Automatic Program Repair using Evolutionary Computing

ThanhVu Nguyen

November 3, 2014
The Problem

SOFTWARE BUGS
The Cost

– Mozilla Developer

“Everyday, almost 300 bugs appear [...] far too many for only the Mozilla programmers to handle.”

Software bugs annually cost 0.6% of the U.S GDP and $312 billion to the global economy.

Average time to fix a security-critical error: 28 days
How do we repair bugs now?

- Ignore them
- Pay expensive programmers to fix them manually
- Develop tools to help the programmers
  - Debuggers, profilers, smart compilers
  - Type checkers
- Build mathematical models to analyze program correctness
  - Don’t scale up to production software
GenProg: Evolutionary Program Repair

- A generic method for automated software repair
- Uses genetic algorithm
- Works with legacy code
- Does not assume formal specifications
Darwinian Evolution in a Computer

Genetic Algorithm
- Developed by John Holland (1975)
- A search heuristic that mimics the process of natural evolution
- Uses concepts of *natural selection and genetic inheritance* (Darwin 1859)

Widely-used in business, science, and engineering
- Optimization and search problems
- Routing, scheduling, and timetabling
- Evolvable hardware, encryption, robotics
Generate initial population of solutions

Evaluate fitness of each solution

Selection of individual solutions

Mating (reproduction)

Mutation

New population generated and fitness evaluated

Sufficient solution quality or maximum search terms reached

Yes
End

No
Example: maximize the number of ones in a bit string

1. **Encoding** of an individual: as a bit vector, e.g., $i = 111010101$

2. **Initial population**: create $n$ random individuals
   
   $\text{pop} = \{i_1 = 111010101, i_2 = 011000101, i_3 = 001000100, \ldots, i_n = 001011011\}$

3. **Fitness**: the number of 1's in an individual
   
   $f(i_1) = 7, f(i_2) = 5, f(i_3) = 2, \ldots, f(i_n) = 6$

4. **Selection**: choose $n$ individuals based on their fitnesses
   
   $\text{pop}' = \{i_1, i_n, i_1, i_2, \ldots\}$

5. **Reproduction**
   - Select each pair $m, n \in \text{pop}'$ with probability $p_{\text{xover}}$
   - Crossover $m$ and $n$ at a randomly selected point, i.e., 1-point xover
     
     111010101 and 0010110111 give 1110110111 and 0011010101

6. **Mutation**
   - select each individual $i$ with probability $p_{\text{mut}}$
   - flip the bit at a randomly selected point
     
     0010001000 $\rightarrow$ 0011001000
Given a program
  - C source code
  - .. and evidence of a bug
    - test suite consisting of passed (positive) and failed (negative) test cases
  - .. fix that bug using genetic algorithm
    - returns a textual patch
Fault Localization

- In a large program, not every line is equally likely to contribute to the bug.
- **Insight**: since we have the test cases, run them and collect coverage information.
- The bug is more likely to be found on lines visited when running the failed test case.
- The bug is less likely to be found on lines visited when running the passed test cases.
Define a weighted path to be a list of \( \langle \text{statement}, \text{weight} \rangle \) pairs.

Statements in weighted path:
- The statements are those visited during the failed test case.
- The weight for a statement \( S \) is
  - High (1.0) if \( S \) is not visited on a passed test
  - Low (0.1) if \( S \) is also visited on a passed test
Genetic Representation and Operations

- **Population**: each individual is an AST of the program
- **Mutation**: select a statement $S \in AST$ biased by the weight of $S$
  - Insert $S_1$ after $S$, delete $S$, replace $S$ with $S_2$
  - Choose $S_1$ and $S_2$ from the entire AST

![Mutation Diagram]

- **Crossover**: 1-point crossover
- **Insight**
  - don’t invent new code
  - assume program contains the seeds of its own repair (e.g., has another null check elsewhere)
Fitness Evaluation

- Take in a program source $P$ to be evaluated
- Compile $P$ to an executable program $P'$
  - If cannot compile, assign fitness 0
- Fitness score of $P'$: weighted sum of test cases that $P'$ passes
  \[ f(P') = \# \text{ pos pass} \times W_{pos} + \# \text{ neg pass} \times W_{neg} \]
  \[ W_{pos} = 1 \text{ and } W_{neg} = 10 \]
- If $P'$ passes all test cases, then $P$ is a solution candidate
- Note: the original (buggy) program passes all positive test cases
/ * requires: a >= 0, b >= 0 */
void print_gcd(int a, int b) {
  if (a == 0) {
    printf("%d", b);
  }
  while (b != 0) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
    printf("%d", a);
  }
  exit(0);
}

