PyshGP: PushGP in Python

Edward Pantridge
MassMutual Financial Group
Amherst, Massachusetts, USA
epantridge@massmutual.com

Lee Spector
Hampshire College
Amherst, Massachusetts, USA
lspector@hampshire.edu

ABSTRACT

The PushGP genetic programming system, which evolves programs expressed in the Push programming language, has been used for a variety of research projects and applications over its sixteen-year history. PushGP relies on an implementation of the Push language in a host language, and it is generally easiest to use PushGP in projects in which most other components, such as fitness functions and data access instructions, are written in the same host language. While versions of Push have been written in nearly a dozen different languages, a full-featured implementation in Python would make it available to a particularly large user base, and facilitate its integration with a wide range of existing data science tools. This paper presents `pyshgp` as an open-source PushGP framework implemented in the Python programming language, and describes some of its features for data science applications.

CCS CONCEPTS

•**Software and its engineering** → **Genetic programming;**
•**Theory of computation** → *Evolutionary algorithms;*

KEYWORDS

Genetic programming, Machine Learning

ACM Reference format:
Edward Pantridge and Lee Spector. 2017. PyshGP: PushGP in Python. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082468

1 INTRODUCTION

PushGP is a genetic programming system that has been under continuous development, and has been used for a variety of research projects, since its initial development in 2001 [17]. PushGP evolves programs in the Push programming language, which was designed to combine a minimal syntax with maximal expressive semantics; this combination was sought in order to allow for relatively unconstrained variation while searching the space of arbitrary computational processes.

Both the Push language and the PushGP evolutionary framework must be implemented in a host language. While implementations have been developed in many host languages (see below), the most complete systems have been developed in languages with relatively small user bases, limiting participation in PushGP reseach and application development.

This paper presents the `pyshgp` package, which implements PushGP in the Python programming language. Python is an extremely popular language in the machine learning and artificial intelligence communities. Given that `pyshgp` is an open source project, it is our goal that these larger communities will easily be able to get involved with PushGP through the use of `pyshgp`, and that contributions will be made to the project and to research and development with PushGP more generally.

2 PUSHGP

In the Push programming language, computations are performed by instructions that take inputs from, and put outputs onto, typed data stacks. Any implementation will include stacks for some particular set of data types, which will generally include a range of common types such as `integer`, `boolean`, and `string`, but may also include more complex or special-purpose types. The collection of stacks, together with their contents, constitutes the entire state of a Push interpreter [17].

In addition to common and special purpose types, Push supports a `code` type and includes a special stack, called the `exec` stack, onto which programs are pushed for execution. The `exec` stack contains, at all times during the execution of a program, the code that remains to be executed, and the process of executing a Push program primarily involves a loop of popping and executing the top element of the `exec` stack. When a literal (such as the integer 3) is executed, it is pushed onto the stack of the appropriate data type (in this case, the `integer` stack). When an instruction is executed, all of the arguments that it requires are popped off of the appropriate stacks, and the results are pushed onto the appropriate stacks. For example, the `integer_add` instruction pops two integers off of the `integer` stack and pushes their sum back onto the `integer` stack. If an instruction requires arguments that are not present on the stacks (for example, if `integer_add` is executed in a context with fewer than 2 integers on the `integer` stack), then the instruction acts as a "no-op" and has no effect. Instructions that explicitly manipulate the contents of the `exec` stack can alter the future of a computation and thereby express novel and complex control structures; they are responsible for much of Push's expressiveness [16].

Push's syntactic minimality stems from the fact that arguments are passed between instructions via typed stacks, rather than through the relative positions of instructions and arguments in a program's text. This means that an instruction can never be given an input of the wrong type, because the instruction itself will always take its inputs from the stacks of the correct types, if they are available. In conjunction with the "no-op" behavior described

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082468

above, this also means that any sequence of Push literals and instructions is valid and executable. Push programs can be nested (with parentheses or brackets), which allows subprograms to be delineated and independentently manipulated on the `exec` stack, and this introduces the only real syntax constraint on Push programs: parentheses or brackets must be balanced.

