Master Thesis intermediary report

Ben Cardoen

December 15, 2016

1 Introduction

This report details the current state of the master thesis "Convergence of symbolic regression using metaheuristics".

2 Research Question

The aim of this project is to analyze convergence of symbolic regression by combining a baseline GP implementation with metaheuristics.

3 Installation and Running

You can obtain the source code (and documentation) by:

```
$git clone git@bitbucket.org:bcardoen/csrm.git
```

The project is implemented in Python3 and has minimal dependencies. In the accompanying README.md file all dependencies are listed as well as instructions on how to run the tests. The file "test.py" provides an example run of the algorithm. The results in this work can be reproduced by executing:

```
$ python3 runbenchmarks.py
```

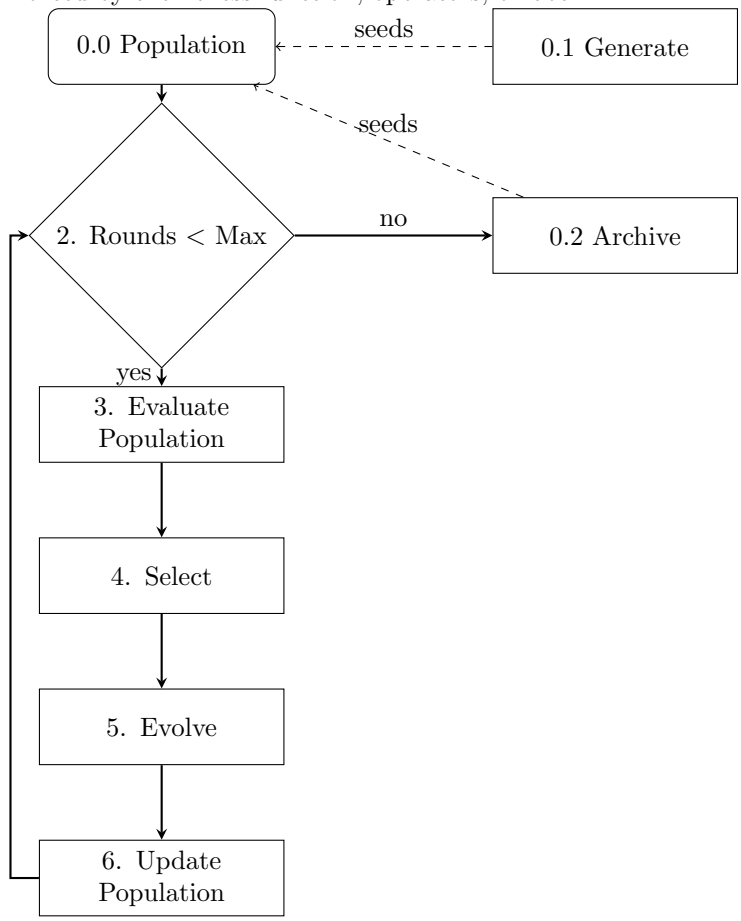
This produces html output files visualizing the algorithm output on each of the benchmark functions. These functions were taken from literature as instances of hard problems for GP SR solutions. Documentation can be found in ./doc/html/index.html.

4 Genetic Programming Implementation

4.1 Algorithm

The algorithm is a depth constrained variant of Genetic Programming. The following flowchart shows the outline of the algorithm. The restriction on depth is necessary to avoid bloat in the generated solutions. This restriction can be

enforced by the fitness function, operators, or both.



Datastructures The population is kept in a sorted collection with O(1) random access and O(log N) insertion/removal complexity.

Control flow The algorithm performs k runs, each run consists of j generations comprised of the following stages:

Selection Select given a criteria k of the population to modify. The current implementation works on the entire population, this can be easily tuned to select only the k best specimens.

Evolution In the evolution stage 2 operators modify the selected set of specimens. Mutation will generate a modified specimen which replaces the original based on the fitness score. Crossover will select 2 specimens and exchange

subtrees randomly chosen between the two specimens. The best 2 specimens of the resulting set of 4 are then retained.

Update In the update stage a validation check is done on the new specimens, the modified specimens are added to the population and if needed new specimens are generated.

Statistics At the end of a generation statistics are gathered and stored. The fitness scores and their evolution is tracked for later analysis. The algorithm can use these to detect stalling convergence. The entire set of statistics allow for a perfect trace of the algorithm stage by stage.

Archive At the end of a run a selection of the surviving population is stored in the archive. The population is then emptied and reseeded using the archive and random specimens. The combination of these approaches ensures the best results of the previous generations are not lost, while introducing new information into the process.

