AUTOMATICALLY FINDING PATCHES USING GENETIC PROGRAMMING

AUTHOR: WESTLEY WEIMER

XU SHENGZHE (PID: shengzx)
Motivation

- Software Quality remains a key problem
  - Over one half of 1 percent of US GDP each year
  - The cost of fixing a defect increases (25k – 16k)
  - Even security-critical bugs take 28 days (avg)
  - Despite bug detection and test suites
  - Programs ship with known bugs

- How can we reduce debugging costs?

- Bug reports accompanied by patches are addressed more rapidly
Main Idea

- So we need Automated Patch Generation
- Software Repair via Genetic Programming
  - Transform a program with a bug
  - Into a program without the bug
  - By modifying relevant parts of the program
Genetic Programming

- Start with an input program
- Generate new program through code ‘mutation’ and ’crossover’
- Evaluate new program, and discard it if it is highly unsatisfactory.
- Programs that are somewhat satisfactory for input cases.
Discussion 1

- Do you have used or heard about Genetic Programming?
- In what scenario it can been used?
Repair Process

**Inputs:** { Buggy program, Passing test cases, Failing test case }

A syntax tree representing the program. Code is weighted based on association with failing test case.

*Failed some tests* → *Evaluation*

*Passed all tests* → *Minimize changes with delta debugging*

*New Program* → *Mutation and Crossover*

*Select code to change*

1-minimal patch for original program

**Output:** { program that passes all given test cases }
Buggy Example:

```c
/* requires: a >= 0, b >= 0 */
void 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);
}
```

- `gcd(1071, 1029)`: Pass!
- `gcd(0, 55)`: Failed!
- `a = 0, b > 0`:
  - Loops forever
Example:
Possible Mutation & Minimal Patch

```c
void gcd_3(int a, int b) {
    if (a == 0) {
        printf("%d", b);
        exit(0); // inserted
        a = a - b; // inserted
    }
    while (b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
        printf("%d", a);
        exit(0);
    }
}
```

- **Why mutate here?**
  - it is associated with failure

- **How mutate?**
  - This code was chosen because it was seen in the whole code

- **Minimal Patch**
  - Only retain necessary code
Overview of the strategy

1. Create simplified representation of program
2. Select code to change
3. Genetic Operation
4. Fitness Function: evaluate new program; Go to step 5 if all tests pass, else go to step 2
5. Reduce the changes to a minimal patch
Overview of the strategy

Input: Program $P$ to be repaired.
Input: Set of positive testcases $PosT$.
Input: Set of negative testcases $NegT$.
Output: Repaired program variant.

1. $Path_{PosT} \leftarrow \bigcup_{p \in PosT} \text{statements visited by } P(p)$
2. $Path_{NegT} \leftarrow \bigcup_{n \in NegT} \text{statements visited by } P(n)$
3. $Path \leftarrow \text{set_weights}(Path_{NegT}, Path_{PosT})$
4. $Popul \leftarrow \text{initial_population}(P, \text{pop_size})$
5. repeat
6. $Viable \leftarrow \{\langle P, Path_P, f \rangle \in Popul \mid f > 0\}$
7. $Popul \leftarrow \emptyset$
8. $NewPop \leftarrow \emptyset$
9. for all $\langle p_1, p_2 \rangle \in \text{sample}(Viable, \text{pop_size}/2)$ do
10. $\langle c_1, c_2 \rangle \leftarrow \text{crossover}(p_1, p_2)$
11. $NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
end for
12. for all $\langle V, Path_V, f_V \rangle \in NewPop$ do
13. $Popul \leftarrow Popul \cup \{\text{mutate}(V, Path_V)\}$
end for
14. until $\exists \langle V, Path_V, f_V \rangle \in Popul. f_V = \text{max_fitness}$
15. return $\text{minimize}(V, P, PosT, NegT)$
Step 1: Program Representation