We use the name "PushGP" to describe any genetic program that evolves Push programs. Aside from using Push as the language in which evolving programs are expressed, a PushGP system might use techniques employed in many other genetic programming systems. For example, it might use various common techniques for parent selection and genetic variation, and many PushGP systems have used techniques borrowed from tree-based GP [9], including tournament selection and crossover based on the swapping of sub-expressions.

With respect to genetic variation, however, a linear representation for Push programs has recently been developed, called Plush (the "l" is for "linear"). This allows programs to be generated and varied as linear sequences ("Plush genomes"), facilitating uniformity of variation [8]. Plush genomes are "expressed" to produce possibly-nested Push programs prior to execution.

2.1 Existing PushGP Implementations

It is relatively easy to build a basic Push interpreter, with support for a few core types and instructions, in any programming language. As a result, Push implementations have been developed in at least the following languages: C++, Clojure, Common Lisp, Elixir, Java, Javascript, Racket, Ruby, Scala, Scheme, and Swift.

The most complete and actively maintained implementations of Push at present are written in Clojure (a Lisp for the Java Virtual Machine). Clojush is an example of one such PushGP implementation [1]. Prior to the use of Clojure, most PushGP research and development was conducted in Common Lisp or Scheme. While these Lisp dialects provide a convenient platform for experimentation, they have limited the number of users who might find it convenient to integrate PushGP into their work.

Because Python is widely used, both by new programmers and by professionals, and because many easy-to-use data science tools are also available in Python, we decided to develop a full-featured implementation of Push and PushGP in Python. This should allow many more users to try PushGP in conjunction with their other tools, without significant effort. It should also allow many more developers to contribute to PushGP research and the PushGP software ecosystem.

3 PUSHGP IN PYTHON: PYSHGP

This paper presents `pyshgp`, an open source python package that implements PushGP. The main goal of `pyshgp` is to be easily usable in more contexts than just genetic programming research. Ideally, this would foster an increase in awareness and usage of PushGP not only in the genetic programming community, but also in the wider machine learning community and among application-oriented data scientists looking for powerful new tools.

This goal is what led to the decision to use Python as the host language for `pyshgp`, as Python is a commonly used language in machine learning. The desire for usability also caused the structure of the `pyshgp` package to split into two main parts: A Push language interpreter, and a genetic programming framework.

3.1 The pyshgp Push Interpreter

The Push language interpreter that exists as part of `pyshgp` is implemented independently of the genetic programming framework. It contains the implementation of a standard Push instruction set, as well as a class capable of executing Push programs.

The set of instructions included with the current version of the `pyshgp` interpreter is similar to that of the `Clojush` PushGP framework. There are five basic supported data types that each have a set of instruction associated with them. Examples of the provided instructions include the following (which were selected from much larger instruction sets):

Boolean: AND, OR, NOT, XOR
Character: char_is_letter, char_is_digit, char_is_white_space
Code: do_times, do_while, exec_when, exec_if
Integer: $+, -, *, \div, modulus, <, >, \leq, \geq, min, max, increment$
Float: $+, -, *, \div, modulus, <, >, \leq, \geq, min, max, sin, cos, tan$
String: concat, head, tail, split, length, reverse, replace

The `pyshgp` instruction set also contains type casting instructions that convert elements on the Push stacks to different types and move them to the appropriate stacks. For example, the `integer_to_boolean` instruction takes the top integer off of the integer stack, and pushes a `False` to the boolean stack if the integer was a 0; otherwise a `True` is pushed to the boolean stack. Type casting instructions exist between all types, where appropriate.

Some supported stack types also have a corresponding vector type. The `pyshgp` Push interpreter currently supports the following vector types: `integer_vector`, `float_vector`, `boolean_vector`, and `string_vector`. All vector types support the same instructions. These vector instructions perform typical vector operations, such as concatenation of vectors, appending new values, splitting vectors, reversing vectors, and retrieving values at particular indices of a vector.

All stacks, both scalar and vector, contain common stack instructions that manipulate the order of items, remove items, or add items. Some examples of these common instructions include:

dup: Duplicates the top item of the stack.
swap: Swaps the position of the top two items on the stack.
flush: Removes all elements on the stack.
stack_depth: Pushes the size of the stack to the integer stack.
yank: Moves the element at index i in the stack to the top of the stack, where i is given by the top integer on the integer stack.
eq: If the top two elements of the stack are equal, pushes `True` to the boolean stack otherwise pushes `False`.

