Automatically Finding Patches Using Genetic Programming

Team 7

CS454 Team Paper Presentation

Main Paper

Automatically Finding Patches Using Genetic Programming *

Westley Weimer
University of Virginia
weimer@virginia.edu

ThanhVu Nguyen
University of New Mexico
tnguyen@cs.unm.edu

Claire Le Goues
University of Virginia
legoues@virginia.edu

Stephanie Forrest
University of New Mexico
forrest@cs.unm.edu

Abstract

Automatic program repair has been a longstanding goal in software engineering, yet debugging remains a largely manual process. We introduce a fully automated method for locating and repairing bugs in software. The approach works on off-the-shelf legacy applications and does not require formal specifications, program annotations or special coding practices. Once a program fault is discovered, an extended form of genetic programming is used to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Standard test cases are used to exercise the fault and to encode program requirements. After a successful repair has been discovered, it is minimized using structural differencing algorithms and delta debugging. We describe the proposed method and report experimental results demonstrating that it can successfully repair ten different C programs totaling 63,000 lines in under 200 seconds, on average.

To alleviate this burden, we propose an automatic technique for repairing program defects. Our approach does not require difficult formal specifications, program annotations or special coding practices. Instead, it works on off-the-shelf legacy applications and readily-available testcases. We use genetic programming to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Our technique takes as input a program, a set of successful positive testcases that encode required program behavior, and a failing negative testcase that demonstrates a defect.

Genetic programming (GP) is a computational method inspired by biological evolution, which discovers computer programs tailored to a particular task [19]. GP maintains a population of individual programs. Computational analogs of biological mutation and crossover produce program variants. Each variant's suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. GP has solved an impressive range

WEIMER, Westley, et al. Automatically finding patches using genetic programming.
In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009. p. 364-374.

Introduction

What we are going to explain...

Fixing bug is really tough... time-consuming...

Why don't we fix bug with "Genetic Programming"?

What we are going to explain…

```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) => 21
gcd(0, 55) => inf loop

```c
void gcd_2(int a, int b) {
  printf("%d", b);
  exit(0);
}
```

gcd(1071, 1029) => 1029
gcd(0, 55) => 55

```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);
}
```

extraneous

Two key innovations

In this idea, the GP potentially has infinite-size search space.

So, the author suggest this two key as innovation to tackle this problem.

1. Restrict the changes of code based on other parts of the program.

2. Constrain the genetic operations of mutation and crossover to operate only on the region of the program that is relevant to the error.

Approach

1. What is it doing wrong?
2. What is it supposed to do?
3. Where should we change it?
4. How should we change it?
5. When are we finished?

Okay... Good.
But, How to...?

- Genetic Programming
- Initialisation
- Representation
- Mutation
- Fitness
- Optimisation

Procedure

remove all variants that fail every testcase.

take a weighted random sample
in the remaining variants.

include parent and child variants in the population,
and apply the mutation operator to each variant.

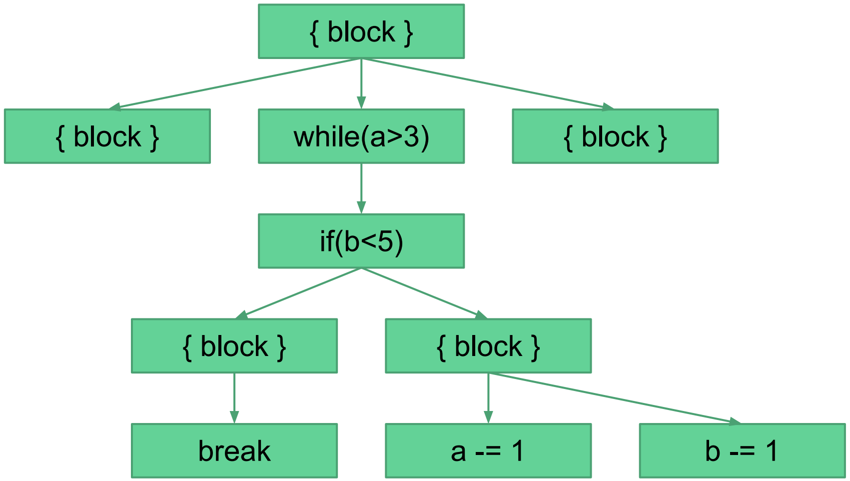
Input: Program P to be repaired.
Input: Set of positive testcases $PosT$.
Input: Set of negative testcases $NegT$.
Output: Repaired program variant.

$Path_{PosT} \leftarrow \bigcup_{p \in PosT}$ statements visited by $P(p)$
$Path_{NegT} \leftarrow \bigcup_{n \in NegT}$ statements visited by $P(n)$
$Path \leftarrow \text{set\_weights}(Path_{NegT}, Path_{PosT})$
$Popul \leftarrow \text{initial\_population}(P, \text{pop\_size})$
repeat
  $Viable \leftarrow \{\langle P, Path_P, f\rangle \in Popul \mid f > 0\}$
  $Popul \leftarrow \emptyset$
  $NewPop \leftarrow \emptyset$
  for all $\langle p_1, p_2\rangle \in \text{sample}(Viable, \text{pop\_size}/2)$ **do**
    $\langle c_1, c_2\rangle \leftarrow \text{crossover}(p_1, p_2)$
    $NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
  end for
  for all $\langle V, Path_V, f_V\rangle \in NewPop$ **do**
    $Popul \leftarrow Popul \cup \{\text{mutate}(V, Path_V)\}$
  end for
until $\exists \langle V, Path_V, f_V\rangle \in Popul \,.\, f_V = \text{max\_fitness}$
return $\text{minimize}(V, P, PosT, NegT)$

Abstract Syntax Tree

- A tree representation of the **abstract syntactic structure** of source code written in a programming language.

