Automated Program Repair

Motivation

- **Software maintenance is expensive**
  - Up to 90% of the cost of software
    - [Seacord]
  - Up to $70 Billion per year in US
    - [Jorgensen, Sutherland]
  - Bug repair is the majority of maintenance
    - [Erlikh, Ramamoothy, Williamson]
- **Bugs are ubiquitous**
  - Outstanding bugs exceed the resources available to address them
    - [Anvik, Liblit]

Fixing Bugs Early Would Help

- The cost of a defect is $25 during the coding phase, $100 during the build phase, $450 during the QA/testing phase, and $16,000 once released as a product.
  - Leigh Williamson, IBM Distinguished Engineer
- On average, a software vendor loses 0.63% of its market value on the day of any vulnerability announcement.
  - [Telang and Wattal]

Fixing Many Bugs Would Help

- "Everyday, almost 300 bugs appear ... far too much for only the Mozilla programmers to handle."
  - Mozilla Developer [Anvik]

Central Claim

- We can **automatically repair** certain classes of bugs in off-the-shelf, unannotated legacy software.
  - We can automatically repair many types of high-impact security bugs.
  - We can automatically repair many "general software engineering" bugs.

Insights

- Given a buggy program, search the space of **nearby programs** until a valid repair is found.
- Use **test cases** to encode the program specification and the defect
- Find nearby programs by mimicking human edits and leveraging existing **human insight**
- **Reduce the search space** by restricting attention to areas likely to contain the bug
- **Genetic programming** guides the search, tolerating noise and admitting parallelism

Correctness

- **Testing** gives confidence that a program implementation adheres to its specification (as refined from its requirements).
- Loosely, in the **Oracle-Comparator** model, a test case includes:
  - A Test Input
  - An Oracle Answer (expected answer)
  - A Comparator (is an answer close enough?)
- The **current bug** is demonstrated by a test case that currently fails

Fault Localization

- Even in large programs, not every component is equally likely to contribute to a given fault
- Given a program, a bug, and a test suite, **fault localization** produces a mapping from program components to weights
  - "High weight" means "likely related to the bug"
- Many techniques exist
  - [Renieris and Reiss, Jones and Harrold]
- Loosely: print statement debugging, note all statements only visited on bug test case

Fix Localization

- There are a large number of ways to **change a given statement**
  - e.g., deleting it, adding more code to it, replacing it with something completely different, etc.
- When inserting, we **leverage human insight** by using code from elsewhere in the program
  - Program is probably correct elsewhere [Engler]
- Simple Operations: Insert, Replace, Delete
- **Fix localization** is the set of things to insert

Genetic Algorithm Overview

- **Genetic Programming** is a search heuristic
  - Based on a computational analog of biology
  - **Represent** solutions to the problem (genotypes)
  - Uses a **population** of variant solutions
  - Applies simple **mutations**
  - Evaluate the **fitness** of a variant (phenotype)
  - High-fitness variants survive and mate (**crossover**)
  - Repeat until a solution is found

Patch Representation

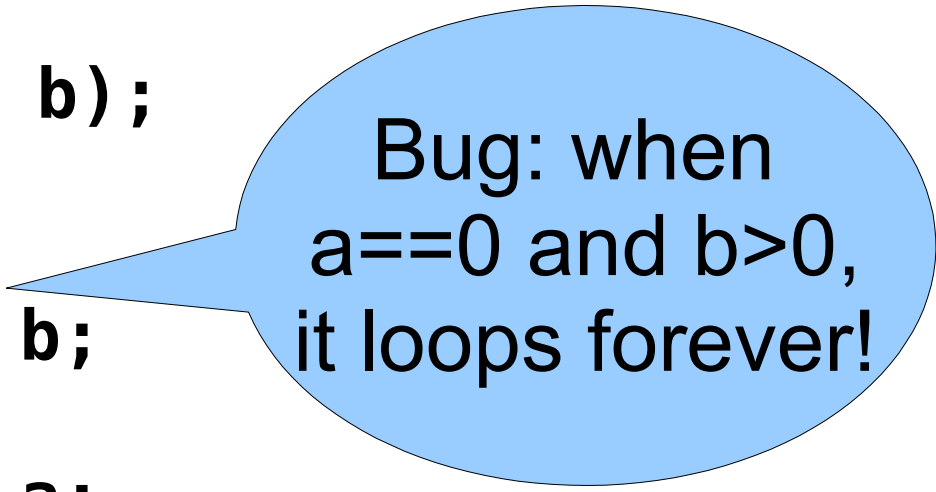
- A **patch** is represented as a **list of edits**
- Edits are **statement-level** operations
  - Example: < Delete(5), Insert(33,2) >
- **Mutation**: add to edit list
  - Choose a statement X from Fault Localization
  - Choose: Delete X *or* Insert(X,FixLocalization(X)) *or* Replace(X,FixLocalization(X))
- **Crossover**: random sublist of parent lists
- **Fitness**: print out patched program, run tests