The instructions listed in this section give an idea of the types of operations included in the `pyshgp` instruction set, but are not the entire set of instructions. To summarize the types of operations included in the instruction set one might say they are roughly the same operations provided by popular languages humans write code in (Python, Javascript, Java, etc).

This set of instructions, which deals with a variety of data types including code and vectors, is part of what makes PushGP so capable

[1] https://github.com/lspector/Clojush

of solving general program synthesis tasks [3, 6]. It is also possible to perform Symbolic Regression and Classification tasks using this instruction set.

Often if it is helpful to define new instructions when applying PushGP to a particular domain. In pyshgp, each instruction is an instantiation of the `Instruction` class. This class requires a Python function that takes a `PushState` object as its argument and returns a modified `PushState`. All instantiations of the `Instruction` class are stored in a set of registered instructions. To add additional instructions to the pyshgp instruction set, one must instantiate a new `Instruction` object and add it to the set of registered instructions.

Although the Push interpreter is used heavily during evolutionary runs in pyshgp, it can also be used standalone to execute Push programs. This allows for other search methods, aside from evolution, to be used to search for a program. Also, previously evolved programs can be reused in other contexts using just the Push interpreter in pyshgp.

3.2 Genetic Programming in pyshgp

PushGP systems evolve programs in the Push language, which can be executed using a Push interpreter. During genetic programming runs, these programs must be evaluated by a user defined error function. These error functions must take the following arguments: A push program, and a set of training cases. Each training case consists of a list of input values to be supplied to each program and a list of target output values.

In most cases, the user defined error function will iterate over the list of training cases. At each iteration, the push program is executed using the list of inputs for the current training case. The output of the push program is compared against the target output values of the training case to compute a single error value. Each of the error values calculated on each training case are returned in a "error vector".

Note that pyshgp does not aggregate error values during evaluation. An individual holds its entire error vector and its total error so that selection methods can use either the aggregated error, or the entire error vector.

The evaluation stage of evolution is where pyshgp relies on some basic parallel computation. Each individual's evaluation is independent, thus evaluation of an entire population can be made an "embarrassingly" parallel task. In pyshgp this is accomplished by the `pathos` library [11]. Pathos contains the functionality to create a pool of process running on separate CPU cores. Pathos, unlike Python's built in `multiprocessing` library, is capable of "pickling" (converting to a byte stream) anonymous and nested python functions to be sent to the processes in the pool. This is crucial for sending `pyshgp's` instruction set to each process so the programs can be executed.

Although some speed increase is observed when using parallel evaluation, it has been expressed by some users that requiring the Pathos library as a dependency is not ideal, and it is likely that more can be done in this area to improve how pyshgp parallelizes the evaluation of the population.

Three selection methods are supported by pyshgp. The default selection method is Lexicase Selection. Epsilon-Lexicase Selection and Tournament Selection are the two other currently implement selection methods. Both forms of Lexicase selection select parents based on their error vectors, while tournament selection selects based on total error [7, 10, 13].

Push programs consist of nested lists of *instructions* and *literals*. Although genetic operators can be performed on Push programs directly, more recent PushGP systems use linear representations of these programs to allow for more flexible genetic operators. These linear representations are called Plush genomes [8]. The genetic programming that is implemented in pyshgp uses these Plush genomes during the variation stage of evolution. These genomes are easily translated into Push programs for the evaluation stage of evolution.

Currently uniform mutation is the only form of mutation that is supported by pyshgp [8, 14]. Uniform mutation iterates over a Plush genome, potentially mutating each gene. Genes can either be instructions, or literals of any supported data type. If uniform mutation mutates a gene which is an instruction, it is either replaced by a random instruction or a random literal. If uniform mutation mutates a gene which is a literal, it has a probability of being replaced by an instruction, otherwise the literal is mutated using a "constant mutator". These mutators vary based on the data type of the literal. Integers and Floats are perturbed by Gaussian noise. Strings have random characters replaced by new characters. Booleans are randomly set to a new value.