```
while(a>3)
{
    if(b<5)
    {
        break;
    }
    else
    {
        a -= 1;
        b -= 1;
    }
}
```



Weighted Path

- We define a weighted path to be a **list** of <statements, weight> pairs.
- We use this weighted path:
  - The statements are those visited during the negative test case.
  - The weight for a statement S is
    - High (= 1.0) if S is <u>not</u> visited on a positive test case
    - Low (= 0.01 or 0) if S is <u>also</u> visited on a positive test case

Mutation

- Each **variant** is an <AST, weighted path> pair
- To mutate a variant $V = <AST_V, wp_V>$, randomly choose a statement S from wp_V biased by the weights
- Delete S, swap S with S1, or insert S2 after S
  - Choose S1 and S2 from the entire AST
- Assumes that program contains the seeds of its own repair

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

Crossover

- **Cutoff point:** *probabilistically* swap statements after the cutoff point.
    - Choosing to swap is based on i-th statement's probability (line 6-11)

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

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

Fitness

- Compile a variant
  - If it fails to compile, Fitness = 0
  - Otherwise, run it on the test cases
  - Fitness = Sum(W_k x C_k) for all k (smaller than # of testcase)
- Selection
  - Higher fitness variants are retained into the next generation
- Repeat until a solution is found

Repair Minimization

- We want to *minimize* the repair patch!
- Random mutations may add unneeded stmts
  - (e.g., dead code, redundant computation)
- In essence: try removing each line in the diff and check if the result still passes all tests

Application:

Zune Bug Repair

Can you find bug...?

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

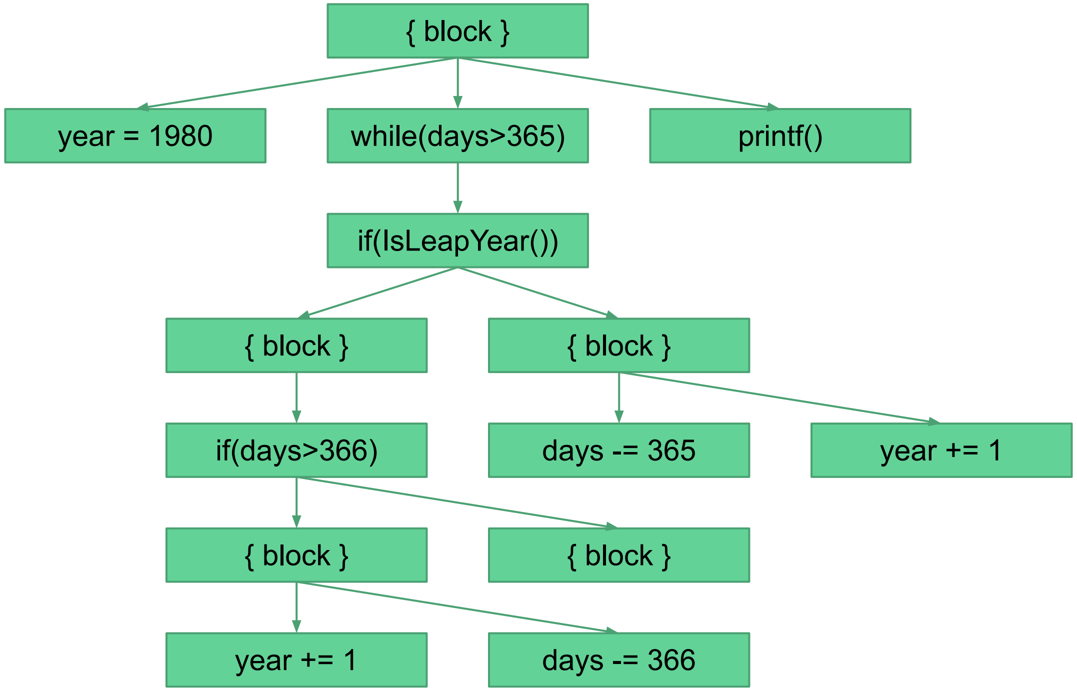
Let's solve it with the algorithm!

Abstract Syntax Tree