Overall Repair Algorithm

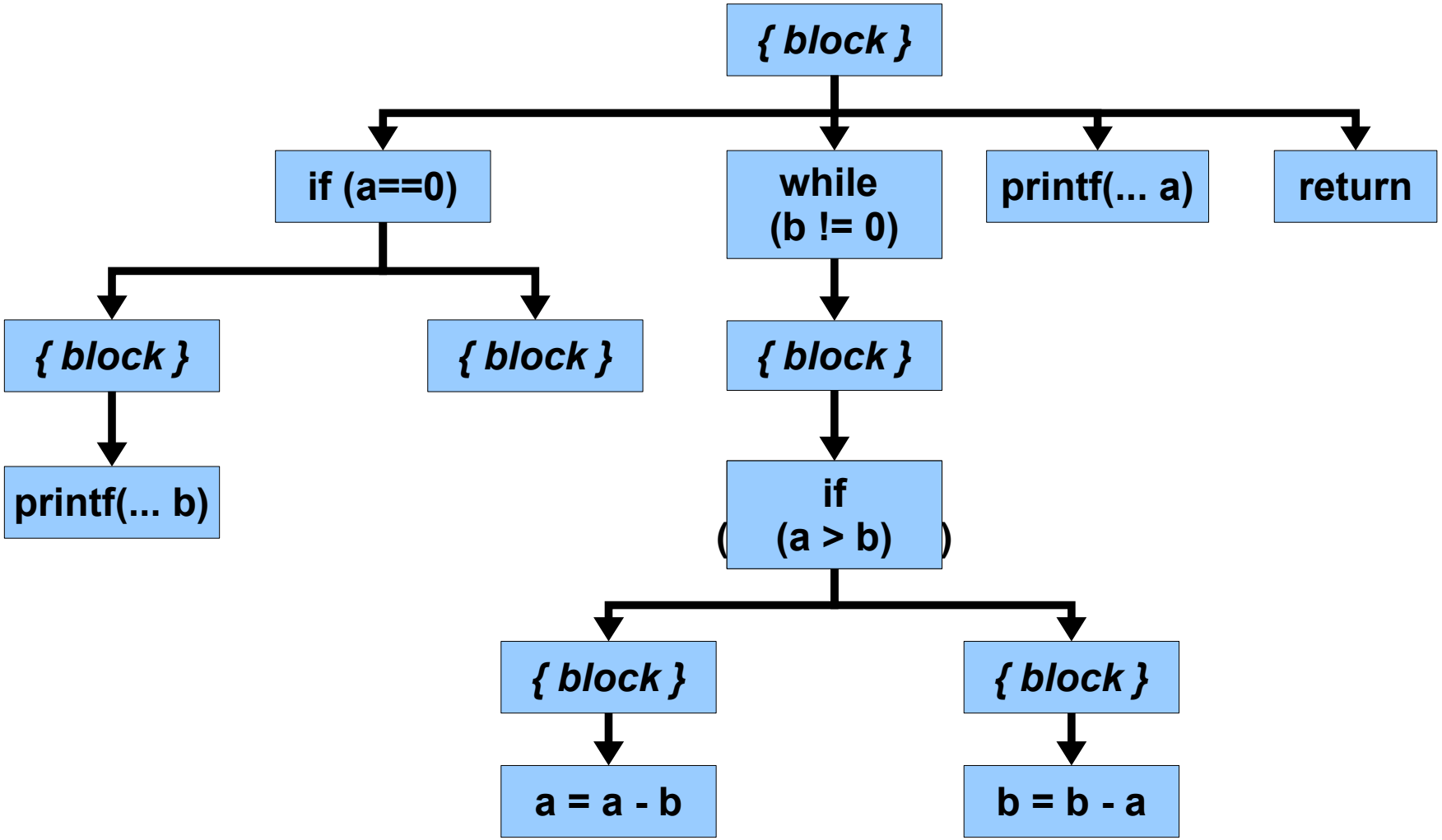
- Input: Program P, Test Suite T
- Loc := Compute Fault and Fix Localizations
- Population := *n* random mutants of P
- Repeat Until Solution Found:
  - HighFit := Select(Population, *n*, T)
  - Offspring := Crossover(HighFit)
  - Population := Mutate(HighFit + Offspring, Loc)