The only recombination operator currently included in the pyshgp genetic programming framework is alternation [8, 14]. Alternation iterates over both parent genomes to build the child genome out of genes taken from both parents. Initially a "read head" begins to iterate over the first parent genome, copying each gene to the child genome. At each iteration, there is a probability of switching which parent the genes are being copied from. If an iteration of alternation switches between parent genomes, the position in the new parent genome which alternation begins to copy from is the same as the position in the previous parent genome after being perturbed by Gaussian noise. In other words, when changing which parent genome to copy genes from, the "read head" may be pushed forward or backward a random number of genes. This is called alignment deviation, and it can cause genes of parent genomes to be skipped or to be repeated in the child genome.

All parameters and tunable probabilities related to Uniform Mutation and Alternation can be set by the user as described in section 4.1 and Figure 1.

Pyshgp allows for the user to specify the percent of each generation that is created via each genetic operator, as well as combinations of genetic operators (ie. Number of children produced by Alternation followed by Uniform Mutation).

It is likely that these large, relatively complex genetic operators are not optimal for all problems that one would want to try pyshgp on. A valuable area of future development and research is the implementation of a many smaller genetic operators, and a more robust way of chaining genetic operators together.

3.3 Automatic Program Simplification

Programs in the Push language are often difficult for human to understand due to their lack of informative syntax. This issue is made more drastic in Push programs produced by evolution, because many instructions that have no effect on the program's

behavior tend to be present. A version of this problem arises in other forms of genetic programming as well.

This issue has prompted most modern implementations of PushGP, including pyshgp, to perform automatic program simplification after GP runs [4, 15]. The process of automatic program simplification in pyshgp operates on a Plush genome using an error function, over a user defined number of simplification steps. The error function used is generally the same as the error function used during evolution.

At each simplification step, a small random number of genes in the genome are flagged as "silent". These genes are not included when the Plush genome is translated into a executable Push program. The genome is translated into a new, slightly smaller Push program which is evaluated using the error function. If the error value of the smaller program is equal to or lower than the original program, the genes remain silent, otherwise all of the genes are returned to their previous state.

Occasionally evolution produces Push programs that contain instruction that do not effect program behavior, but their presence is necessary. For example, consider the follow program.

```
[exec_stack_depth, float_add, string_length]
```

The `exec_stack_depth` instruction pushes the length of the remaining program onto the integer stack. In this case, the `exec_stack_depth` instruction would push a 2 to the integer stack. Assuming this program was run on an empty Push state, the `float_add` and `string_length` instructions will not have any arguments on the float and string stacks respectively, and thus will not be executed by the interpreter. However, if either the `float_add` or `string_length` instructions are removed from the program the result of `exec_stack_depth` is impacted.

The above program can be made arguably easier to understand by replacing the `float_add` and `string_length` instructions in `noop` instructions that explicitly do not have a behavior, but do take up space in the program.

The process of automatic program simplification in pyshgp operates on a Plush genome by following Algorithm 1.

Input: genome
Input: error_func
Input: num_steps
for *i in 1 to num_steps* **do**
  n ← Random number in [1, 2, 3] ;
  action ← Random element of ["silent", "noop"] ;
  if *action == "silent"* **then**
    new_genome ← genome with *n* random genes flagged as silent.;
  else
    new_genome ← Genome with *n* random genes replaced with with noop instructions ;
  end
  old_err ← Translate genome and pass to error_func ;
  new_err ← Translate new_genome and pass to error_func ;
  if *new_err* ≤ *old_err* **then**
    genome ← new_genome
  end
end
return genome

Algorithm 1: The process of automatic program simplification in pyshgp

4 USAGE

Initially, pyshgp was created to be a standalone Python package. It included the Push interpreter and a genetic programming framework but little was included to help use the two together. The user had a to write a new fitness function for every application, and set the many hyperparameters related to evolution manually.

In many situations this is still true for the current state of pyshgp but some improvements have been made which help integrate pyshgp with the popular machine learning package, Scikit-learn (see below).