```
year = ORIGINYEAR; /* = 1979 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

printf("%d\n", year);
```

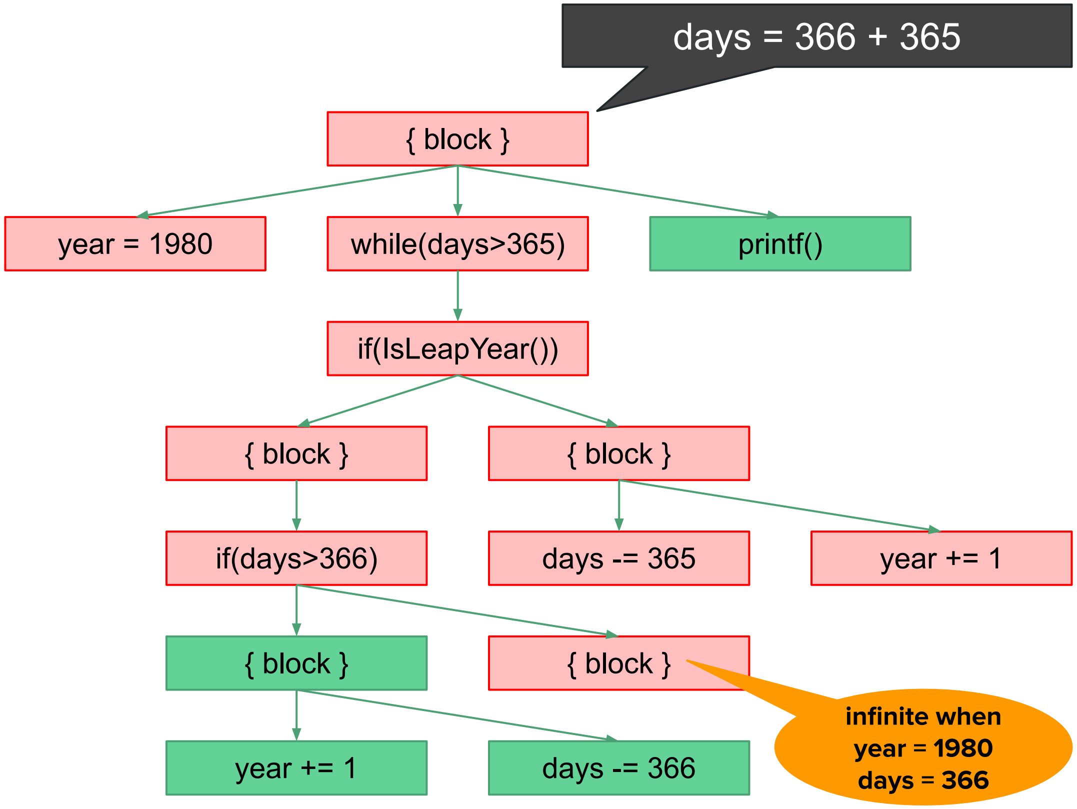


Weighted Path