4.2 Fitness function

A wide range of fitness functions is possible, in this project a distance function is used that evaluates the tree on a sample of the domain and compares the result with the expected output. Several options are available as distance function, the current implementation uses a root mean square. Experiments with a normalized variant and Pearson correlation coefficient are planned. The fitness functions is a multiplied by the complexity ratio explained next.

4.3 Complexity

A complexity measure is implemented based on the operator complexity and depth of the tree. Functions are ordered by complexity giving a higher weight to tanh compared to +. The idea behind this is that a complex function such as tanh, while more expressive, will lead to overfitting more easily than a simpler function. The complexity score is a ratio of the current complexity of all nodes compared to the maximum complexity the tree could have given its depth. This ratio (within [1,2]) is used as a multiplier for the fitness score to influence the algorithm.

4.4 Initialization

Initialization requires constructing a population of expression trees. In a probabilistic directed search algorithm this is a random process, but the generation of a random tree can easily generate an invalid tree. Consider a function node with division as operator. It is clear that the right subtree of such a node should never evaluate to zero. Several approaches in literature exist to solve this problem. As the depth of the tree grows, the complexity of this problem grows exponentially.

The problem is split into detection and remediation. To detect an invalid tree requires full knowledge of the domain the tree represent. Even if we have this domain, it is not feasible to validate the tree for the entire domain. Invalid function input is caught in Python by way of exceptions. Exceptions are only justified in high performance code if the probability of an exception is very low. It is clear that this is not the case, as the depth of the tree grows the probability of invalid input for a single node grows exponentially. To address this any function node that is invalid will return None to signal an invalid domain. A naive approach then constructs an entire tree and evaluates it on the domain. With the increasing propability of invalid nodes as the depth of the tree increases, this becomes quickly infeasible. We can constrict the domain to a few values to test at construction. Should other values invalidate the tree, the corresponding fitness score will result in the elimination of the tree. We find thus a balance between generation of semantically correct trees and computational cost. It is not feasible to delay all validation to the algorithm itself, this simply moves the naive approach to the algorithm loop. Finally, by constructing and validating a tree from the bottom up, we can detect early when a tree is invalid. In contrast with a top down approach this allows us to recover from invalid choices in the random process without discarding valid subtrees. The initialization stage is repeated in the algorithm itself at a smaller scale when generating subtrees for the mutation operator and when reseeding the algorithm for a new run.

4.5 Operators

This GP algorithm features 2 operators.

4.5.1 Mutation

Mutation operates on a single specimen and modifies its structure. Typically a random subtree is chosen, removed and replaced with a newly generated subtree. This operation is, depending on the depth of the subtree and the problem domain, expensive as the newly generated subtree has to result in a valid outcome. The same bottom up construction from the initialization code is reused here as a compromise between validity of the specimen and performance.

Configuration The mutation operator can be configured to keep the depth of the tree equal. This prevents bloating but also potentially restricts the search space. Other configurations allow unrestricted growth, or variable growth but limited by a preset value. Experiments are underway to observe the effect on convergence. Mutation is very invasive, if a large subtree is selected (selection is a random process), a fit specimen can lose most of its value. Similar to the approach in Simulated Annealing, we can use a target depth configuration that forces mutation to make small changes in highly fit individuals and large in unfit individuals as a perfect balance. The current implementation mutates only on the lesser fit specimens in the population at an observed increase in runtime

without losing fitness. A mutated specimen replaces its original depending on its fitness.

4.5.2 Crossover

Two specimens are selected and copied. Each of the copies exchanges a randomly chosen subtree with the other. This exchange of information leads quickly to convergence, but does not introduce new information (e.g. diversity).

Configuration Crossover can be configured to preserve depth by exchanging subtrees of equal depth. The same concept of modifying highly fit individuals to a smaller extent than less fit individuals can be applied here but with caution, the fitness of the subtree is higher than that of a randomly generated subtree. Crossover in the current implementation picks random pairs of specimens, and from the resulting 4 specimens (2 parents, 2 offspring) picks the 2 best to replace the parents depending on fitness.

4.6 Constants

We differentiate here between 2 types of constants in an expression.

$$y = a + b * ln(c * x_0)$$

In this expression a is a standalone constant, b and c are coefficients. We will refer to b and c as embedded constants. The encoding of constants in the expression tree allows for several optimizations. In the analysis section we will observe the convergence behavior of the algorithm when we approximate the expression:

$$y = 213.80940889 - 213.80940889 * e^{-0.54723748542 * x_0}$$

The large constant values here lead to poor convergence, the algorithm cannot easily generate them by random search. The parse tree in Figure 1 for this expression has 8 nodes with a depth of 4.

