Automatically Finding Patches Using Genetic Programming

Authors: Westley Weimer, ThanhVu Nguyen, Claire Le Goues, Stephanie Forrest
Presented by: David DeBonis, Qi Lu, Shuang Yang

Department of Computer Science

University of New Mexico

February 11, 2013

THE UNIVERSITY of NEW MEXICO

Outline

Error Correction in Source Code

- Based on positive / negative test cases
- Program isolation
- Repair by insert / delete / swap
- Repeat until a variant passes all tests
- Minimize difference

Example of Errant Code

Listing 1: Euclid's greatest common divisor algorithm

```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);
}
```

Variants Creation and Representation

- Restricted to code substitutions from other parts
- Mutation / Crossover constrained to area relevant to error
- Abstract Syntax Tree (AST)
- Weighted program path
- Fitness $= \sum_{i=1}^{N} w_i$ passed test cases

Program Representation

- An *abstract syntax tree(AST)*. It includes all of the statements in the program.
- A *weighted path* through the program, $(statement, weight)$. The weighted path is a list of pairs, each pair contains a statement and a weight based on that statement's occurrences in various testcases.

The algorithm of mutation

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

for all $\langle stmt_i, prob_i \rangle \in Path_P$ **do**
    if $\mathsf{rand}(0,1) \leq prob_i \wedge \mathsf{rand}(0,1) \leq W_{mut}$ **then**
        let $op = \mathsf{choose}(\{\mathsf{insert}, \mathsf{swap}, \mathsf{delete}\})$
        if $op = \mathsf{swap}$ **then**
            let $stmt_j = \mathsf{choose}(P)$
            $Path_P[i] \leftarrow \langle stmt_j, prob_i \rangle$
        else if $op = \mathsf{insert}$ **then**
            let $stmt_j = \mathsf{choose}(P)$
            $Path_P[i] \leftarrow \langle \{stmt_i; stmt_j\}, prob_i \rangle$
        else if $op = \mathsf{delete}$ **then**
            $Path_P[i] \leftarrow \langle \{\}, prob_i \rangle$
        end if
    end if
end for
return $\langle P, Path_P, \mathsf{fitness}(P) \rangle$

The example for Mutation

```
while (b != 0)
  if (a > b)
    a = a - b;
  else
    b = b + a;
return a;
```

The example for Mutation

```
while (b != 0)
  if (a > b)
    a = a - b;
  else
    b = b + a;
return a;
```

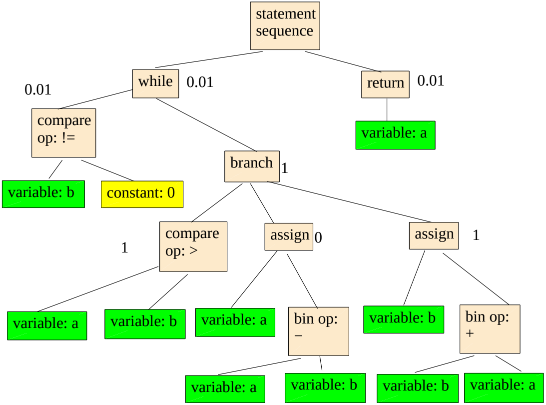
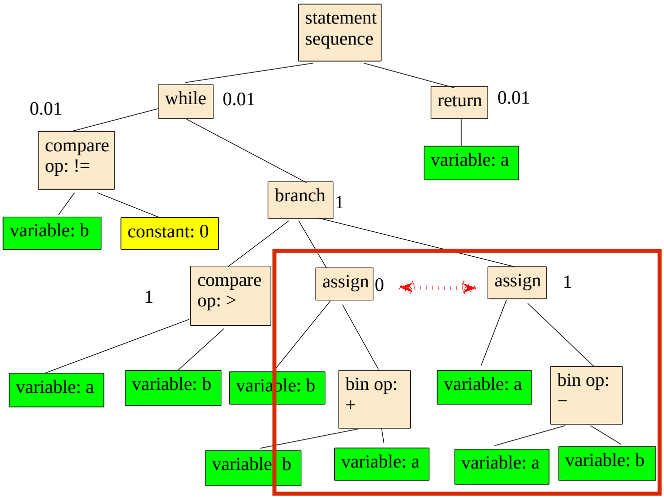


Figure: The AST of the program

Mutation: Swap

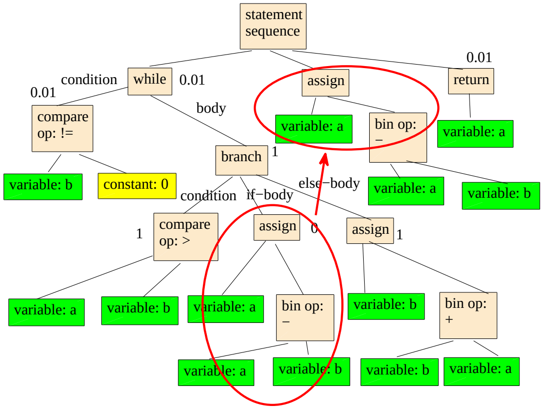


Mutation: Swap

```
while (b != 0)
  if (a > b)
    a = a - b;
  else
    b = b + a;
return a;
```

```
while (b != 0)
  if (a > b)
    b = b + a;
  else
    a = a - b;
return a;
```

