Comparing Genetic Programming Approaches for Non-Functional Genetic Improvement

Case Study: Improvement of MiniSAT's Running Time

Aymeric Blot Justyna Petke

University College London, UK

UK EPSRC grant EP/P023991/1

EuroGP (EvoStar) — 15 April 2020

Genetic Improvement (GI)

Automated software improvement:

- Program repair / bug fixing
- Feature transplantation
- Running time
- Memory/energy consumption

Non-functional GI in practice:

- Start from original software
- Accumulate sequences of edits
- Deletion/replacement/insertion
- Lines/statements/data

Petke et al., IEEE Transactions on Evolutionary Computation, 2018

Non-Functional GI So Far: Success Stories

Non-functional GI literature usually:

- Focuses on software and final improvements
- Fine tunes GI approach to the application
- Only reports positive results

Motivation: focus on the evolutionary process

Focus on the Evolutionary Process

Case study:

- Pre-existing GI scenario: MiniSAT
- Running time $\rightarrow$ CPU instructions
- Eight GP approaches; four random approaches
- k-fold cross-validation

Research Questions:

- Effectiveness? (*how often*)
- Efficiency? (*how good*)
- Robustness? (*how sensible to parameters*)
- Consistency? (*impact of data*)

Experimental Protocol

Training:

- To find improved software variants
- Using the search process (GP)
- Until budget exhaustion

Validation:

- To avoid overfitting
- Filter out potentially harmful mutations

Test:

- To assess generalisation

Experimental Protocol

Some issues in some previous GI work:

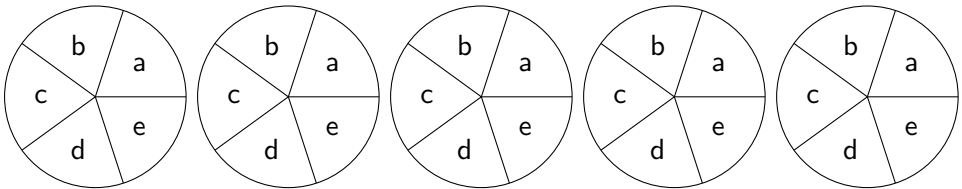
- Report a single GI run
- Do not report intermediary results
- Reuse training data in validation and test steps
- Use a single random data split
- Use different types of data between steps

k-fold cross-validation:

- Report k GI runs
- Use disjoint data on three steps
- Assess generalisation on the same type of data

Cross-validation ($k = 5$)

Data is separated into k disjoint *"folds"*
Then labelled in k different ways:



Test: (X)

- Single fold
- Sequentially

Validation: (V)

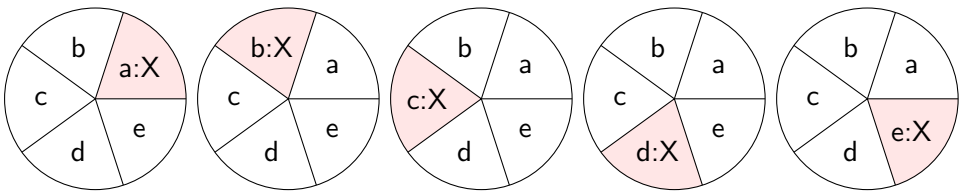
- Single fold
- Uniform at random

Training: (T)

- $k - 2$ folds
- All remaining

Cross-validation ($k = 5$)

Data is separated into k disjoint *"folds"*
Then labelled in k different ways:



Test: (X)

- Single fold
- Sequentially

Validation: (V)

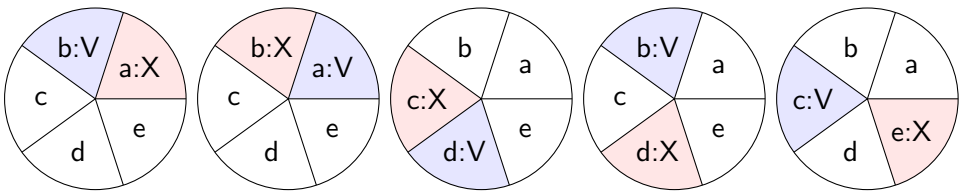
- Single fold
- Uniform at random

Training: (T)