**Bug:** it loops forever when 
\textit{a}=0 \textit{and b}>0
\textit{e.g., a}=0, \textit{b}=55

Example: GCD
Abstract Syntax Tree

```
AST of print_gcd
```

```
{ block }
    if (a==0)
        { block }
        printf(... b)
    { block }
    while (b != 0)
        { block }
        if (a > b)
            { block }
            a = a - b
        { block }
        b = b - a
        printf(... a)
    return
```
Weighted Path: Negative Test case

Nodes (stmts) visited on a negative test case, e.g., $a=0, b=55$
Weighted Path: Positive Testcase

Nodes visited on a positive test case \((a=5, b=15)\)
Weighted Path

Concentrate on nodes visited on negative, but not, positive test cases
Randomly pick a node and insert after another node


```c
void print_gcd(int a, int b) {
    if (a == 0) {
        printf("%d", b);
        return; //repair insert
    }
    while (b != 0)
        if (a > b)
            a = a - b;
        else
            b = b - a;
    printf("%d", a);
    return;
}
```
Minimize Repair

- Repair Patch is a diff between orig and variant
- Mutations may add unneeded statements (e.g., dead code, redundant computation)
- In essence: try removing each line in the diff and check if the result still passes all tests
- Delta Debugging finds a 1-minimal subset of the diff in $O(n^2)$ time
- Removing any single line causes a test to fail
- We use a tree-structured diff algorithm (diffX)
- Avoids problems with balanced curly braces, etc.
Example: Zunebug

- Dec. 31, 2008. Microsoft Zune players freeze up
- **Bug**: infinite loop when input is last day of a leap year
- Negative test case: 10593 (Dec 31, 2008)
- Repair is not trivial

```c
int zunebug(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
    }
    return year;
}
```
if (days > 366) {
    days -= 366;
    if (days > 366) { // insert #1
        days -= 366; // insert #1
        year += 1; // insert #1
    } // insert #1
    year += 1;
} // insert #1

days -= 366; // insert #2

Pass negative test cases, but fail some positive test cases.
Final Repair

```c
int zunebug(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
    }
    return year;
}
```

```c
int zunebug_repair(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                // repair deletes
                days -= 366;
                year += 1;
            }
            // repair inserts
            days -= 366;
        } else {
            days -= 365;
            year += 1;
        }
    }
    return year;
}
```

- Final repair produced in 42 seconds
- One of the several possible repairs
Evolution of Repair

![Graph showing the evolution of repair with generations. The graph plots fitness against generation. There are two lines: one for the average and another for the best individual. The fitness increases over generations, with a significant jump from generation 6 to generation 7.](image-url)
## Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>example</td>
<td>22</td>
</tr>
<tr>
<td>uniq</td>
<td>duplicate text processing</td>
<td>1146</td>
</tr>
<tr>
<td>ultrix look</td>
<td>dictionary lookup</td>
<td>1169</td>
</tr>
<tr>
<td>svr4 look</td>
<td>dictionary lookup</td>
<td>1363</td>
</tr>
<tr>
<td>units</td>
<td>metric onversion</td>
<td>1504</td>
</tr>
<tr>
<td>deroff</td>
<td>document processing</td>
<td>2236</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>web server</td>
<td>5575</td>
</tr>
<tr>
<td>indent</td>
<td>source code processing</td>
<td>9906</td>
</tr>
<tr>
<td>flex</td>
<td>lex analyzer generator</td>
<td>18775</td>
</tr>
<tr>
<td>atri6</td>
<td>graphical Tetris game</td>
<td>21553</td>
</tr>
</tbody>
</table>

