An Unsystematic Review of Genetic Improvement

David R. White
University of Glasgow
UCL Crest Open Workshop, Jan 2016
A Systematic Study of GI

...is currently under preparation.

This presentation is based on a “snapshot” of this work.
Review Unmethod

1. Working list of ~ 300 papers.

2. Reduced to 150 by eliminating those I didn’t think would be relevant to my view of GI.

3. Cherry-picked meta-papers, significant papers for my definition of significant, controversial papers.

4. Read. A lot.

5. Summary stats and observations.
Outline

1. Trends.
2. Progress in the field.
3. Challenges.
4. New Directions.
Outline

1. Trends.

2. Progress in the field.

3. Challenges.

4. New Directions.
GI Papers by Year

GI Workshop

GenProg

Paragen

SEBASE Project
Earliest GI

ParaGen System

Paul Walsh, Conor Ryan, ParGen1995.

Formally verifiable results.
Bug-Fixing vs Non-Functional
Target Language

- C
- C++
- Java
- Other
- No Language
Outline

1. Summary statistics.
2. Progress in the field.
3. Challenges.
4. New Directions.
void sort(int* a, int length) {
    for (; 0 < (length - 1); length--) {
        for (int j = 0; j < (length - 1); j++) {
            if (a[j] > a[1 + j]) {
                k = a[j];
                a[j] = a[j + 1];
                a[1 + j] = k;
            }
        }
    }
}
2009: The Zune Bug

void year_from_days_since_1980(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)){
            if (days > 366) {
                days -= 366;
                year += 1;
            } else {
            }
        } else {
            days -= 365;
            year += 1;
        }
    }
    printf("current year is %d\n", year);
}

Try input days = 366.

Drops into empty else.

2009: The Zune Bug

```c
void year_from_days_since_1980(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                //days -= 366; // repair deletes
                year += 1;
            } else {
            }
        } else {
            days -= 365; // repair inserts
            year += 1;
        }
    }
    printf("current year is %d\n", year);
}
```

Awards

GECCO Human-Competitive Awards

ICSE Distinguished Paper Award
Code 'transplant' could revolutionise programming

Code has been automatically “transplanted” from one piece of software to another for the first time, with researchers claiming the breakthrough could radically change how computer programs are created.

The process, demonstrated by researchers at University College London, has been likened to organ transplantation in humans. Known as MuScalpel, it works by isolating the code of a useful feature in a ‘donor’ program and transplanting this “organ” to the right “vein” in software lacking the feature. Almost all of the transplant is automated, with minimal human involvement.

Automated transplants of features between apps could free human programmers from tedious, manual work and make developing software faster and cheaper.

“As any programmer will attest, a large amount of programming work consists of this kind of manual transplantation work; redesigning, implementing and reinventing functionality that already exists in some form on some other system.” Mark Harman, head of software systems engineering at UCL tells WIRED.co.uk. “By automating it, we make it much faster and cheaper.”
Progress: Representation

Originally, GI directly manipulated the AST as with traditional GP.

Most work uses a patch-based representation. [1, 2]

Empirical evidence that patches are more effective [3].


Traditional GP Search Space

\[ c(d) = \begin{cases} 
  n_0 & \text{if } d = 1 \\
  \max_{a=0} \sum_{a=1}^{n_a} c(d-1)^a & \text{if } d > 1 
\end{cases} \]

- \( c(d) \) is the number of possible trees of depth \( d \).
- \( d \) is the maximum tree depth.
- \( n_0 \) the number of terminals in the function set.
- \( n_a \) is the number of functions of arity \( a \).
Patch Search Space

\[ O(m^{k^2}) \]

\( m \) is the number of operators.
\( k \) is the number of program nodes.
Progress: Fault Localisation

GenProg: simple statistical approach.

Reduce the value of $k$ by focusing on instructions executed by the failing test cases.

Other work has used program counter sampling with Gaussian Convolution! [1]

Applications

Targeting GPGPU.


Objectives

**Bug-fixing**

**Execution time**

**Power consumption**

**Memory**
A Methodology to Automatically Optimize Dynamic Memory Managers Applying Grammatical evolution. Risco-Martína et al. JSS 91, 109-123.

**Diversity**
Objectives Cont.

**Parallelisation**
Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System. Walsh and Ryan. ParCo '95.

**Translation**
Evolving a CUDA kernel from an nVidia template. Langdon. CEC 2010.

**Simplification**
Genetic programming for shader simplification. Sitthi-amorn et al. SA '11.

**Extending Functionality**
Outline

1. Summary statistics.
2. Progress in the field.
3. Challenges.
4. New Directions.
Empirical Method

There has been criticism of existing work, including GenProg [1].

Biggest issue is overfitting.

*Observation*: Many papers focus on statistics rather than code. We need more qualitative examination of results. There is not much code in the papers!


Correctness

Generate and Validate Approaches have problems with overfitting.

They may also break more functionality than they fix.

Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. Smith et al. ESEC/FSE 2015.
Credibility

Also a wider issue for GP and EC.

Evolution is self-evidently a powerful general learning method.

Are programs a suitable subject? Evidence they are surprisingly robust [1], and neutral networks exist [2].

Theory

Limited work on the theoretical side.

Some work on “mutational robustness”.

Where is the landscape analysis?
Outline

1. Summary statistics.
2. Progress in the field.
3. Challenges.
4. New Directions.
Hardware Measurement

Simulators are out, physical measurements are in.

Binaries

Bug-fixing and program thinning without source code.

Interesting problem: harder to intuit whether a patch is “correct”.

Removing the Kitchen Sink from Software. Lanesborough et al. GECCO 2015.


Deep Parameter Optimisation

Optimise parameters within code rather than code itself: thus possess an oracle.

Use mutation testing to determine impact of parameters.

Transform non-functional properties. Target key components, e.g. malloc [1].

Also a suitable target for Amortised Optimisation. [2]


Code Transplantation

Extend functionality of existing code by importing features from other codebases.

Process of locating a transplantation point; extracting relevant code; finding correct bindings; minimisation; passing regression tests.

Semantic Search

Index and search large code repositories: it’s about existing code.

Find patches with similar semantics expressed in SMT.

Much higher success rate in producing repairs that generalise.

Summary

We’re at a crucial juncture for GI Research.

We must address empirical method problems.

Solid theory or systematic empirical methods required. Qualitative, explanatory, evidence is key.

Wide range of possible avenues for work: many circumvent objections and address challenging problems.
Call For Papers

The Second International Genetic Improvement Workshop (GI-2016) will be held in Denver, Colorado, during the Genetic and Evolutionary Computation Conference (GECCO-2016).

Key Dates

| Paper submission deadline: | April 2, 2016 |
| Authors notification:     | April 19, 2106 |

Website: [http://geneticimprovementofsoftware.com](http://geneticimprovementofsoftware.com)

We invite submissions that discuss recent developments in all areas of research on, and applications of, Genetic Improvement. The workshop also provides an opportunity for researchers interested in GI to exchange ideas and find out about current research directions in the field and receive guidance on the application of GI to their problem domain. Topics of interest include, but are not limited to, using genetic improvement to automatically:

- fix bugs
- improve efficiency
- decrease memory consumption
- decrease power consumption
- transplant new functionality
- specialise software