- $k - 2$ folds
- All remaining

Cross-validation ($k = 5$)

Data is separated into k disjoint *"folds"*
Then labelled in k different ways:



Test: (X)

- Single fold
- Sequentially

Validation: (V)

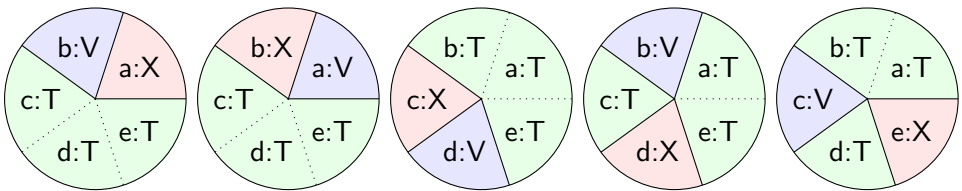
- Single fold
- Uniform at random

Training: (T)

- $k - 2$ folds
- All remaining

Cross-validation ($k = 5$)

Data is separated into k disjoint *"folds"*
Then labelled in k different ways:



Test: (X)

- Single fold
- Sequentially

Validation: (V)

- Single fold
- Uniform at random

Training: (T)

- $k - 2$ folds
- All remaining

Training: Random Search (Baseline), GP, GP_e

Rand(m): with $m = 1, 2, 5, 10$

- Generate sequences of up to m mutations
- Independent; uniformly at random

GP(n): with $n = 10, 20, 50, 100$

- Population: fixed size n
- Initialisation: single random mutation
- Offspring: 50% crossover, 50% mutation

GP_e(n): **(new)** with $n = 10, 20, 50, 100$

- GP(n) with elitism
- Best 10% forwarded ($+$ 45% crossover, 45% mutation)

Genetic Programming Main Loop

Selection:

- Filter invalid individuals and sort by fitness

Elitism: (new)

- Forward best p_e individuals to offspring

Crossover:

- Select best p_c individuals, 1-point crossover with a random parent

Mutation:

- Select best p_m individuals, append a random mutation

Regrow:

- If not enough offspring, add new random individuals of size 1

After every generation, update the fitness function

Validation: Filtering

First pass: (new)

- Sequentially remove edits with no impact
- To reduce size of edit sequences and shorten the second pass

Second pass:

- Evaluate every edit independently
- Sort them by fitness
- Sequentially re-add them, keep if improving

Experimental Setup

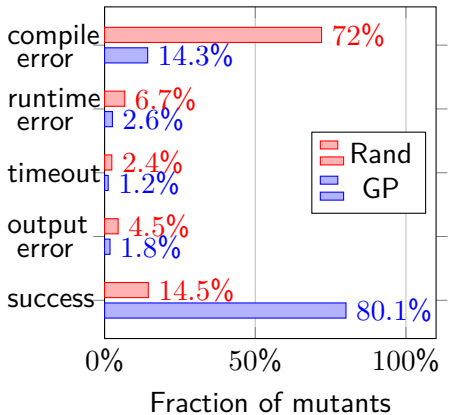
MiniSAT: (http://minisat.se/)

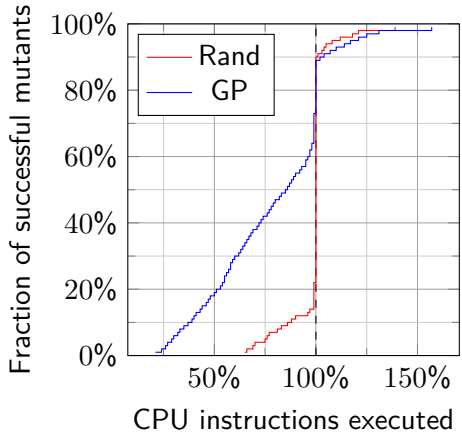
- Award winning SAT solver, still relevant today
- Designed to be simple, modular, and extensible.
- `minisat2-070721` (2007), `minisat-2.2.0` (2008, latest version)
- Search-related code in a single `C++` file (428 AST nodes)