Figure 1: Embedding of constants in nodes.

In our implementation each node has a built in constant multiplier, by using this representation the parse tree for the equivalent expression only needs 4 nodes with depth 2. Given our earlier discussion on the impact of depth on the algorithm it should be clear that this is a valuable contribution. Furthermore, these constants can be optimized by a metaheuristic, instead of being generated by the GP algorithm. The latter process is very ineffective and leads to much of the convergence problems in GP inspired SR solutions. Both standalone constants, with a dedicated node in the tree, and embedded coefficients are easily accessible as a vector for any optimization algorithm.

Finally the range in which constants are generated influences convergence. If the optimal solution has constants outside of the chosen range, the algorithm will have to generate an expression resulting in the correct value, a very ineffective process. If the range is larger than any of the constants required in the optimal expression, the algorithm will invariable generate values that do not contribute to the solution. With embedded constants this issue is still present. The underlying problem of generating a constant expression with an optimal value remains. To contrast this with the problem of approximating a base function such as ln, consider that the probability of picking ln from the range of available base functions is approximately $\frac{1}{17}$, depending on the base function set. With a reasonable floating point range the probability of generating a number close to 213.80940889 is several orders of magnitude smaller.

4.7 Parameters

The algorithm has several key parameters. We will list each and detail the challenges in finding an optimal value.

- Population size : A balance between maximizing diversity and information versus optimal fitness requires different optimal population sizing for each problem instance.
- Operator configuration : The operators can restrict bloating by preventing specimens from growing beyond a set limit. The key question remains what the exact valid limit is, and if this excludes optimal solutions from being found.
- Depth limit : The algorithm can force each specimen to maintain a set depth. As before with the operators, this value is at most trial and error. A rule of thumb is to set this value so that at each feature can be present in a leaf in the tree. E.g. with 16 features, depth should be greater or equal to 4. Too large a value will lead to an exponential increase in memory and runtime. Especially validating specimens will become prohibitively expensive.
- Constant ranges: The bounds for the generated constants influence the algorithm greatly. A small range allows for fast execution but risks losing optimal solutions. A large value leads to high variance in fitness values.

- PRNG: the random process and distribution has a significant effect in all stages of the algorithm. For the moment a uniform distribution is used, but experiments are planned with other distributions such a Levy Flight.
- Generations : A simple problem instance will convergence in very few generations, although this also depends on the population size. More intractable problems require more generations. The optimal value is not known in advance, but the algorithm will detect convergence stalling so picking a large value is preferred.
- Runs: From the analysis we clearly see that rerunning the algorithm with an archive of previous generations has a benefit on convergence and quality of solutions. New information is introduced while maintaining the best solutions of the previous runs. There is no ideal values for this parameter, but a similar convergence detection as used with the generations is planned. A higher value will then be preferred.

5 Visualization and Tracing

Each specimen in the population can be inspected by plotting it as a .dot file. Viewing such a tree in svg format allows for quick insights into how the operators work, how the tree is balanced and how it evolves during the algorithm. The convergence statistics gathered during execution can be displayed in interactive plots. This is vital for the analysis of the algorithm, especially the convergence over time and the effect of parameter configuration. Debugging and introspection of a probabilistic search algorithm is notoriously hard. To alleviate this each functioncall can be decorated with a logger, enabling a full trace of the entire algorithm. The code itself requires no functional changes to enable this, simply configuring the right logging object and level is enough.

6 Analysis of results

To analyze the behavior of the algorithm we selected a series of functions from literature. These are characterized as being hard problems for GP inspired SR solutions, and therefore ideal in this project. Each function is approximated by the algorithm in a number of runs. The output of all benchmark functions is saved in "output_{x}.html" files in the current working directory. In this section we will use these results to illustrate behavior of the algorithm.