Example: GCD

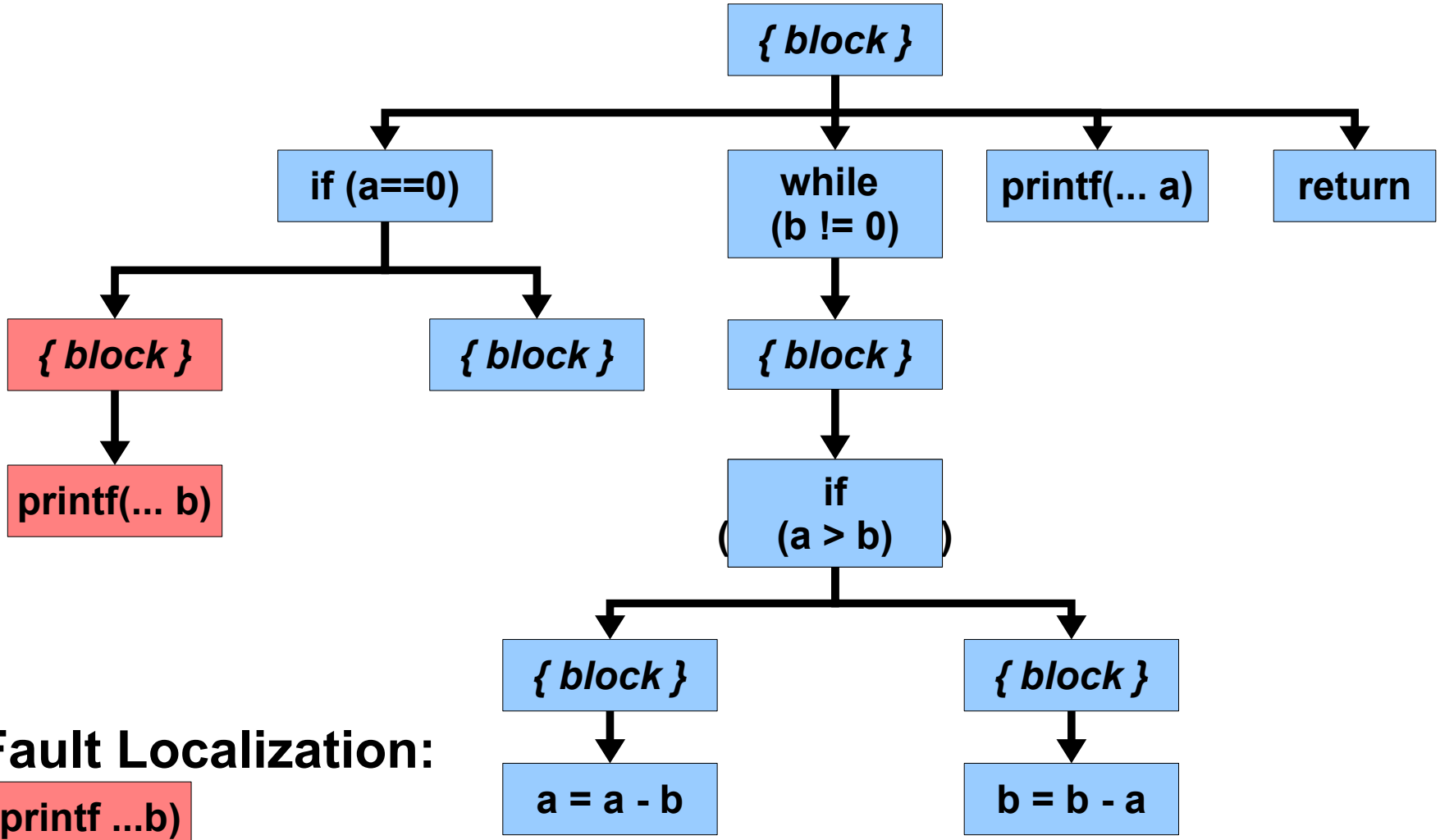
```
void print_gcd(int a, int b) {
     if (a == 0)
          printf("%d", b);
     while (b != 0) {
          if (a > b)
               a = a – b;
          else
               b = b – a;
     }
   printf("%d", a);
     return;
}
```



