                 throw ProblemNotInitialisedException(
246                     "The initialisation routines for this problem must be run."
247                 )
248             }
249         }
250     }
```