4.1 Hyperparameters For Evolution

Like many evolutionary computation frameworks, pyshsp has a large number of tunable hyperparamenters that need to be set for each evolutionary run. The most important hyperparameters can be found in Figure 1, although pyshgp also includes hyperparamters to toggle which values should be printed during evolution to get a sense of run health and diversity of the population.

All of these hyperparameters have default values although it is highly unlikely one would want to keep all of these settings when applying pyshgp to a particular problem. There are two ways to easily override the default hyperparameters when using pyshgp as a standalone Python package.

If the user has hyperparameter values they would like to consistently use in their application of pyshgp, they can create a Python dictionary containing the new values and pass the dictionary to the pyshgp main evolution function. This will override the default hyperparameter values that are present in the dictionary. This is the most common way to set the `atom_generators`, which dictate which instructions and literals will be included in random program generation.

If the user would like to easily launch multiple runs of pyshgp with different hyperparameter values, this can be achieved through command line arguments. Hyperparameters set via the command line when starting a pyshgp run will overwrite both the default hyperparameter values and the hyperparameter values specified in the dictionary passed to the evolution function. This is the best way to experiment with different hyperparameter values without changing any code.

An example usage of pyshgp as a standalone Python package can be found in Figure 2. This source code denotes how one might use pyshgp to evolve a program that determines if a number is odd or even. This is one of many benchmark problems included included in the pyshgp project on Github.

4.2 Scikit-Learn Integration

As previously mentioned, a major goal of pyshgp is to bring PushGP to a larger audience. An important feature of pyshgp that helps achieve this goal is the integration with Scikit-learn [12].

Scikit-learn is a popular and accessible machine learning package written in python. Many commonly used machine learning

Hyperparameter Name	Description
error_threshold	If any total error of individual is below this, that is considered a solution.
population_size	Size of the population at each generation.
max_generations	Max generations before evolution stops. Will stop sooner if solution is found.
max_genome_initial_size	Maximum size of random genomes generated for initial population.
max_points	Maximum size of push genomes and push programs, as counted by points in the program.
atom_generators	The instructions that pushgp will use in random code generation.
genetic_operator_probabilities	Probabilities of parents undergoing each genetic operator to produce a child.
selection_method	Options are 'lexicase'[7], 'epsilon_lexicase'[10] or tournament'.
epsilon_lexicase_epsilon	Defines a hard-coded epsilon. If None, automatically defines epsilon using MAD.
tournament_size	If using tournament selection, the size of the tournaments.
alternation_rate	When using alternation, how often alternates between the parents.
alignment_deviation	When using alternation, the standard deviation of how far alternation may jump between indexes when switching between parents.
uniform_mutation_rate	The probability of each token being mutated during uniform mutation.
uniform_mutation_constant_tweak_rate	The probability of using a constant mutation instead of simply replacing the token with a random instruction during uniform mutation.
uniform_mutation_float_gaussian_standard_deviation	The standard deviation used when tweaking float constants with Gaussian noise.
uniform_mutation_int_gaussian_standard_deviation	The standard deviation used when tweaking integer constants with Gaussian noise.
uniform_mutation_string_char_change_rate	The probability of each character being changed when doing string constant tweaking.
final_simplification_steps	The number of simplification steps that will happen upon finding a solution.
max_workers	If 1, pyshgp runs in single thread. Otherwise, pyshgp runs in parallel. If None, uses number of cores on machine.

Figure 1: Most of the parameters needed to be set for a genetic programming run with pyshgp. Some parameters were not included to conserve space. Excluded parameters were regarding which values to print at each generation to monitor run health.

techniques are available in Scikit-learn including: Support Vector Machines, Random Forrest Classifiers, Artificial Neural Networks, various clustering algorithms, and Principle Component Analysis.

Scikit-learn also has model selection and data preprocessing capabilities. This includes implementations of k-fold cross-validation, grid search hyperparameter optimization, and a variety of feature transformations.

Pyshgp includes implementations of classes that inherit from Scikit-learn's base estimators. This allows for pyshgp to easily be compared against other machine learning methods. Also, Scikit-learn's model selection and data preprocessing capabilities can be easily combined with pyshgp's estimator classes to create pipelines that perhaps perform better than using only pyshgp.