Example: Representation



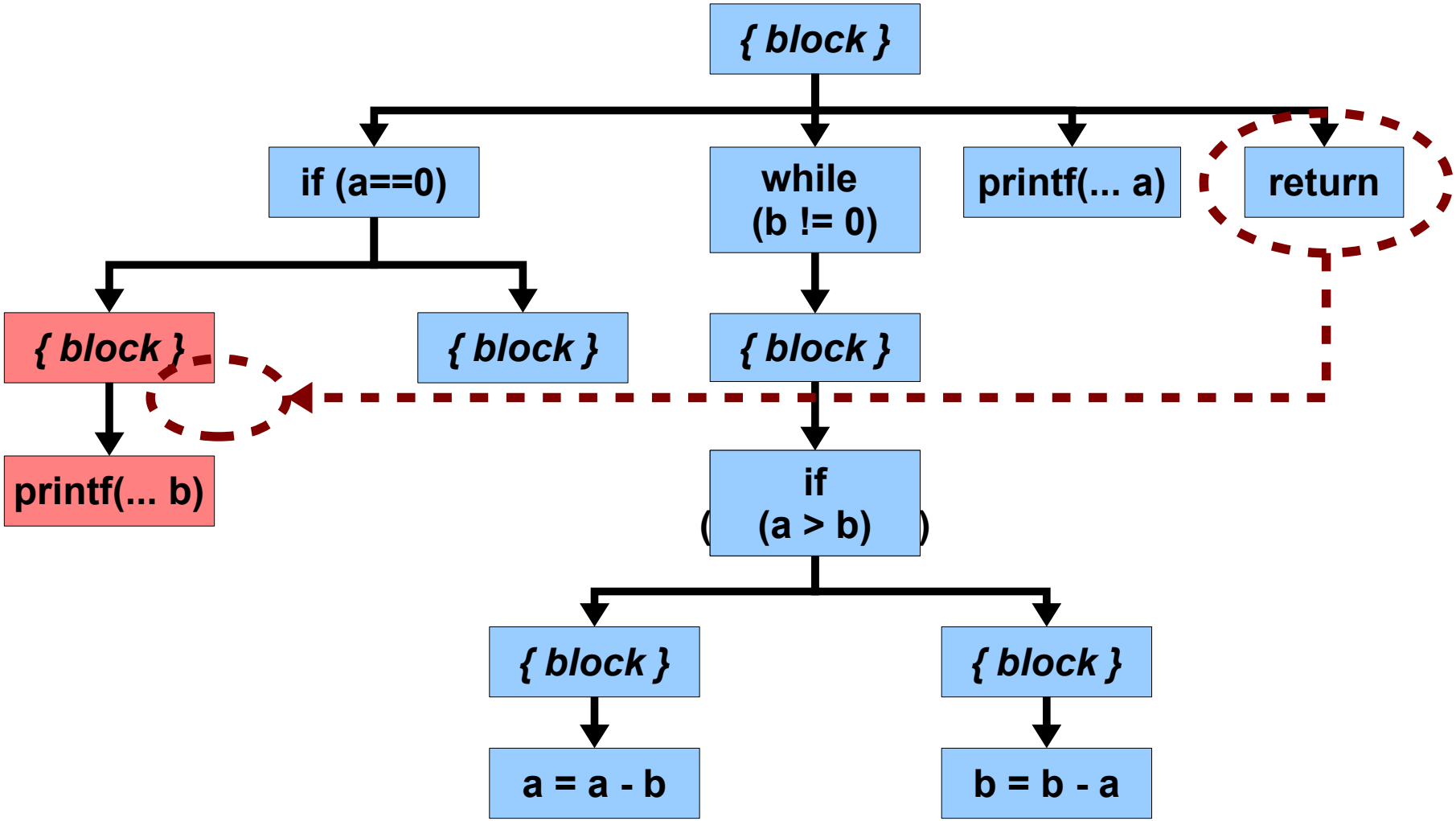