```
year = ORIGINYEAR; /* = 1979 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

printf("%d\n", year);
```

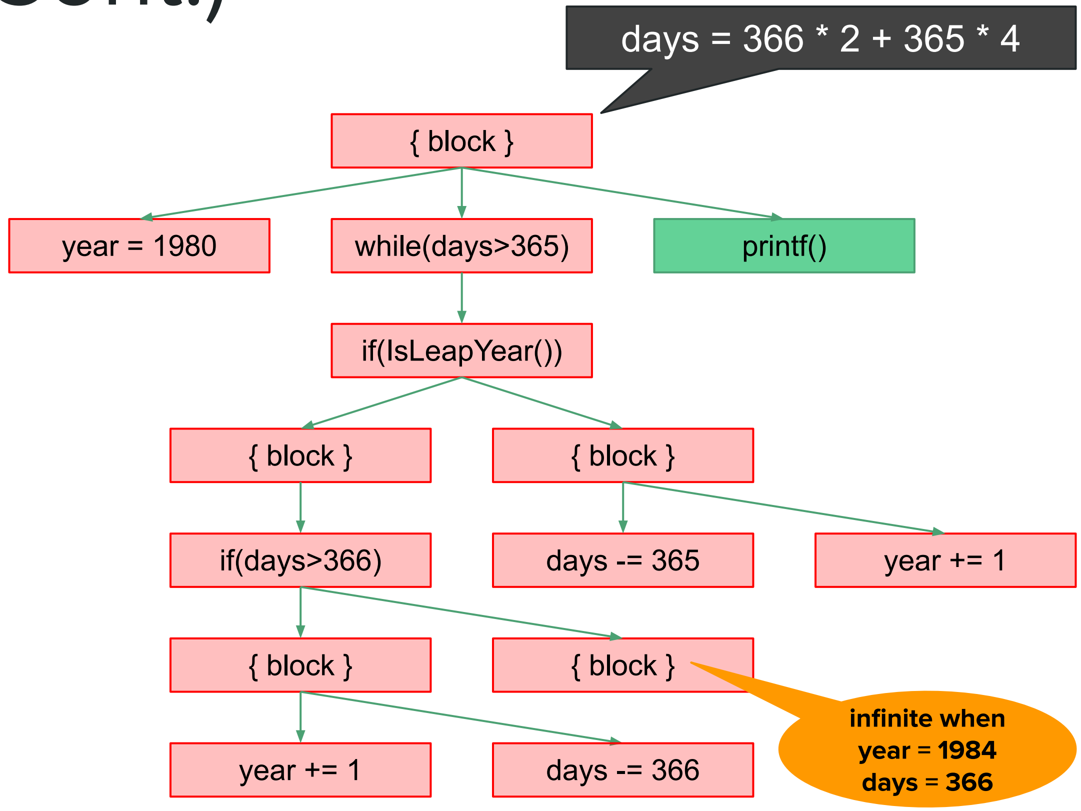


Weighted Path (Cont.)

```
year = ORIGINYEAR; /* = 1979 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

