Automatically Finding Patches Using Genetic Programming

Westley Weimer
Claire Le Goues
ThanVu Nguyen
Stephanie Forrest
Motivation

• Software Quality remains a key problem
  • Over one half of 1 percent of US GDP each year
  • Programs ship with known bugs
• Reduce debugging costs
  • Bug reports accompanied by patches are addressed more rapidly
• Automated Patch Generation
  • Transform a program with a bug
  • Into a program without the bug
  • By modifying relevant parts of the program
The Cunning Plan

- We can automatically and efficiently repair certain classes of bugs in off-the-shelf, unannotated legacy programs.
- Basic idea: Biased search through the space of all programs until you find a variant that repairs the problem. Key insights:
  - Use existing test cases to evaluate variants.
  - Search by perturbing parts of the program likely to contain the error.
The Process

• Input:
  • The program source code
  • Some regression test cases passed by the program
  • A test case failed by the program (= the bug)

• Genetic Programming Work:
  • Create variants of the program
  • Run them on the test cases
  • Repeat, retaining and combining variants

• Output:
  • New program source code that passes all tests
  • or “no solution found in time”
This Talk

- Genetic Programming
- Weighted Paths
- Our Technique
- Example
- Repair Experiments
- Big Finish
What's In A Name?

- Genetic programming is the application of evolutionary or genetic algorithms to program source code.
  - Population of variants
  - Mutation, crossover
  - Fitness function
- Similar in ways to search-based software engineering:
  - Regression tests to guide the search
Two Secret Sauces

• 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.

• Also: Do not try to invent new code!
The Weighted Path

• We define a **weighted path** to be a list of \(<\text{statement, weight}>\) pairs.

• We use this 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, 0.0)** if S is also visited on a passed test
Genetic Programming for Program Repair: Mutation

- Population of Variants:
  - Each variant is an <AST, weighted path> pair

- Mutation:
  - To mutate a variant \( V = <\text{AST}_V, \text{wp}_V> \), choose a statement \( S \) from \( \text{wp}_V \) biased by the weights
  - Delete \( S \), replace \( S \) with \( S_1 \), or insert \( S_2 \) after \( S \)
    - Choose \( S_1 \) and \( S_2 \) from the entire AST
  - Assumes program contains the seeds of its own repair (e.g., has another null check elsewhere).
Genetic Programming for Program Repair: Fitness

- Compile a variant
  - If it fails to compile, Fitness = 0
  - Otherwise, run it on the test cases
  - Fitness = number of test cases passed
  - Weighted: passing the bug test case is worth more

- Selection and Crossover
  - Higher fitness variants are retained and combined into the next generation

- Repeat until a solution is found
Example: GCD

/* 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);
    return;
}

Bug: when a==0 and b>0, it loops forever!
if (a==0)
  while (b != 0)
    { block }
    printf(... a)
    { block }
    { block }
    return
  { block }
  { block }
  printf(... b)
if (isLeapYear)
  if (a > b)
  { block }
  { block }
  if (a > b)
    { block }
    a = a - b
    { block }
    b = b - a
    printf(... a)
    return
Example: Weighted Path (1/3)

Nodes visited on Negative test case (a=0,b=55) : (printf ...b)
Example: Weighted Path (2/3)

Nodes visited on Positive test case (a=1071, b=1029):
- a = a - b
- b = b - a

Nodes visited on Negative test case (a=0, b=55):
- (printf ...b)
Example: Weighted Path (3/3)

Weighted Path:

```
if (a==0)
    printf(... a)
```

```
while (b != 0)
    printf(... b)
    a = a - b
```

```
if (isLeapYear)
    if (a > b)
        { block }
```

```
printf(... a)
```

```
while (b != 0)
    printf(... b)
    b = b - a
```

```
return
```

```
{ block }
```

```
printf(... b)
```
Example: Mutation (1/2)

Mutation Source: Anywhere in AST
Mutation Destination: Weighted Path
Example: Mutation (2/2)

Mutation Source: Anywhere in AST
Mutation Destination: Weighted Path
if (a==0)

while (b != 0)

printf(... a)

if (isLeapYear)

if (a > b)