6.1 Algorithm parameters

Base functions : [+,-,*,/,%,log,ln,exp,sin,cos,tan,tanh,sqrt,pow,abs,min,max]
Generations : 20
Runs : 5
Population : 40

Fitness function : Root mean square multiplied by complexity
Featureset : 5

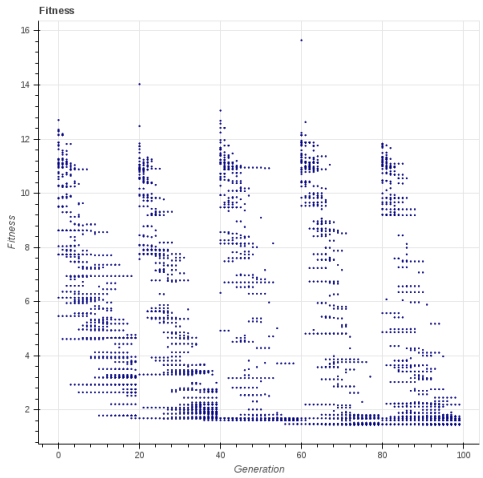
6.2 Benchmark functions

Case 1 The benchmark function

$$y = 0.23 + (14.2 * ((x_3 + x_1)/(3.0 * x_4))$$

is reasonably simple, using only 3 features of the 5 we give the algorithm. In Figure 2 we see that fitness values converge in only 20 generations. Using multiple runs we obtain an increase in fitness by introducing new specimens, but the gain is small.

Figure 2: Fitness values over generations.



The complexity plot in Figure 3 shows that despite the penalty, functions with higher complexity survive more toward the end of the algorithm.

Figure 3: Complexity values over generations.

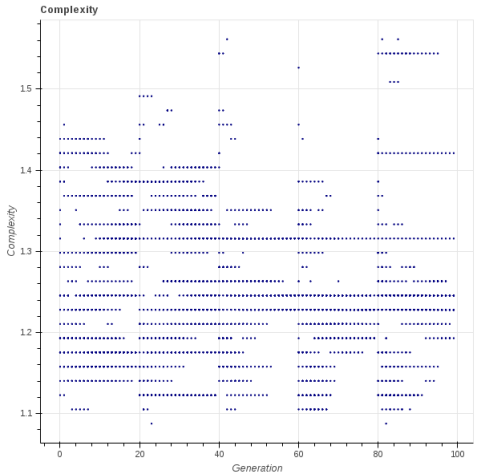
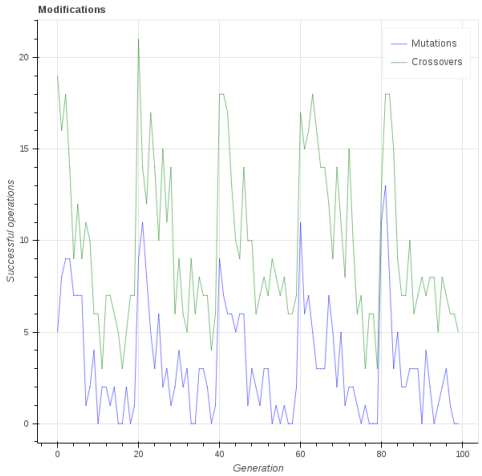


Figure 4 shows for each generation how many operator applications were successful. Success is defined here as the resulting specimen being fitter than its predecessor. As explained before, the mutation is only applied to the least fit half of the population, explaining the factor 2 offset showing in the initial values. Despite this offset, we can also observe that mutations are far less successful as the generations increase. From the previous plots we know that fitness is decreasing (a lower value denotes a better fitness). The explanation for this phenomenon is simple. When a random mutation is applied to a fit individual, the probability of the random subtree being fitter than the subtree it replaces is lower. A similar effect is seen with crossover, to a lesser extent. As the generations proceed, the information that can lead to a fitter individual is distributed over all specimens.