4.3 PushGPRegressor

The `PushGPRegressor` class that is included in `pyshgp` can be used to easily perform symbolic regression alongside other Scikit-learn components.

The advantage provided by the `PushGPRegressor` class is the generation of a suitable error function from a given dataset. When the `fit` method is called, a function is created which uses the training data to produce a simple error function. This error function uses the Mean Squared Error (MSE) as the default error metric, but other metrics supported by Scikit-learn can be used as well.

4.4 PushGPClassifier

The `PushGPClassifier` class is similar to the `PushGPRegressor` class, except it is designed to simplify using `pyshgp` alongside other Scikit-learn components for Classification tasks.

Much like the `PushGPRegressor`, the `PushGPClassifier` generates an error function when the `fit` method is called with training data. To produce class predictions, `pyshgp` relies on the use of voting instructions. Each vote instruction is associated with a class in the classification problem, as well as one of the numerical stack types (integer or float). Each time one of these instructions is processed by the Push interpreter, a vote amount it taken from the numerical stack and added to the current vote level of the associated class. After program execution, the class with the highest vote level is considered the prediction.

5 BENCHMARKS

This paper presents some basic benchmarks of the `pyshgp` framework. These benchmarks consist of some of the example problems that can be found in the `pyshgp` repository on GitHub.

Below are the descriptions for each problem used to benchmark for this paper [1, 2, 5, 6].

Access: Given a vector of integers, and an index integer, i, return the integer at position i in the vector.

Decrement: Given a vector of integers, return the same vector with each element decremented by 1.

Iris: Based on a small dataset containing measurements of iris flowers, classify the flower species.

Odd: Evolve a program which produces True if the input integer is odd, and False otherwise.

Replace Space With Newline (RSWN): Given an input string, print the string with each space character replaced by a newline character and return an integer equal to the number of non-whitespace characters.

Integer Regression: Fit the polynomial $x^3 - 2x^2 - x$ using integer constants.

String: Remove the last 2 characters of input string, and concatenate the result with itself.

Below is a table of the success rates for each of these benchmark problems.

Problem	Solutions	Number of Runs
Access	5	5
Decrement	5	5
Iris	5	7
Odd	10	10
RSWN	6	10
Integer Regression	10	10
String	10	10

Unfortunately gathering meaningful runtime data on these benchmark runs proved to be difficult. Each run was performed on the ANONYMIZED College Cluster Computing Facility, where each run is assigned a variable number of CPU cores and memory based on resource availability. These differences in resources make getting consistent timings for each benchmark problem difficult.

In the below table, the average runtime of each problem is given, as well as the average number of generations taken to find a solution. For runs where no solution was found, the maximum number of generations for the run was used.

Problem	Avg. Runtime (min)	Avg. Generations
Access	14.5	9.5
Decrement	12.8	29.0
Iris	628.51	108.14
Odd	8.01	25.3
RSWN	277.25	130.4
Integer Regression	4.60	5.0
String	1.39	5.0

Although these benchmarks are not impressive results on their own, they demonstrate that pyshgp has a working PushGP framework and can be applied to regression, binary logic, list comprehension, string manipulation, and general software synthesis tasks.

6 OPEN SOURCE DEVELOPMENT

Pyshpg is hosted on GitHub where the source code is freely available to anyone[2]. The use of git version control and GitHub allows for easy use of continuous integration tools.

6.1 Contributing

Pyshgp is not the first PushGP implementation to utilize open source contributions. Clojush is one example of a PushGP framework that has benefited from being open source. Multiple researchers, students, and enthusiasts have contributed to the Clojush project via GitHub. Ideally, pyshgp will be able to leverage these same benefits of being open source to the same degree, or possibly more due to the fact that pyshgp's host language of Python is more popular than Clojush's host language of Clojure.

6.2 Automated Testing Suite

The pyshgp project contains an automated test suite that is triggered on every Pull Request to the repository on GitHub. This test suite performs unit tests on most of the pyshgp code base, as well as tests of the Push instruction set and small validation tests of the genetic programming framework.