**average**

Initial result (ICSE '09): over 63,000 lines of code, 10 bugs

Current result: over 5 Million LoC, 100+ bugs in php, python, gzip, etc
# Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LoC</th>
<th>WPath</th>
<th>Time(s)</th>
<th>Success</th>
<th>Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>example</td>
<td>22</td>
<td>1.3</td>
<td>149</td>
<td>54%</td>
<td>inf loop</td>
</tr>
<tr>
<td>uniq</td>
<td>duplicate text processing</td>
<td>1146</td>
<td>81.5</td>
<td>32</td>
<td>100%</td>
<td>segfault</td>
</tr>
<tr>
<td>ultrix look</td>
<td>dictionary lookup</td>
<td>1169</td>
<td>213.0</td>
<td>42</td>
<td>99%</td>
<td>segfault</td>
</tr>
<tr>
<td>svr4 look</td>
<td>dictionary lookup</td>
<td>1363</td>
<td>32.4</td>
<td>51</td>
<td>100%</td>
<td>inf loop</td>
</tr>
<tr>
<td>units</td>
<td>metric onversion</td>
<td>1504</td>
<td>2159.7</td>
<td>107</td>
<td>7%</td>
<td>segfault</td>
</tr>
<tr>
<td>derooff</td>
<td>document processing</td>
<td>2236</td>
<td>251.4</td>
<td>129</td>
<td>97%</td>
<td>segfault</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>web server</td>
<td>5575</td>
<td>768.5</td>
<td>502</td>
<td>36%</td>
<td>buffer overrun</td>
</tr>
<tr>
<td>indent</td>
<td>source code processing</td>
<td>9906</td>
<td>1435.9</td>
<td>533</td>
<td>7%</td>
<td>inf loop</td>
</tr>
<tr>
<td>flex</td>
<td>lex analyzer generator</td>
<td>18775</td>
<td>3836.6</td>
<td>233</td>
<td>5%</td>
<td>segfault</td>
</tr>
<tr>
<td>abris</td>
<td>graphical Tetris game</td>
<td>21553</td>
<td>34.0</td>
<td>69</td>
<td>82%</td>
<td>buffer overrun</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initial result (ICSE '09): over 63,000 lines of code, 10 bugs

Current result: over 5 Million LoC, 100+ bugs in php, python, gzip, etc
## Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LoC</th>
<th>WPath</th>
<th>Time(s)</th>
<th>Success</th>
<th>Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>example</td>
<td>22</td>
<td>1.3</td>
<td>149</td>
<td>54%</td>
<td>inf loop</td>
</tr>
<tr>
<td>uniq</td>
<td>duplicate text processing</td>
<td>1146</td>
<td>81.5</td>
<td>32</td>
<td>100%</td>
<td>segfault</td>
</tr>
<tr>
<td>ultrix look</td>
<td>dictionary lookup</td>
<td>1169</td>
<td>213.0</td>
<td>42</td>
<td>99%</td>
<td>segfault</td>
</tr>
<tr>
<td>svr4 look</td>
<td>dictionary lookup</td>
<td>1363</td>
<td>32.4</td>
<td>51</td>
<td>100%</td>
<td>inf loop</td>
</tr>
<tr>
<td>units</td>
<td>metric onversion</td>
<td>1504</td>
<td>2159.7</td>
<td>107</td>
<td>7%</td>
<td>segfault</td>
</tr>
<tr>
<td>deroff</td>
<td>document processing</td>
<td>2236</td>
<td>251.4</td>
<td>129</td>
<td>97%</td>
<td>segfault</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>web server</td>
<td>5575</td>
<td>768.5</td>
<td>502</td>
<td>36%</td>
<td>buffer overrun</td>
</tr>
<tr>
<td>indent</td>
<td>source code processing</td>
<td>9906</td>
<td>1435.9</td>
<td>533</td>
<td>7%</td>
<td>inf loop</td>
</tr>
<tr>
<td>flex</td>
<td>lex analyzer generator</td>
<td>18775</td>
<td>3836.6</td>
<td>233</td>
<td>5%</td>
<td>segfault</td>
</tr>
<tr>
<td>atris</td>
<td>graphical Tetris game</td>
<td>21553</td>
<td>34.0</td>
<td>69</td>
<td>82%</td>
<td>buffer overrun</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58.7%</td>
</tr>
</tbody>
</table>