- **Abstract Syntax Tree (AST)**
  - Represents statements and structure of program

- **Weighted Path**
  - Weights indicate association with passing/failing test cases
  - Weight is 1 for statements only reached in the failing case
  - Weight is 0.01 for statements ever reached in a passing case
Example: Abstract Syntax Tree (AST)

```
if (a == 0)
  { block }
  printf(... b)

while (b != 0)
  { 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):
- b = b - a
- a = a - b

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

Weighted Path:

(printf ...b)
Step 2: Select

- Prototype Tool: stochastic universal sampling
- Select the population or variable of each phase of the GP
- Example:
  - Discard individuals with fitness 0
  - Select pop_size/2 new members of the population
Step 3.1: Mutation

- Changes a statement on the weighted path
  - Individual mutation rate: favor changing statements with high weight
  - Global mutation rate: parameter $W$
- Any code inserted/swapped is copied from another part of the program rather than newly created by the algorithm
- In normal GP mutation: single bit flips
- Primitive unit here: deletion, insertion and swap.
Step 3.1: Mutation

- Changes a statement on the weighted path
- Biased to favor changing statements with high weight
- Any new code inserted/swapped is copied from another part of the program

**Input:** Program $P$ to be mutated.
**Input:** Path $Path_P$ of interest.
**Output:** Mutated program variant.

1. for all $(stmt_i, prob_i) \in Path_P$ do
2. if $\text{rand}(0, 1) \leq prob_i \land \text{rand}(0, 1) \leq W_{mut}$ then
3. let $op = \text{choose}([\text{insert, swap, delete}])$
4. if $op = \text{swap}$ then
5. let $stmt_j = \text{choose}(P)$
6. $Path_P[i] \leftarrow (stmt_j, prob_i)$
7. else if $op = \text{insert}$ then
8. let $stmt_j = \text{choose}(P)$
9. $Path_P[i] \leftarrow (\{stmt_i; stmt_j\}, prob_i)$
10. else if $op = \text{delete}$ then
11. $Path_P[i] \leftarrow (\{\}, prob_i)$
12. end if
13. end if
14. end for
15. return $(P, Path_P, \text{fitness}(P))$

**Figure 2.** Our mutation operator. Updates to $Path_P$ also update the AST $P$. 
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
Example: Finial Repair

```
{ block }
if (a==0)
  { block }
  printf(... b)
  return

{ block }
while (b != 0)
  { block }
  printf(... a)
  return

{ block }
if (a > b)
  { block }
  a = a - b
{ block }
  b = b - a
```
Step 3.2: Crossover

- Combines two parts from different individuals
  - One individual is always the original program
  - Highly weighted statements are more likely to be crossed over

- Example:
  - Input: [P1, P2, P3, P4] & [Q1, Q2, Q3, Q4], cutoff 2
  - =>
  - Children: C=[P1, P2, Q3, Q4] & D=[Q1, Q2, P3, P4]
Step 3.2: Crossover

- Two unusual feature
- Crossing Back
- Probability of crossover is based on weight

**Input:** Parent programs $P$ and $Q$.
**Input:** Paths $Path_P$ and $Path_Q$.
**Output:** Two new child program variants $C$ and $D$.