The tests of the instruction set are called "instruction unit tests", as they aim to test each instruction individually. To perform these tests, a mock Push state is created that contains the necessary arguments for the instruction. The instruction is then run against the Push state and the resulting Push state is compared against the expected Push state. Unlike normal program execution, the interpreter does not continue to process the elements of the exec stack until the exec stack is empty. Only the one instruction being tested is executed.

The validation tests consist of small genetic programming runs using the pyshgp genetic programming framework. These small runs are 5 generations of the example programs included in the pyshgp GitHub repository.

6.3 Automatic Documentation Generation

PushGP has undergone many changes since its creation in 2001. These changes include modification to the Push instruction set, genetic operators, Plush genomes, host language, and more. PushGP continues to be used in genetic programming research and improvements continue to be made. As the pyshgp open source community adds these changes (and perhaps pioneers some of its own) it will be important to keep documentation on the project up to date. This is an area which other current PushGP frameworks struggle with.

To address this difficulty, the Sphinx python documentation generator is used to generate documentation files from docstrings and rich text files found in the pyshgp GitHub repository. These generated documentation files take the form of HTML web documents. The pyshgp project on GitHub currently uses the ReadTheDocs service to host these web documents. This distributes the pysh documentation to many potential users and contributors.[3]

Given that all documentation is generated from resources found inside the repository, all contributions to pyshgp can be accompanied by modifications to the documentation. This will cause the ReadTheDocs documentation to always be synchronous with the current pyshgp codebase.

[2] https://github.com/erp12/pyshgp

[3] http://pysh2.readthedocs.io/en/latest/

7 FUTURE WORK

pyshgp is a young project that could benefit from many improvements. One such improvement is the more thoughtful use of parallel computation. Currently, pyshgp uses fairly simplistic multi-core population evaluation, and performance gains are modest.

Aside from improved parallel computation, pyshgp could also benefit from the use of a more lightweight Push interpreter. Given the Push interpreter mostly consists of simple operations on stacks, it could be possible to replace large parts of the Push interpreter with Python wrapped lower level language, such as C.

The current state of the genetic programming framework in pyshgp could also be improved in a number of ways. Firstly, more genetic operators that exist in genetic programming literature could be included in addition to the existing genetic operators. Second, the larger genetic operators could be broken into smaller operators which are chained together. This would allow for more experimentation and tuning of genetic operators.

The pyshgp open source community is currently extremely small, and it is not clear how easy it is to make contributions. A major goal of the pyshgp project is that the genetic programming and machine learning communities engage in maintaining and improving the codebase. It remains to be seen how feasible this will be.

A ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grants No. 1617087, 1129139 and 1331283. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Ronald A Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of eugenics* 7, 2 (1936), 179–188.

[2] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *CoRR* abs/1608.04428 (2016). http://arxiv.org/abs/1608.04428

[3] Thomas Helmuth. 2015. *General Program Synthesis from Examples Using Genetic Programming with Parent Selection Based on Random Lexicographic Orderings of Test Cases.* Ph.D. dissertation. http://scholarworks.umass.edu/dissertations_2/465/

[4] Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving Generalization of Evolved Programs through Automatic Simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017.*

[5] Thomas Helmuth and Lee Spector. 2015. *Detailed Problem Descriptions for General Program Synthesis Benchmark Suite.* Technical Report UM-CS-2015-006. Computer Science, University of Massachusetts, Amherst. https://web.cs.umass.edu/publication/docs/2015/UM-CS-2015-006.pdf

[6] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15).* ACM, New York, NY, USA, 1039–1046. DOI: http://dx.doi.org/10.1145/2739480.2754769

[7] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. DOI: http://dx.doi.org/10.1109/TEVC.2014.2362729

[8] Thomas Helmuth, Lee Spector, Nicholas Freitag McPhee, and Saul Shanabrook. 2017. Linear Genomes for Structured Programs. In *Genetic Programming Theory and Practice XIV*, William P. Worzel, William Tozier, Brian W. Goldman, and Rick Riolo (Eds.). Springer.