printf("%d\n", year);
```

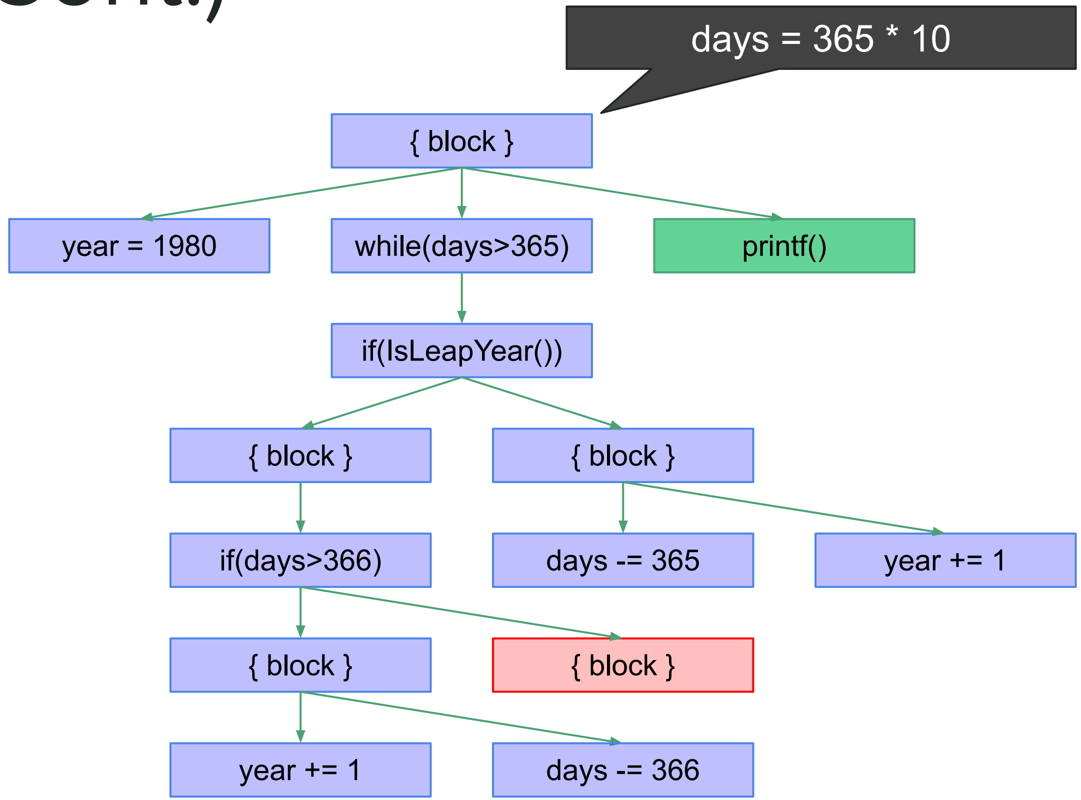


Weighted Path (Cont.)

```
year = ORIGINYEAR; /* = 1979 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

printf("%d\n", year);
```

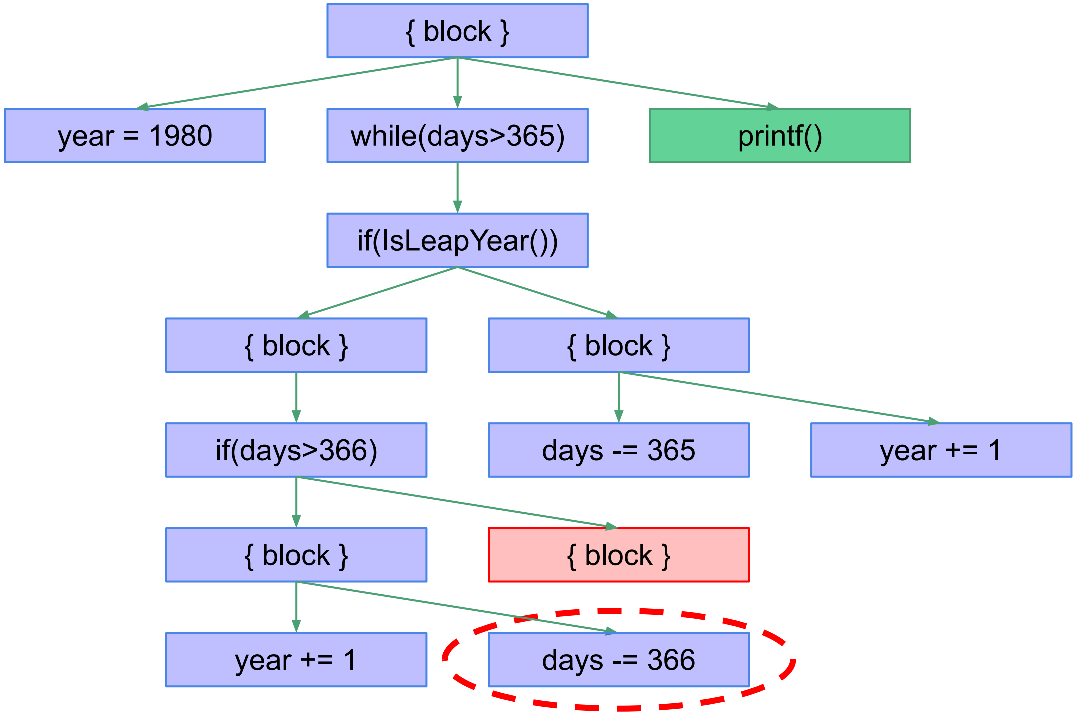


Mutation

```c
year = ORIGINYEAR; /* = 1979 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

