Genetic improvement of software: a case study

Justyna Petke

Centre for Research on Evolution, Search and Testing
Department of Computer Science, UCL, London
Genetic Improvement Programming

- Automatically improving a system's behaviour with respect to some desired criteria using Genetic Programming
- The criteria for improvement can be non-functional properties of the system, such as execution time
- Relies on a set of test cases, obtained from running the original system
- Genetic Programming tries many possible options, leaving software designer to choose between best
Bowtie2 is one of the tools used in processing DNA sequences generated by next-generation DNA sequencing machines.

▸ 50,000 lines of C++
▸ over 50 main system modules and 67 header files
▸ focused GP search on 2744 heavily used lines
Results

- Wanted to trade-off performance v. speed:
  - On “1000 genome” nextgen DNA sequences
  - 70+ faster on average
  - Very small improvement in Bowtie2 results
- Only 7 lines of code changed in 3 C++ files
Motivation

Try another example

▶ Easy to analyse
▶ Popular
▶ (Competition)
Software chosen

Example well-known SAT solver: MiniSAT

*Boolean satisfiability problem (SAT)*
is the problem of deciding whether there is a variable assignment
that satisfies a given propositional formula.
SAT solver Applications

- Bounded Model Checking
- Planning
- Software Verification
- Automatic Test Pattern Generation
- Combinational Equivalence Checking
- Combinatorial Interaction Testing
- and many others..
Representation of the System to be Evolved

- Source code
- Grammar used to constrain changes (syntactically valid)
  - more chance of compiling
  - thus high chance of running
  - timeouts to force termination
Representation: Move operations

- Change code by re-using existing human written code
  - Copy a line
  - Replace a line with another line from the program
  - Delete a line
- Evolve a list of changes
- Grammar rule: a line of code or a part of loop/condition (for, if, while, else)
BNF grammar

< Solver_135 > ::= "{Log_count64++;/*135*/} if" < IF_Solver_135 > " return false;\n"
< IF_Solver_135 > ::= "(!ok)"
< Solver_138 > ::= "" < _Solver_138 > "{Log_count64++;/*138*/}\n"
< _Solver_138 > ::= "sort(ps);"
< Solver_139 > ::= "Lit p; int i, j;\n"
< Solver_140 > ::= "for(" < for1_Solver_140 > ";" < for2_Solver_140 > ";" < for3_Solver_140 > ") \n"
< for1_Solver_140 > ::= "i = j = , p = lit_Undef"
< for2_Solver_140 > ::= "i < ps.size()"
< for3_Solver_140 > ::= "i++"
Representation: Combining moves

- Mutation: append another random change to the list
- Crossover: append lists from two parents
- Only creating a new individual shortens the list
Fitness function

- Run program and count lines used
- 2 measures:
  - Quality of answers produced (right/wrong, automatic oracle)
  - Resources used (number of lines used)
GP Improvement

Original code

BNF Grammar

Test cases

Improved system

Population of modifications

Mutation and Crossover

Fitness

Modified code

Population of modifications
MiniSAT

- SAT solver
- 16 header files, 6 C++ files (core solving algorithm in Solver.cc)
- of the 582 lines of C++ code in Solver.cc file, BNF produces 321 lines that genetic programming can manipulate (delete, replace, insert)
GP evolution parameters

- training data set size: 71
- population size: 20
- generations: 100
- 50% crossover
- 50% mutation (delete, replace, insert)
- selection (top half)
- 5 test examples, reselected every generation
Results

- around 14 hours
- around 73% compiled
- no clear winner so far..
- mainly stats and optimisations removed
SAT example

\[ x_1 \lor x_2 \lor \neg x_4 \]
\[ \neg x_2 \lor \neg x_3 \]

- \( x_i \) : a Boolean variable
- \( x_i, \neg x_i \) : a literal
- \( \neg x_2 \lor \neg x_3 \) : a clause
Example

```cpp
bool Solver::satisfied(const Clause& c) const {
    for (int i = 0; i < c.size(); i++){
        if (value(c[i]) == l_True){
            return true;
        }
    }
    return false;
}
```
bool Solver::satisfied(const Clause& c) const {
    for (int i = 0; ; i++){
        if (value(c[i]) == l_True){
            return true;
        }
    }
    return false;
}
Research directions

- specialise test sets for GP
- include pre-processing
- change population and generation size
- try to discover historical changes using an older version of the solver
Summary

- Genetic Improvement Programming automatically improves system behaviour according to some desired criteria using GP
- Bowtie2: 70+ runtime improvement
- MiniSAT: ?