```
1: cutoff ← choose(|Path_P|)
2: $C, Path_C ← copy(P, Path_P)$
3: $D, Path_D ← copy(Q, Path_Q)$
4: for $i = 1$ to $|Path_P|$ do
5: if $i > cutoff$ then
6: let $(stmt_p, prob) = Path_P[i]$
7: let $(stmt_q, prob) = Path_Q[i]$
8: if rand(0, 1) $\leq$ prob then
9: $Path_C[i] ← Path_Q[i]$
10: $Path_D[i] ← Path_P[i]$
11: end if
12: end if
13: end for
14: return $\langle C, Path_C, \text{fitness}(C) \rangle, \langle D, Path_D, \text{fitness}(D) \rangle$
```

**Figure 3.** Our crossover operator. Updates to $Path_C$ and $Path_D$ update the ASTs $C$ and $D$. 
Discussion 2

- Does crossing back matter the algorithm result?

- What do you think is the meaning of weighted crossover.
Step 4: Fitness evaluation

- Individuals are evaluated based on how many test cases they pass.
- If no individual passes all test cases, the best one is chosen for further mutation/crossover.
- If an individual passes all test cases, it becomes the patch candidate and is passed to the minimizer.
Step 5: Patch Minimization

- 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
Step 5: Patch Minimization

- Delta Debugging finds a 1-minimal subset of the diff in $O(n^2)$ time
  - Removing any single line causes a test to fail
- Use a tree-structured diff algorithm (diffX)
  - Avoids problems with balanced curly braces, etc.
Result & Evaluation

- The experiments evaluate:
  - Performance and scalability
  - Runtime
  - Success rate of search
  - Effect of test cases on repair quality

- Tested 10 open source programs
  - One negative test case per program
  - Set pop_size = 40
  - Set max generations to 10
  - 100 random trials per program
Result & Evaluation

- Avg. successful trial takes 184.7 seconds after 36.9 fitness evaluations
  - 54% of time spent executing test cases
  - 30% of time spent compiling individuals
- Over half the trials produced a repair (success)
  - Standard deviation of success rates is very large
- Avg. patch candidate produced after 3.5 crossovers, 1.8 mutations, over 6 generations.
- Final minimized patch is 4 lines on average
### Result & Evaluation

| Program  | LOC  | Positive Tests | |Path| | Time | Initial Repair | Minimized Repair |
|----------|------|----------------|------|-----|------|----------------|------------------|
|          |      |                |      |     |      | fitness        | Success | Size | Time | fitness | Size |
| gcd      | 22   | 5x human       | 1.3  | 149 | 41.0 | 54%            | 21      | 4    | 2    | 4       | 2    |
| uniq     | 1146 | 5x fuzz        | 81.5 | 32  | 9.5  | 100%           | 24      | 2    | 6    | 6       | 4    |
| look-u   | 1169 | 5x fuzz        | 213.0| 42  | 11.1 | 99%            | 24      | 3    | 10   | 10      | 11   |
| look-s   | 1363 | 5x fuzz        | 32.4 | 51  | 8.5  | 100%           | 21      | 4    | 5    | 5       | 3    |
| units    | 1504 | 5x human       | 2159.7| 107 | 55.7 | 7%             | 23      | 2    | 6    | 6       | 4    |
| deroff   | 2236 | 5x fuzz        | 251.4| 129 | 21.6 | 97%            | 61      | 2    | 7    | 7       | 3    |
| nullhttpd| 5575 | 6x human        | 768.5| 502 | 79.1 | 36%            | 71      | 76   | 16   | 16      | 5    |
| indent   | 9906 | 5x fuzz        | 1435.9| 533 | 95.6 | 7%             | 221     | 13   | 13   | 13      | 2    |
| flex     | 18775| 5x fuzz        | 3836.6| 233 | 33.4 | 5%             | 52      | 7    | 6    | 6       | 3    |
| atris    | 21553| 2x human       | 34.0 | 69  | 13.2 | 82%            | 19      | 11   | 7    | 7       | 3    |
| average  |      |                | 881.4| 184.7 | 36.9 | 58.7%          | 53.7    | 12.4 | 8.0  | 4.0     |
Discussion 3

- What are the main constraints on runtime?
  - i.e. huger project not necessarily means more time consuming.
Discussion 4

- How do you think the scalability of this algorithm?
Discussion 4

- How do you think the scalability of this algorithm?
  - Search space
  - Population of each generation
  - Accuracy
  - Time
Discussion 5

- Are there any problems in the process of this paper?
- Can any new ideas be raised today?
- Any Comment?
Thanks for listening