12 search processes: Rand $\times 4$, GP $\times 4$, GP_e $\times 4$

130 CIT instances: from previous GI work

Training Overall Analysis





Training budget: 10000 SAT instances

Average execution time: Rand: 2 hours << GP: 10 hours

Experimental Results (Fold 1)

Training			Validation				Test
Search	Size	CPU	Size'	CPU	Size*	CPU*	CPU*
$Rand(1)$	1	66.5%	1	114.0%	0	—	—
$Rand(2)$	2	67.0%	2	114.5%	0	—	—
$Rand(5)$	1	75.0%	1	109.0%	0	—	—
$Rand(10)$	2	74.9%	2	107.2%	1	100.0%	100.0%
$GP(10)$	16	99.9%	11	99.9%	7	99.9%	99.9%
$GP(20)$	32	92.7%	12	123.4%	5	93.5%	67.4%
$GP(50)$	23	69.6%	11	102.6%	3	99.4%	99.6%
$GP(100)$	16	63.8%	13	111.3%	4	99.9%	99.9%
$GP_e(10)$	1304	33.5%	26	114.4%	13	90.8%	62.8%
$GP_e(20)$	268	57.7%	21	105.5%	4	91.0%	63.0%
$GP_e(50)$	15	78.2%	7	123.6%	5	96.7%	98.5%
$GP_e(100)$	6	64.8%	6	107.1%	2	100.0%	100.0%

Experimental Results (Fold 4)

Training			Validation				Test
Search	Size	CPU	Size'	CPU	Size*	CPU*	CPU*
$Rand(1)$	1	57.4%	1	77.2%	1	77.2%	122.8%
$Rand(2)$	1	77.1%	1	75.4%	1	75.4%	92.0%
$Rand(5)$	3	57.7%	3	99.9%	1	99.8%	96.1%
$Rand(10)$	1	77.1%	1	75.4%	1	75.4%	92.0%
$GP(10)$	26	93.8%	9	91.6%	6	91.6%	126.9%
$GP(20)$	54	22.2%	13	55.0%	6	50.2%	124.7%
$GP(50)$	9	82.8%	7	91.0%	6	54.0%	115.8%
$GP(100)$	7	57.8%	5	75.4%	3	75.4%	92.0%
$GP_e(10)$	2	99.8%	2	99.9%	2	99.9%	99.8%
$GP_e(20)$	49	22.2%	9	54.9%	8	49.8%	123.8%
$GP_e(50)$	6	82.8%	6	99.7%	4	99.7%	130.6%
$GP_e(100)$	10	48.9%	9	119.6%	5	50.1%	124.7%

Results Overview

GP as search process:

- Much more successful than random search
- Not very parameter-sensitive
- Large overfits

Repeated experiments:

- Very variable results
- Highly heterogeneous dataset

Research Questions

Effectiveness: (*how often*)

- $> 5\%$ after *training*: almost always
- $> 5\%$ after *either validation or test*: half of the time
- $> 5\%$ after *validation AND test*: only $5/40$ GP, $2/20$ Rand

Efficiency: (*how good*)

- Down to 36% CPU instructions (64% faster) on some unseen folds
- Two-third of improvements $> 25\%$ (validation or test)

Robustness: (*how sensible to parameter*)

- Inconclusive (due to dataset?)

Consistency: (*impact of data*)

- Inconclusive as revealed by protocol

Conclusion

What we did:

- Re-used existing GI scenario
- Much more rigorous experimental protocol

What we obtained:

- Consistent results for fixed data
- Inconsistent results when controlling data
- Some very good mutants

What we learned:

- Many potential hidden flaws
- Controlling data is essential
- Potential for better approaches

Final Words

Take-home message:

- GI exists, and GI works!
- But it can work better!
- Success stories $\rightarrow$ standardisation
- First step towards future investigation

Selected References

Niklas Eén and Niklas Sörensson.

An extensible SAT-solver.

In *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518.

Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David Robert White, and John R. Woodward.

Genetic improvement of software: A comprehensive survey.

IEEE Transactions on Evolutionary Computation, 22(3):415–432, 2018.