Example: Fault Location



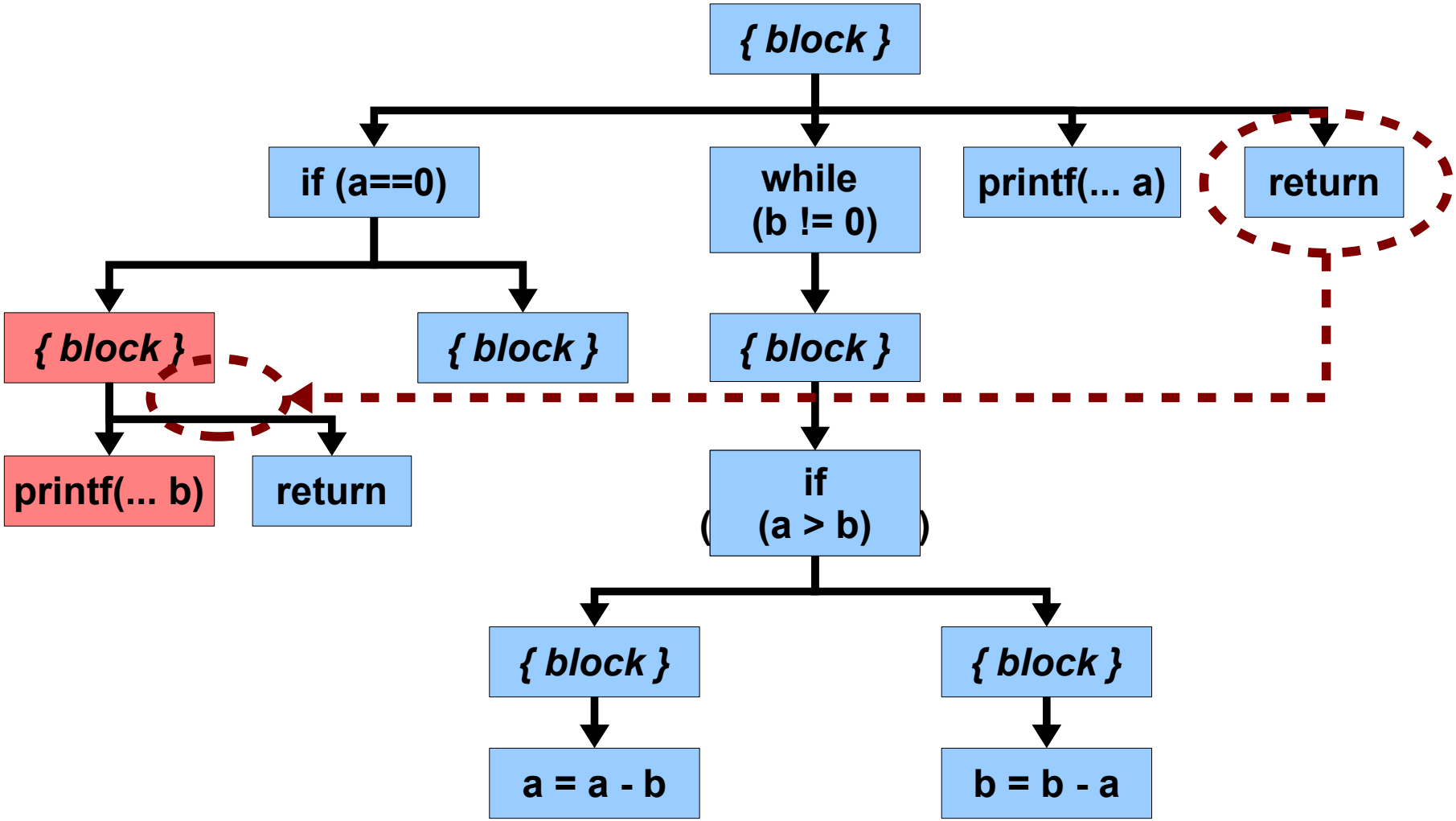
Fault Localization:

(printf ...b)

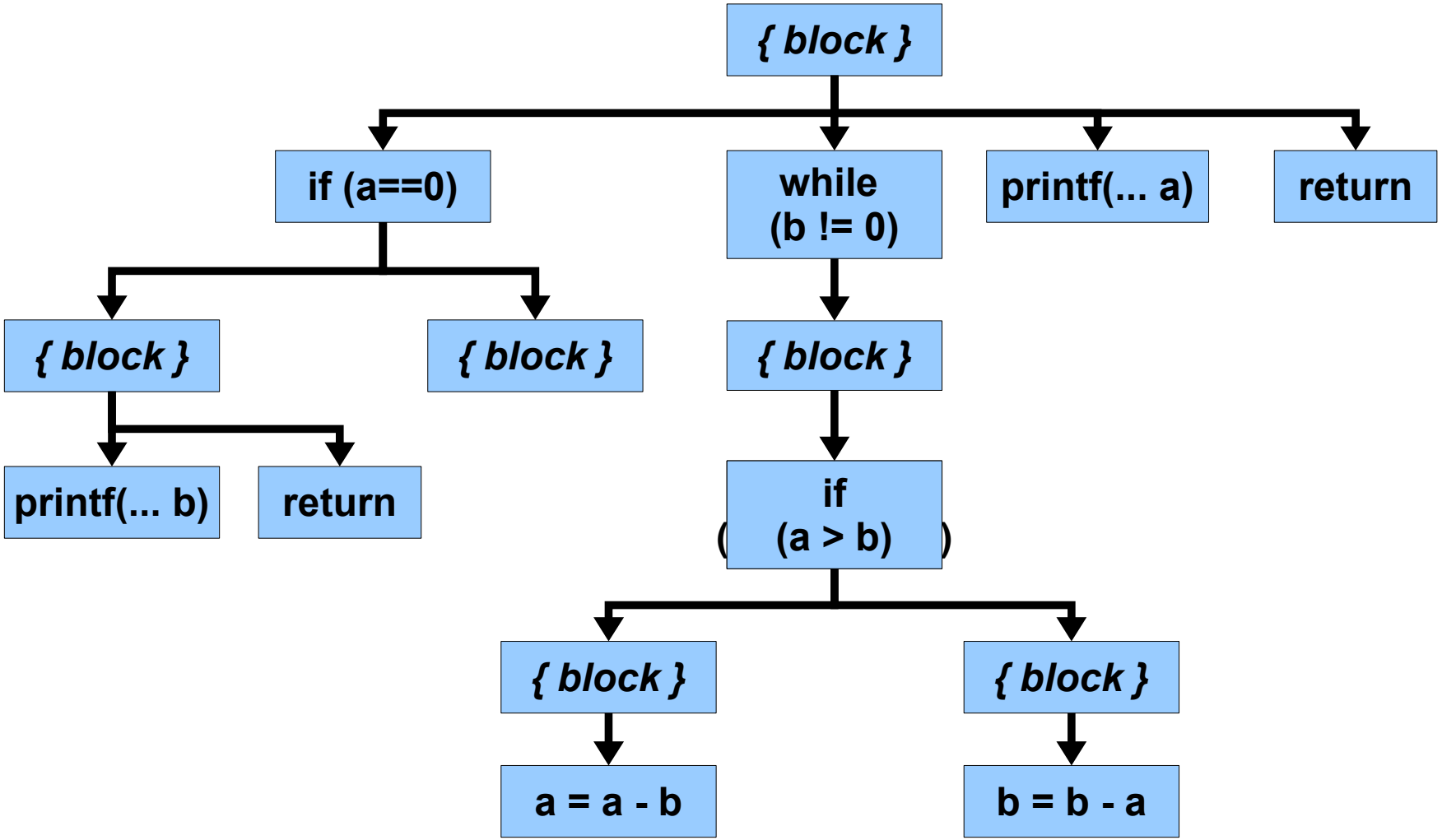
Example: Mutation (1/2)