Figure 4: Operator success rate over generations

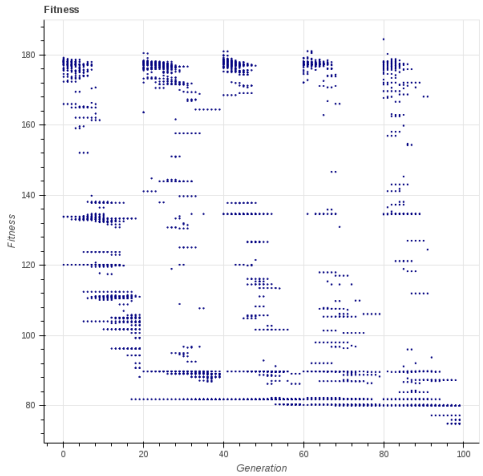


Case 2 The function

$$y = 213.80940889 - (213.80940889 * exp(-0.54723748542 * x_0))$$

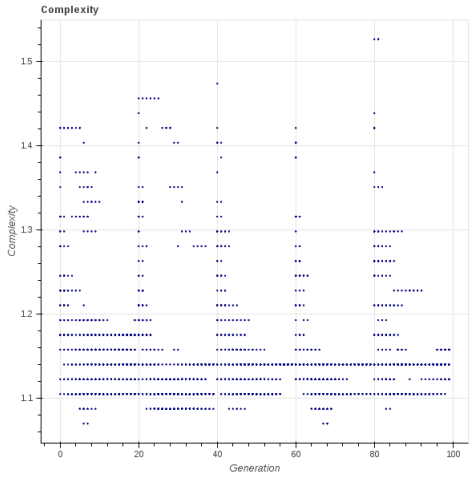
is a more difficult problem for GP, due to the large constants present in the function. Only one feature is used, but the algorithm itself is not given this information, it has to select from 5 as in the previous case. The fitness values converge but to a far higher value than before. The algorithm fails to derive the constants using the existing population. Deriving such a constant is possible if the value is generated as a random value (with very low probability), or if a constant expression is generated that results in this value. The probability that an average random subtree results in this specific value is again very low. It would also require a higher depth, and in this instance the algorithm is limited to depth 4. In the last run we see a sudden gain in fitness. This demonstrates that introducing new information by using several runs can prevent premature convergence. Finally we see that the initially generated specimens have a very low fitness. The exponential term can be a factor in this result, as well as the constant generation problem. This effect repeats itself each run. An interesting approach could be to reduce the number of random specimens when seeding a new run, and mutating the archived solutions instead. Or alternatively, seed the new run by smaller specimens (in this case depth 3 instead of 4). These will still introduce new information to the process, but at a reduced rate and hopefully smaller initial error.

Figure 5: Fitness values over generations.



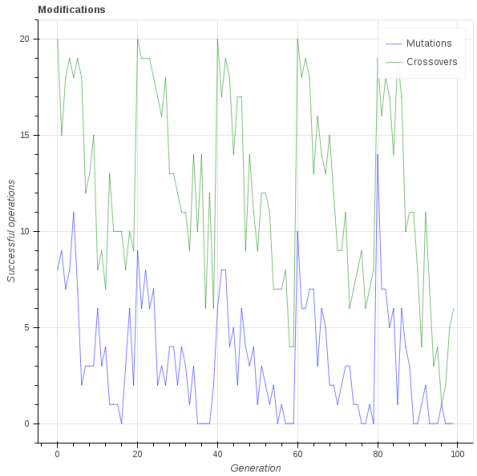
The complexity plot 6 shows that fitter specimens prefer simpler functions. If we look at the given expression we see that once the multiplication and exponentiation are approximated, all that remains are simple base functions. Furthermore, generating the constants using subtrees is more efficient using simple functions. (e.g. sin is less likely to contribute to 213.80 than for instance * or +).

Figure 6: Complexity values over generations.



From the operator activity plot 7 we can see that crossover is far more effective here. Apart from the initial state mutation does not contribute here as much as in the previous case.

Figure 7: Operator success rate over generations



7 Open Issues

7.1 Parameter configuration

It is clear from the results that configuring the algorithm itself requires tuning of optimal parameters. The domain and samples taken from the domain are defining for the results, as are the generations, runs and other parameters involved. It is hard to estimate these values in advance, but the code has been extended with checks to make this process easier. For instance convergence detection is present, if the convergence stalls during k generations the current run is halted and a new begun.

7.2 Complexity measure and Pareto Front

Defining a 'good' complexity measure is still a work in progress. A balance has to be found between computational complexity (e.g. depth of the tree) and the resulting behavior of the tree (e.g. tendency to over and underfitting)

7.3 Planning

Despite some delay in the last weeks of November, the project is on schedule.
The current phase is nearly concluded : implementation and analysis of a baseline GP algorithm with benchmarks
The next phase starts after the exam period : implementation and integration of a selection of metaheuristic algorithms.

8 Supporting Code

8.1 Testing

Extensive tests are available guaranteeing correctness of each part of the project. Considering the probabilistic nature of the algorithms this is required to ensure the results are valid and deterministic (given a seed), leading to reproducible results.

8.2 Expressions

Conversion functions were written to enable parsing expressions from and to tree forms, including but not limited to a simple expression parser and tokenizer, and a number of utility functions.

8.3 Visualization

The expression trees can be written out to .dot format. Such a graphic visualization has proven to be invaluable in debugging the code as well as analyzing the behavior of the algorithm. The statistics gathered during the execution of the algorithm are output as interactive plots. This representation allows analysis of the convergence of the algorithm in time. This functionality will become even more useful in the remainder of the project.

8.4 Constants

To make the integration of the metaheuristic optimizers easier, the design has integrated a set of accessors retrieving all Constant objects from an expression tree in vector form. This provides the optimization algorithm a quick interface to the tree structure.