HUMIES
(HUMAN-COMPETITIVE RESULTS)
Awards

Conditions

- The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
- The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.
- The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.
- The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.
- The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.
- The result solves a problem of indisputable difficulty in its field.
- The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

Problems

- Fixing software bugs (2009)

Software is THE problem

- Software faults and debugging are expensive:
  - US corporate development organizations spend $5.2 - $22 million annually fixing software defects (IDC Software Quality Survey, 2008)
  - Cost of repairing bugs increases throughout the development process. A $25 fix while the program is under development increases to $16,000 after the software has gone live (IBM Rational group, 2008)
- Security violations are expensive:
  - Average total per-incident costs in 2008 were $6.65 million, compared to an average per-incident cost of $6.3 million in 2007.
  - Monetary loss by 639 companies in 2005 totaled $130 million (FBI 2005)

Software is THE problem

- Bugs are plentiful:
  - Mozilla project received 51154 bug reports in 2002-2006.
  - In 2005, a Mozilla developer reported that “almost 300 bugs appear every day that need triaging.”
- Fixing bugs is time-consuming:
  - Industrial software repair:
    - 1/2 of all fixed bugs in Mozilla from 2002-2006 took more than 29 days for developers to fix;
    - Median repair time for ArgoUML project in 2002-2003 was 190 days;
    - Median repair time per bug for PostgreSQL was 200 days.

Why the problem is difficult for GP

- Search space is huge !
  - Linux 0.01 had 10.000 lines of code.
  - Assume that on each line we have 1 instruction.
  - Assume that we have 256 instructions available.
  - Size of the search space is $256^{10000} \approx 10^{24000}$
  - A computer explores 10^6 OSs per second
  - We have a network with 10^{10} computers
  - One year $\cong 10^8$ seconds
  - We can explore 10^{24} OSs / year
  - We need 10^{23976} years (compare this with the age of universe which is about 10^{10} years).
  - Linux 2.6.31 has 12.000.000 lines of code

Automatic fixing software bugs

- Assume:
  - Access to *C* source code
  - Negative test case that executes the buggy code
  - Positive test cases to encode required program functionality
- Evolve repair that avoids negative test case and passes positive test case
- Minimize repair using program analysis methods

Performance

- Fixing bugs on a source code with 21553 lines of code.

Questions to be answered

- What is it doing wrong? (negative cases)
- What is it supposed to do? (positive cases)
- Where should we change it?
- How should we change it?
- When are we finished?

GP Implementation

- Each GP contains
  - a variant of the program (AST)
  - a weighted program path.
- Crossover and mutation
  - Modify both the program and the weighted path
- Fitness
  - We run the program on all cases.
- Repair the code (post optimization)

Greatest common divisor

```
1 /* requires: a >= 0, b >= 0 */
2 void gcd(int a, int b) {
3   if (a == 0) {
4     printf("%d", b);
5   }
6   while (b != 0) {
7       if (a > b) {
8          a = a - b;
9       } else {
10          b = b - a;
11      }
12  }
13  printf("%d", a);
14  exit(0);
15 }
```

Bugs in GCD

- a = 0, b > 0 -> prints ok, but loops forever
- Does not handle negative values

Negative and positive cases

- Negative gcd (0, 55)
  - Solved by
    - 1 **void gcd_2(int** a, **int** b) {
    - 2 printf("%d", b);
    - 3 exit(0);
    - 4 }
  - Fails on gcd(1071, 1029)

Improvement

- bias the modifications towards the regions of code that are most likely to change behavior on the negative testcase
  - without damaging performance on the positive testcases
- Record all of the lines visited when processing the testcases.
- The positive testcase *gcd* (1071,1029) visits lines 2–3 and 6–15.
- The negative testcase *gcd* (0,55) visits lines 2–5, 6–7, and 9–12.
- When selecting portions of the program to modify, we favor locations that were visited during the negative testcase and were not also visited during the positive ones.

Further improvements

- We could add arbitrary code, delete existing code, or change existing code into new arbitrary code.
  - We make the assumption that most defects can be repaired by adopting existing code from another location in the program.
  - In practice, a program that makes a mistake in one location often handles the situation correctly in another.
  - Example: a program missing a null check or an array bounds check is likely to have a similar working check somewhere else that can be used as a template.
- When mutating a program we may insert, delete or modify statements, but we insert only code that is similar in structure to existing code.
- Thus, we will not insert an arbitrary **if** conditional, but we might insert **if**(*a*==0) or **if**(*a*>*b*) because they already appear in the program.
- Similarly, we might insert printf("%d",a), a=a-b, b=b-a, printf("%d",b), or exit(0), but not arbitrary statements.