printf("%d\n", year);
```

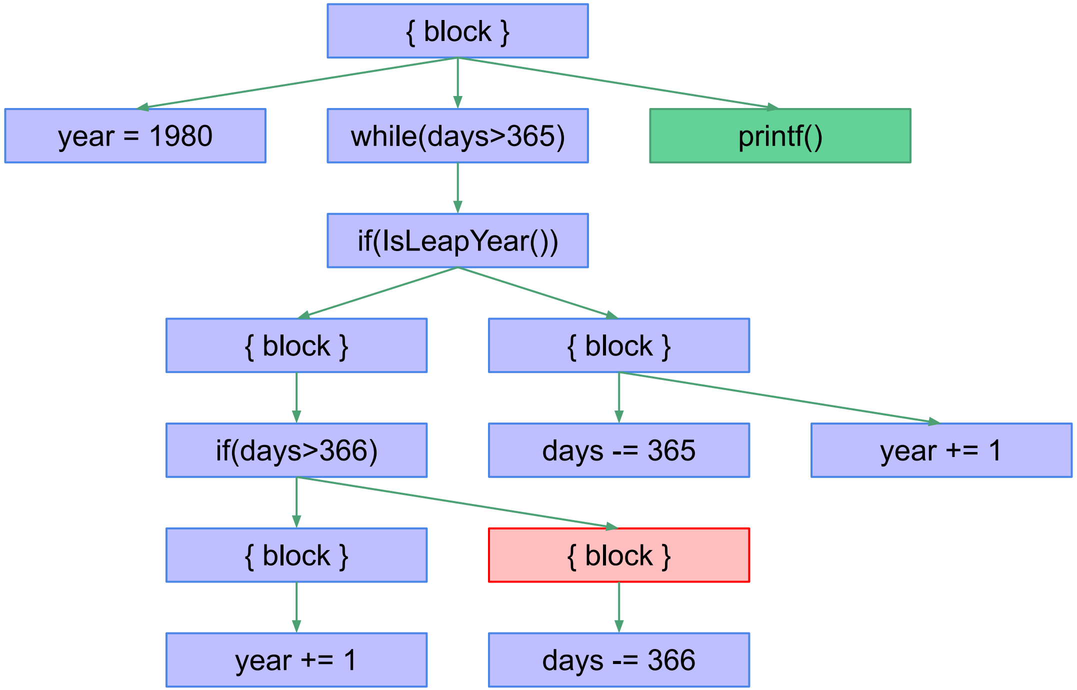


Mutation (Cont.)

```
year = ORIGINYEAR; /* = 1979 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            year += 1;
        }
        else
        {
            days -= 366;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