[9] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press. http://mitpress.mit.edu/books/genetic-programming

[10] William La Cava, Lee Spector, and Kourosh Danai. 2016. Epsilon-Lexicase Selection for Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016 (GECCO '16).* ACM, New York, NY, USA, 741–748. DOI: http://dx.doi.org/10.1145/2908812.2908898

[11] M.M. McKerns, L. Strand, T. Sullivan, A. Fang, and M.A.G. Aivazis. 2011. Building a framework for predictive science. *Proceedings of the 10th Python in Science Conference* (2011). http://arxiv.org/pdf/1202.1056

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[13] Lee Spector. 2012. Assessment of Problem Modality by Differential Performance of Lexicase Selection in Genetic Programming: A Preliminary Report. In *1st workshop on Understanding Problems (GECCO-UP)*, Kent McClymont and Ed Keedwell (Eds.). ACM, Philadelphia, Pennsylvania, USA, 401–408. DOI: http://dx.doi.org/doi:10.1145/2330784.2330846

[14] Lee Spector and Thomas Helmuth. 2013. Uniform Linear Transformation with Repair and Alternation in Genetic Programming. In *Genetic Programming Theory and Practice XI*, Rick Riolo, Jason H. Moore, and Mark Kotanchek (Eds.). Springer, Ann Arbor, USA, Chapter 8, 137–153. DOI: http://dx.doi.org/doi:10.1007/978-1-4939-0375-7_8

[15] Lee Spector and Thomas Helmuth. 2014. Effective simplification of evolved push programs using a simple, stochastic hill-climber. In *GECCO Comp '14: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion*, Christian Igel, Dirk V. Arnold, Christian Gagne, Elena Popovici, Anne Auger, Jaume Bacardit, Dimo Brockhoff, Stefano Cagnoni, Kalyanmoy Deb, Benjamin Doerr, James Foster, Tobias Glasmachers, Emma Hart, Malcolm I. Heywood, Hitoshi Iba, Christian Jacob, Thomas Jansen, Yaochu Jin, Marouane Kessentini, Joshua D. Knowles, William B. Langdon, Pedro Larranaga, Sean Luke, Gabriel Luque, John A. W. McCall, Marco A. Montes de Oca, Alison Motsinger-Reif, Yew Soon Ong, Michael Palmer, Konstantinos E. Parsopoulos, Guenther Raidl, Sebastian Risi, Guenther Ruhe, Tom Schaul, Thomas Schmickl, Bernhard Sendhoff, Kenneth O. Stanley, Thomas Stuetzle, Dirk Thierens, Julian Togelius, Carsten Witt, and Christine Zarges (Eds.). ACM, Vancouver, BC, Canada, 147–148. DOI: http://dx.doi.org/doi:10.1145/2598394.2598414

[16] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation.* ACM Press, Washington DC, USA, 1689–1696. DOI: http://dx.doi.org/10.1145/1068009.1068292

[17] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (March 2002), 7–40. DOI: http://dx.doi.org/10.1023/A:1014538503543

```
import random
import pyshgp.utils as u
import pyshgp.gp.gp as gp
import pyshgp.push.interpreter as interp
import pyshgp.push.instructions.registered_instructions as ri
import pyshgp.push.instruction as instr

def odd_error_func(program, debug = False):
    errors = []

    for i in range(10):
        # Create the push interpreter
        interpreter = interp.PushInterpreter([i])
        # Run program
        interpreter.run_push(program, debug)
        # Get output from top of boolean stack.
        prog_output = interpreter.state.stacks["_boolean"].ref(0)
        # compare to target output
        target_output = bool(i % 2)

        if prog_output == target_output:
            errors.append(0)
        else:
            errors.append(1)
    return errors

# Define some new hyperparameter values.
odd_params = {
    "population_size" : 500,
    "atom_generators" : list(u.merge_sets(ri.registered_instructions,
                                          [lambda: random.randint(0, 100),
                                           lambda: random.random(),
                                           instr.PyshInputInstruction(0)]))
}

if __name__ == "__main__":
    # Run evolution using given error function and parameters.
    gp.evolution(odd_error_func, odd_params)
```

Figure 2: Example source code for the Odd problem. This file starts a pushgp run that evolve a Push program that returns True of the input problem is odd and False otherwise.