{ block }

{ block }

printf(... b)

return

{ block }

a = a - b

{ block }

b = b - a

{ block }

return

Example: Final Repair
Minimize The 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.
## Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Path</th>
<th>Time (s)</th>
<th>Success</th>
<th>Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>22</td>
<td>1.3</td>
<td>149</td>
<td>54%</td>
<td>inf loop</td>
</tr>
<tr>
<td>uniq</td>
<td>1146</td>
<td>81.5</td>
<td>32</td>
<td>100%</td>
<td>segfault</td>
</tr>
<tr>
<td>ultrix look</td>
<td>1169</td>
<td>213.0</td>
<td>42</td>
<td>99%</td>
<td>segfault</td>
</tr>
<tr>
<td>svr4 look</td>
<td>1363</td>
<td>32.4</td>
<td>51</td>
<td>100%</td>
<td>inf loop</td>
</tr>
<tr>
<td>units</td>
<td>1504</td>
<td>2159.7</td>
<td>107</td>
<td>7%</td>
<td>segfault</td>
</tr>
<tr>
<td>derooff</td>
<td>2236</td>
<td>251.4</td>
<td>129</td>
<td>97%</td>
<td>segfault</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>5575</td>
<td>768.5</td>
<td>502</td>
<td>36%</td>
<td>buffer overrun</td>
</tr>
<tr>
<td>indent</td>
<td>9906</td>
<td>1435.9</td>
<td>533</td>
<td>7%</td>
<td>inf loop</td>
</tr>
<tr>
<td>flex</td>
<td>18775</td>
<td>3836.6</td>
<td>233</td>
<td>5%</td>
<td>segfault</td>
</tr>
<tr>
<td>atris</td>
<td>21553</td>
<td>34.0</td>
<td>69</td>
<td>82%</td>
<td>buffer overrun</td>
</tr>
<tr>
<td>average</td>
<td>881.4</td>
<td>184.7</td>
<td></td>
<td>58.7%</td>
<td></td>
</tr>
</tbody>
</table>

Average minimization time: 12 seconds.
Total: 10 repaired programs, over 63,000 lines of code.
Repair Quality

• Repairs are typically *not* what a human would have done
  • Example: our technique 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 regression test cases
  • When POST test is omitted from nullhttpd, the generated repair eliminates POST functionality
  • Tests ensure we do not sacrifice functionality
  • Minimization prevents gratuitous deletions
  • Adding more tests helps rather than hurting
Technique Limitations

- May not handle nondeterministic faults
  - Difficult to test for race conditions, etc.
  - Long term: put scheduler constraints into the variant 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 from CVS repositories
- Slower on large test suites: test case selection
Want to hear more?

ICSE 2009
- Formal algorithm, crossover, mutation
- Representation, parsing, stmt details
- Test cases used
- Sensitivity
- Repair quality
- “Does it work?”

GECCO 2009
- Evolutionary questions
  - nonstandard crossover
  - really evolutionary?
  - operator frequency
- Effect of more test cases
- Scaling behavior
- “Why did it work?”
Conclusions

- We can automatically and efficiently repair certain classes of bugs in off-the-shelf legacy programs.
  - Ten programs totaling 63kloc in about 6 minutes each, on average
- We use regression tests to encode desired behavior.
  - Existing tests encode required behavior
- The genetic programming search focuses attention on parts of the program visited during the bug but not visited during passed test cases.
Questions

- I encourage difficult questions.
Bonus Slide: Test Cases

```bash
#!/bin/sh
# Positive Test Case for nullhttpd (POST data)
ulimit -t 5
/usr/bin/wget --tries=1 --post-data 'name=my_name&submit=submit'
   "http://localhost:/$PORT/cgi-bin/hello.pl"
if diff hello.pl ../known-good-hello.pl-result ; then
  # if the current output matches the known-good output
  echo "passed hello.pl test case" >> ../list-of-tests-passed
fi
```

Figure 2: Positive test case for `nullhttpd`. `wget` is a command-line HTTP client; `ulimit` cuts the test off after five seconds. The test assumes that the sandboxed webserver is accepting connections on `PORT` and has its own copy of `htdocs`, including `cgi-bin/hello.pl`. Note the oracle comparison using `diff` against `known-good-hello.pl-result` on line 6.

```bash
#!/bin/sh
# Negative Test Case for nullhttpd
ulimit -t 5
../nullhttpd-exploit -h localhost -p $PORT -t2
/usr/bin/wget --tries=1 "http://localhost:/$PORT/index.html"
if diff index.html ../known-good-index.html.html-result ; then
  # if the current output matches the known-good output
  echo "passed exploit test case" >> ../list-of-tests-passed
fi
```

Figure 3: Negative test case for `nullhttpd`. If the exploit (line 4) disables the webserver then the request for `index.html` (line 5) will fail.