printf("%d\n", year);
```

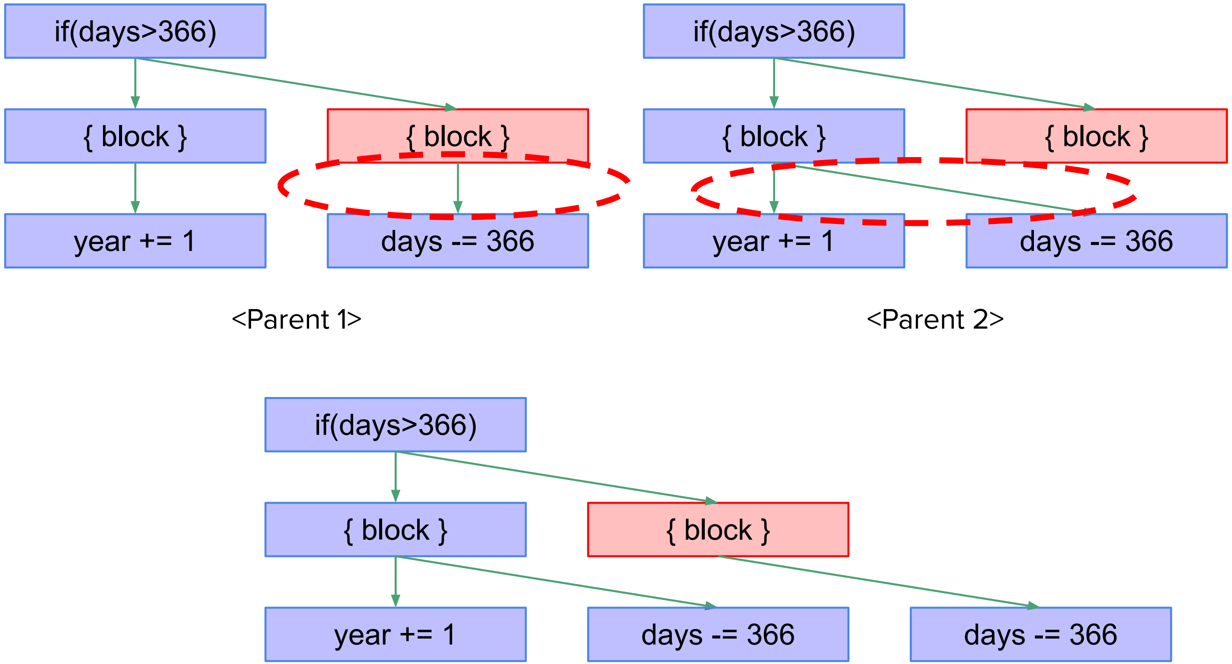


Crossover

```c
year = ORIGINYEAR; /* = 1979 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
        else
        {
            days -= 366;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

printf("%d\n", year);
```



Experiments:

open source benchmarks

Experiment Goals

- Evaluate [performance] and [scalability]
- Measure run-time cost by [# of fitness evaluations] and [elapsed time]
- Evaluate [success rate]
- Understand how testcases affect repair quality

Open Source Benchmarks

Programs with known faults

Program	Version	LOC	Statements	Program Description	Fault
`gcd`	example	22	10	example from Section 2	infinite loop
`uniq`	ultrix 4.3	1146	81	duplicate text processing	segfault
`look`	ultrix 4.3	1169	90	dictionary lookup	segfault
`look`	svr4.0 1.1	1363	100	dictionary lookup	infinite loop
`units`	svr4.0 1.1	1504	240	metric conversion	segfault
`deroff`	ultrix 4.3	2236	1604	document processing	segfault
`nullhttpd`	0.5.0	5575	1040	webserver	remote heap buffer exploit
`indent`	1.9.1	9906	2022	source code processing	infinite loop
`flex`	2.5.4a	18775	3635	lexical analyzer generator	segfault
`atris`	1.0.6	21553	6470	graphical tetris game	local stack buffer exploit
total		63249	15292		

Experiment Setup

- Testcases
  - Single negative testcase
  - Small number of positive testcases (2-6)
- Parameters
- Optimisations
  - Reuse of fitness evaluation

Parameters	Value	
Population	40	
Max generation	10	
W_{PosT}	1	
W_{NegT}	10	
W_{Path}	0.01	0.00
W_{mut}	0.06	0.03

Experiment Result

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

Repair Quality and Testcases

- “nullhttpd” experiment without POST testcase
  - Eliminate POST functionality
  - Aggressively prunes functionality to repair the fault unless that functionality is guarded by testcases
- Tradeoff between
  - Rapid repairs that address the fault
  - Using more testcases to obtain a more human repair

Conclusion

Limitations

- Assumption 1: Defect is reproducible

  ➔ What if program behavior is random?

- Assumption 2: PosT (positive testcases) can encode program requirements

  ➔ What if PosT are not enough / too many?

- Assumption 3: $\text{Path}_{\text{\_PosT}}$ is different from $\text{Path}_{\text{\_NegT}}$

  ➔ What if they overlap completely?

- Assumption 4: Repair can be constructed from statements already in the program

  ➔ … Really?

Conclusion

- First attempt on automatically & efficiently repairing certain classes of bugs in off-the-shelf legacy programs.
- Inspired future researches (remember the first group's presentation?).
- Extended to "**GenProg**: A Generic Method for Automatic Software Repair" in 2011.

Is there any question so far?

Thank you!