GCD fixed

- Given the bias towards modifying lines 4–5 and our preference for insertions similar to existing code, it is reasonable to consider inserting exit(0) and a=a-b between lines 4 and 5, which yields: variant:

```
1 void gcd_3(int a, int b) {
2   if (a == 0) {
3     printf("%d", b);
4     exit(0); // inserted
5     a = a - b; // inserted
6   }
7   while (b != 0) {
8      if (a > b) {
9          a = a - b;
10     } else {
11       b = b - a;
12     }
13  }
14  printf("%d", a);
15  exit(0);
16 }
```

Zune bug

- Infinite loop when input is last day of a leap year.
- Microsoft sold about 1.2 million units of Zune 30, generating thousands of complaints.
- Repair is not trivial.
- Microsoft's recommendation was to let Zune drain its battery and then reset.
- GP discovered the repair in 42 seconds

```
1 void zunebug_repair(int days) {
2   int year = 1980;
3   while (days > 365) {
4     if (isLeapYear(year)){
5       if (days > 366) {
6         // days -= 366; // repair deletes
7         year += 1;
8       }
9       else {
10      }
11      days -= 366; // repair inserts
12    } else {
13      days -= 365;
14      year += 1;
15    }
16 }
17
```

Construct Abstract Syntax Tree (AST)

- An abstract syntax tree (AST) captures the essential structure of the input in a tree form, while omitting unnecessary syntactic details.

- ASTs can be distinguished from concrete syntax trees by their omission of tree nodes to represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments.

Particular implementation of AST

- Program is run on negative cases (that generate the bug).
- Only those nodes are taken into account.
  - Atris program:
    - 8068 nodes
    - Only 34 executed for negative cases.

CIL (C Intermediate Language)

- CIL toolkit used for parsing *C* programs.
- CIL (*C Intermediate Language*) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.
- CIL is both lower-level than abstract-syntax trees, by clarifying ambiguous constructs and removing redundant ones, and also higher-level than typical intermediate languages designed for compilation, by maintaining types and a close relationship with the source program.

Weighted path

- The weighted path is a set of <*statement*, *weight*> pairs that guide the GP search.
- Assume that a statement visited at least once during a negative test case is a reasonable candidate for repair.
- Do not assume that a statement visited frequently (e.g., because it is in a loop) is more likely to be a good repair site.
- Remove all duplicates from each list of statements.

Weighted path

- If there were no positive testcases -> the initial weight on every statement would be 1.0
  - each statement visited along a negative testcase would be a reasonable repair candidate

Mutation

- Based on weight
  - Insert
    - An existing instruction.
  - Swap
    - With any other instruction in the program.
  - Delete

Crossover

- One cutting point
- Called *crossing-back*

Fitness

- Run the program on test cases
- Run in a sandbox:
  - otherwise you might delete all files
- Fitness 0 to programs that do not compile

Repair Minimization

- Find all modified lines
- Try to remove lines one by one to see if the program is still OK.

Parameters

- PopSize = 40
- NumberOfGenerations = 10
- Initial population = mutation (original program)
- Generation:
  - Best 20 are taken and crossed back with the original program.
  - Offspring are mutated.

Other examples

Program	Version	LOC	Time to Repair	Program Description	Fault
gcd	example	22	153s	handcrafted example	infinite loop
zune		28	42s	media player	infinite loop
uniq	ultrix 4.3	1146	34s	duplicate text processing	segfault
look	ultrix 4.3	1169	45s	dictionary lookup	segfault
look	svr4.0 1.1	1363	55s	dictionary lookup	infinite loop
units	svr4.0 1.1	1504	109s	metric conversion	segfault
deroff	ultrix 4.3	2236	131s	document processing	segfault
indent	1.9.1	9906	546s	source code processing	infinite loop
flex	2.5.4a	18775	230s	lexical analyzer generator	segfault
atris	1.0.6	21553	80s	graphical tetris game	loc. stack buffer exploit
nullhttpd	0.5.0	5575	578s	webserver	rem. heap buffer exploit
openldap io.c	2.3.41	6519	665s	directory protocol	non-overflow DOS
lighthttpd fastcgi.c	1.4.17	13984	49s	webserver	rem. heap buf overflow
php string.c	5.2.1	26044	6s	scripting language	int overflow
wu-ftp	2.6.0	35109	2256s	FTP server	format string
total		144933			

Time to discover repair

- Repaired 15 programs totaling nearly 150,000 lines of code
- Average time to repair: 3 minutes (for first 11 programs shown)
- Time includes:
  - GP algorithm (selection, mutation, calculating fitness, etc.)
  - Running test cases
  - Pretty printing and memoizing ASTs
  - gcc (compiling ASTs into executable code)

Repair results

Program	Version	Stmt Nodes / Lines of Code	Pos/Neg Test cases	Final Repair
`zune`	example	14 / 28	5/2	4
`gcd`	example	10 / 22	5/1	2
`uniq`	ultrix 4.3	81 /1146	5/1	4
`look-u`	ultrix 4.3	90 /1169	5/1	11
`look-s`	svr4.0 1.1	100 /1363	5/1	3
`units`	svr4.0 1.1	240 /1504	5/1	4
`deroff`	ultrix 4.3	1604 /2236	5/1	3
`nullhttpd`	0.5.0	1040 /5575	6/1	5
`indent`	1.9.1	2022 /9906	5/1	2
`flex`	2.5.4a	3635 /18775	5/1	3
`atris`	1.0.6	6470 /21553	2/1	3