Example: Mutation (2/2)



Example: Final Repair



Initial Program Repair Results

Program	LOC	Program Description	Defect Repaired	Time	# Fitness
gcd	22	Euclid's algorithm	infinite loop	276 s	909
zune	28	MS Zune example	infinite loop	78 s	460
uniq	1146	text processing	segfault	32 s	139
look-ultrix	1169	dictionary lookup	segfault	42 s	120
look-svr4	1363	dictionary lookup	infinite loop	51 s	42
units	1504	metric conversion	segfault	1528 s	9014
deroff	2236	document processing	segfault	132 s	227
nullhttpd	5575	webserver	heap buff. overflow	1394 s	1800
openldap io	6159	authentication server	non-overflow D.O.S.	665 s	8
leukocyte	6718	computational biology	segfault	544 s	12
indent	9906	source code processing	infinite loop	7614 s	13628
python complexobject	11227	web app. interpreter	overflow error	1393 s	23222
lighttpd fastcgi	13984	webserver CGI module	heap buff. overflow	395 s	52
imagemagick fx	16851	image processing	incorrect image output	1240 s	131
flex	18775	scanner generator	segfault	4660 s	9560
atris	21553	graphical game	stack buff. overflow	84 s	285
php string	26044	web app. interpreter	integer overflow	2678 s	18081
wu-ftpd	35109	FTP server	format string vuln.	5397 s	74
total	179369	(18 distinct programs)	(18 defects in 8 classes)	1567 s	4320

Research Questions

- Your results may not generalize to my situation because the programs and bugs I deal with ...
  - Have huge, long-running test suites (scalability)
  - Require high quality repairs (expressive power, overfitting)
  - Do not have source code (input assumptions, expressive power)
  - Are large and indicative (overfitting)
  - Require an economic business case

Large Test Suites and Fitness

- Thus far, test cases both
  - Validate a variant as a final repair
  - *and also* Determine which variants are retained
- What if there are 100+ test cases?
- Idea:
  - Use **all** tests to validate final repairs
  - Sample **some** tests to decide which variants are retained into the next iteration

Large Test Suites

- Use **test suite selection** techniques
  - Or random sampling
- Sampling reduces repair time by 81%
  - First evaluation on 10 bugs, 1200+ test cases
- "**`leukocyte`** was repaired in 6 minutes instead of over 90 minutes ... and **`imagemagick`** was repaired in 3 minutes instead of 36"
- Helps performance and correctness

Repair Quality

- Produced repairs pass ***all*** tests + minimization
- Objective measures
  - Retains required functionality
  - Does not introduce new bugs
  - Is not a “fragile memorization” of buggy input
- Subjective measures
  - Code review
  - Assurance argument

Repair Quality Benchmarks

- Two webservers with buffer overflows
  - **`nullhttpd`**, **`lighttpd`**
  - 138,226 held-out requests from 12,743 clients
- One web app language interpreter
  - **`php`** (integer overflow vulnerability)
  - 15 kLOC secure reservation system web app
  - 12,375 requests (held out)

Repair Quality Experiments

- Conduct repairs
  - Using ~10 test cases
- Evaluate repairs
  - Using all held-out test cases
  - Need same result bit-for-bit in same time or less
  - Also evaluate using held-out fuzz testing

Repair Quality Results

Program	Requests Lost Making Repair	Requests Lost to Repair Quality
nullhttpd	2.38% ± 0.83%	0.00% ± 0.25%
lighttpd	0.98% ± 0.11%	0.03% ± 1.53%
php	0.12% ± 0.00%	0.02% ± 0.02%

Repair Quality Results

Program	Requests Lost Making Repair	Requests Lost to Repair Quality	General Fuzz Tests Failed	Exploit Fuzz Tests Failed
nullhttpd	2.38% ± 0.83%	0.00% ± 0.25%	0 → 0	10 → 0
lighttpd	0.98% ± 0.11%	0.03% ± 1.53%	1410 → 1410	9 → 0
php	0.12% ± 0.00%	0.02% ± 0.02%	3 → 3	5 → 0