Mutation: Insertion

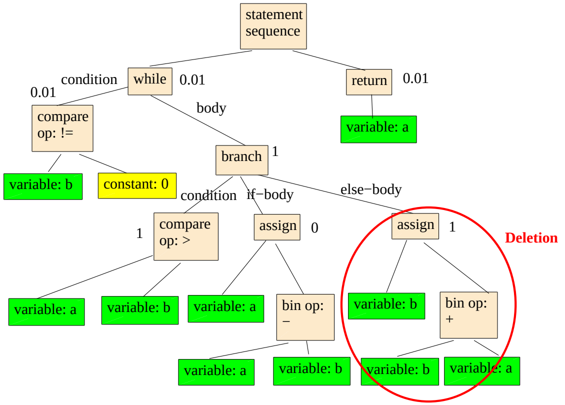


Mutation: Insertion

```
while (b != 0)
  if (a > b)
    a = a - b;
  else
    b = b + a;
return a;
```

```
while (b != 0)
  if (a > b)
    a = a - b;
  else
    b = b + a;
a = a - b;
return a;
```

Mutation: Deletion

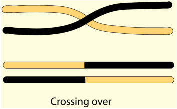


Mutation: Deletion

```
while (b != 0)
  if (a > b)
    a = a - b;
  else
    b = b + a;
return a;
```

```
while (b != 0)
  if (a > b)
    a = a - b;
return a;
```

Crossover



- Choose a cutoff point for each program
- Combine the "first part" of one program with the "second part" of another and vice versa
- Input: $[P_1, P_2, P_3, P_4]$ and $[Q_1, Q_2, Q_3, Q_4]$ with cutoff 2
  Child: $[P_1, P_2, Q_3, Q_4]$ and $[Q_1, Q_2, P_3, P_4]$

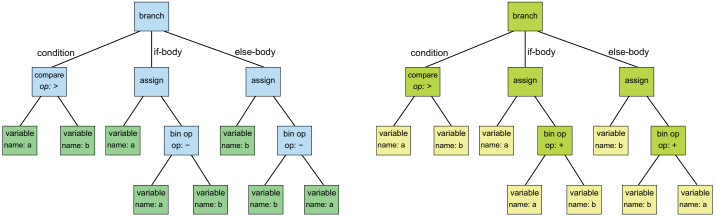
Crossover

```
1 if (a > b)
2   a = a - b;
3 else
4   b = b - a;
```

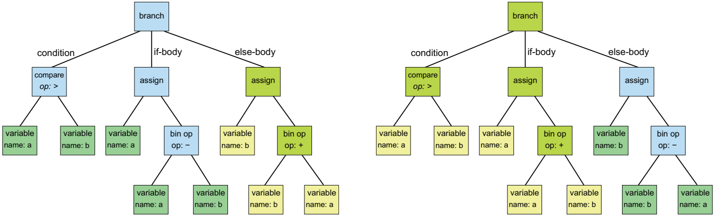
```
1 if (a > b)
2   a = a + b;
3 else
4   b = b + a;
```

- cutoff point at 4

Crossover


branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: −
variable
name: b
bin op
op: −
variable
name: a
variable
name: b
variable
name: b
variable
name: a
branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: +
variable
name: b
bin op
op: +
variable
name: a
variable
name: b
variable
name: b
variable
name: a

Crossover


branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: −
variable
name: b
bin op
op: +
variable
name: a
variable
name: b
variable
name: b
variable
name: a
branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: +
variable
name: b
bin op
op: −
variable
name: a
variable
name: b
variable
name: b
variable
name: a

Crossover

```
if (a > b)
  a = a - b;
else
  b = b + a;
```

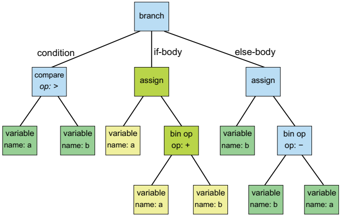
```
if (a > b)
  a = a + b;
else
  b = b - a;
```

Crossover

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

$cutoff \leftarrow \mathsf{choose}(|Path_P|)$
$C, Path_C \leftarrow \mathsf{copy}(P, Path_P)$
$D, Path_D \leftarrow \mathsf{copy}(Q, Path_Q)$
for $i = 1$ to $|Path_P|$ **do**
    if $i > cutoff$ **then**
        let $\langle stmt_p, prob \rangle = Path_P[i]$
        let $\langle stmt_q, prob \rangle = Path_Q[i]$
        if $\mathsf{rand}(0, 1) \leq prob$ **then**
            $Path_C[i] \leftarrow Path_Q[i]$
            $Path_D[i] \leftarrow Path_P[i]$
        end if
    end if
end for
return $\langle C, Path_C, \mathsf{fitness}(C) \rangle, \langle D, Path_D, \mathsf{fitness}(D) \rangle$

Crossover


branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: +
variable
name: b
bin op
op: −
variable
name: a
variable
name: b
variable
name: b
variable
name: a
branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: −
variable
name: b
bin op
op: +
variable
name: a
variable
name: b
variable
name: b
variable
name: a

Conclusion

- Demonstrated GP approach
  - Based on AST representation
  - Mutate / Crossover algorithm
  - Minimal code changes
- Experiments over 10 test cases
  - Sizeable code bases
  - Orthogonal errors present
  - Reasonable solution found (50%)
- Efficacy to code maintenance
  - Experiments using Amazon Cloud
  - Equated to under $8 per bug on average
  - Who wants to manually debug anyway!

Discussion

- Limitations and Assumptions
  - Only as good as your test cases
  - Not necessarily memory / computing time optimized
  - Expects redundant code segments
- How well does it scale?
  - by number of bugs...
  - by number of revisions...
  - by number of LOC...
- Will this technique work on another GP?