- Initial result (ICSE ’09): over 63,000 lines of code, 10 bugs
## Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LoC</th>
<th>WPath</th>
<th>Time(s)</th>
<th>Success</th>
<th>Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>example</td>
<td>22</td>
<td>1.3</td>
<td>149</td>
<td>54%</td>
<td>inf loop</td>
</tr>
<tr>
<td>uniq</td>
<td>duplicate text processing</td>
<td>1146</td>
<td>81.5</td>
<td>32</td>
<td>100%</td>
<td>segfault</td>
</tr>
<tr>
<td>ultrix look</td>
<td>dictionary lookup</td>
<td>1169</td>
<td>213.0</td>
<td>42</td>
<td>99%</td>
<td>segfault</td>
</tr>
<tr>
<td>svr4 look</td>
<td>dictionary lookup</td>
<td>1363</td>
<td>32.4</td>
<td>51</td>
<td>100%</td>
<td>inf loop</td>
</tr>
<tr>
<td>units</td>
<td>metric onversion</td>
<td>1504</td>
<td>2159.7</td>
<td>107</td>
<td>7%</td>
<td>segfault</td>
</tr>
<tr>
<td>deroff</td>
<td>document processing</td>
<td>2236</td>
<td>251.4</td>
<td>129</td>
<td>97%</td>
<td>segfault</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>web server</td>
<td>5575</td>
<td>768.5</td>
<td>502</td>
<td>36%</td>
<td>buffer overrun</td>
</tr>
<tr>
<td>indent</td>
<td>source code processing</td>
<td>9906</td>
<td>1435.9</td>
<td>533</td>
<td>7%</td>
<td>inf loop</td>
</tr>
<tr>
<td>flex</td>
<td>lex analyzer generator</td>
<td>18775</td>
<td>3836.6</td>
<td>233</td>
<td>5%</td>
<td>segfault</td>
</tr>
<tr>
<td>atris</td>
<td>graphical Tetris game</td>
<td>21553</td>
<td>34.0</td>
<td>69</td>
<td>82%</td>
<td>buffer overrun</td>
</tr>
</tbody>
</table>

| average     |                           | 881.4| 184.7 | 58.7%   |         |                  |

- Initial result (ICSE ’09): over 63,000 lines of code, 10 bugs
- Current result: over 5 Million LoC, 100+ bugs in php, python, gzip, etc
Why does it succeed?

- Most bugs are small
- Only reuses existing statements
- Algorithmic innovations
  - Start with an (almost) working program
  - Weighted path greatly reduces search space
  - Minimization eliminates unnecessary fixes
Repair Quality

- Repairs are typically *not* what a human would have done
  - Example: GenProg adds bounds checks to one particular network read, rather than refactoring to use a safe abstract string class in multiple places

- Recall: any proposed repair must pass all test cases
  - When POST test is omitted from `nullhttpd`, the generated repair eliminates POST functionality
  - Tests ensure GenProg do not sacrifice functionality
  - Minimization prevents gratuitous deletions
  - Adding more tests helps rather than hurting
Limitations

- May not handle nondeterministic faults
  - Difficult to test for race conditions, etc.
  - Long term: put scheduler constraints into the individual representation.
- Assumes bug test case visits different lines than normal test cases
- Assumes existing statements can form repair
  - Current work: repair templates
  - Hand-crafted and mined patterns and specifications from CVS repositories
- No formal correct guarantee (may delete functionalities not specified in test suites)
The growth of automatic program repair

- 2009 (a banner year): 15 papers on auto program repair
  - **GenProg**: evolves source code until it passes the rest of a test suite. [Weimer, Nguyen, Le Goues, Forrest]
  - **ClearView**: detects normal workload invariants and anomalies, deploying binary repairs to restore invariants. [Perkins, Kim, et al.]
  - **Pachika**: summarizes test executions to behavior models, generating fixes based on the differences. [Dallmeier et al.]

- 2012: 30 papers on auto program repair
  - At least 20+ different techniques, 3+ best paper awards, etc.
  - Two main approaches
    - uses random mutation to create *multiple* repair candidates and validate them
    - uses constraint solving to create a *single* repair that is correct-by-construction

- 2013: ICSE has a Program Repair session
Invariant Generation and Template-based Synthesis

Invariant Generation

def intdiv(x, y):
    q = 0
    r = x
    while r ≥ y:
        a = 1
        b = y
        while [??] r ≥ 2b:
            a = 2a
            b = 2b
            r = r - b
            q = q + a
        [??]
    return q

- Discovers invariant properties at certain program locations
- Answers the question “what does this program do?”
Invariant Generation

```
def intdiv(x, y):
    q = 0
    r = x
    while r ≥ y:
        a = 1
        b = y
        while r ≥ 2b:
            a = 2a
            b = 2b
            r = r - b
        q = q + a
    return q
```

- Discovers invariant properties at certain program locations
- Answers the question "what does this program do?"

Template-based Synthesis

```
def intdiv(x, y):
    q = 0
    r = x
    while r [**] y:
        a = 1
        b = [**]
        while r ≥ 2b:
            a = [**]
            b = 2b
            r = r - b
        q = q + a
    return [**]
```

- Creates code under specific templates from partially completed programs
- Can be used for automatic program repair
Software is the problem, and industry is already paying untrusted strangers.

Automated Program Repair is a hot research area with rapid growth in the last few years.

GenProg: does not rely on formal specification, no pre-specification of bug type or repair template.

Challenges and Opportunities:
- Invariant Generation
- Program Synthesis