Repair Quality Results

Program	Requests Lost Making Repair	Requests Lost to Repair Quality	General Fuzz Tests Failed	Exploit Fuzz Tests Failed
nullhttpd	2.38% ± 0.83%	0.00% ± 0.25%	0 → 0	10 → 0
lighttpd	0.98% ± 0.11%	0.03% ± 1.53%	1410 → 1410	9 → 0
php	0.12% ± 0.00%	0.02% ± 0.02%	3 → 3	5 → 0
nullhttpd False Pos #1	7.83% ± 0.49%	0.00% ± 2.22%	0 → 0	n/a
nullhttpd False Pos #2	3.04% ± 0.29%	0.57% ± 3.91%	0 → 0	n/a
nullhttpd False Pos #3	6.92% ± 0.09% (no repair!)	n/a	n/a	n/a

Cherry Picking

- Thus far, all repaired programs were chosen arbitrarily by the authors
  - That is, the benchmark set may not be a representative sample of real-world programs
- Let's address that and program size concerns in one fell swoop with a systematic study
  - Use version control, take entire ranges of versions
  - Find all reproducible bugs within that range
  - Must be important enough for devs to fix and test
  - Try to repair them all

“Many Bugs” Benchmarks

Program	LOC	Tests	Defects	Description
fbc	97,000	773	3	legacy programming
gmp	145,000	146	2	multiple precision math
gzip	491,000	12	5	data compression
libtiff	77,000	78	24	image manipulation
lighttpd	62,000	295	9	web server
php	1,046,000	8,471	44	web programming
python	407,000	355	11	general programming
wireshark	2,814,000	63	7	network packet analyzer
total	**5,139,000**	**10,193**	**105**	

Table I

SUBJECT C PROGRAMS, TEST SUITES AND HISTORICAL DEFECTS: Tests were taken from the most recent version available in May, 2011; Defects are defined as test case failures fixed by developers in previous versions.

“Many Bugs” Results

Program	Defects Repaired	Avg. Cost per Non-Repair Hours	Avg. Cost per Non-Repair US$	Avg. Cost Per Repair Hours	Avg. Cost Per Repair US$
`fbc`	1 / 3	8.52	5.56	6.52	4.08
`gmp`	1 / 2	9.93	6.61	1.60	0.44
`gzip`	1 / 5	5.11	3.04	1.41	0.30
`libtiff`	17 / 24	7.81	5.04	1.05	0.04
`lighttpd`	5 / 9	10.79	7.25	1.34	0.25
`php`	28 / 44	13.00	8.80	1.84	0.62
`python`	1 / 11	13.00	8.80	1.22	0.16
`wireshark`	1 / 7	13.00	8.80	1.23	0.17
total	**55 / 105**	**11.22h**		**1.60h**	

Results can be reproduced for $403; successful repairs cost $7.32 on average.

Repair Quality and Bug Bounties

- All patches pass all available tests (e.g., 8471)
- Humans can still inspect patches [Weimer]
  - If this cuts developer time in half, the economic argument still holds ($25 vs. $25/2 + $7.32)
- One commercial developer paid $1,625 for
  - 63 repairs to “harmless” severity bugs
  - 11 repairs to “minor” severity bugs, 1 repair to “major”
  - 125 “false positive” candidate patches
  - Avg: $21 per non-trivial repair, one every 40 hours
  - **“Worth the money? Every penny.”**
  - http://www.daemonology.net/blog/2011-08-26-1265-dollars-of-tarsnap-bugs.html

Project Scope

- Two workshop paper (FoSER, SBST)
- Two journal paper (TSE, Comm. ACM)
- Four conference papers (ASE, ASE, GECCO, ICSE)
- Three best paper awards (ICSE, GECCO, SBST)
- Some associated papers (SIGGRAPH, etc.)
- IFIP TC2 Manfred Paul Award (1024 Euros)
- Gold Human Competitive Award ($5000)
- Four grant proposal awards
- ... since 2009.

Discussion Section

- **Possible stories:**
- Boring grant meeting
- Fail Early
- Bug finding fatigue
- Early experimental result problems
- Don't be all things to all people
- Dagstuhl Seminar
- Tom Ball predictions

Homework

- Project Writeup Due!
  - Need help? Stop by my office or send email.
- Project Presentation Due