Automatically Finding Patches Using Genetic Programming

Westley Weimer,
Stephanie Forrest,
Claire Le Goues,
ThanVu Nguyen,
Ethan Fast,
Briana Satchell,
Eric Schulte

Motivation

- Software Quality remains a key problem
  - Over one half of 1 percent of US GDP each year [NIST02]
  - The cost of fixing a defect increases ($25 - $16k) [IBM08]
  - Even security-critical bugs take 28 days (avg) [Symantec06]
  - 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
- Thus: Automated Patch Generation

Main Claim

- We can automatically and efficiently repair certain classes of bugs in off-the-shelf, unannotated legacy programs.
- Basic idea: Biased search through the space of certain nearby programs until you find a variant that repairs the problem. Key insights:
  - Use existing **test cases** to evaluate variants.
  - Search by perturbing parts of the program **likely** to contain the error.

ICSE'09 Best Paper, GECCO'09 Best Paper, SBST'09 Best Short Paper, 2009 IFIP TC2 Manfred Paul Award, 2009 Gold Human-Competitive Award

Repair Process Preview

- Input:
  - The program source code
  - System/regression tests passed by the program
  - A test case failed by the program (= the bug)
- Genetic Programming Work:
  - Create variants of the program
  - Run them on the test cases
  - Repeat, retaining and combining variants
- Output:
  - New program source code that passes all tests
  - *or* “no solution found in time”

This Talk

- Fixing Real Bugs In Real Programs
  - Representation and Operations
- The Quality of Automated Repairs
  - Self-Healing Systems and Metrics
- Test Suite Selection
  - Success and Explanations
- Open Questions in Automated Repair

Genetic Programming

- **Genetic programming** is the application of evolutionary or genetic algorithms to program source code.
  - Representing a population of program variants
  - Mutation and crossover operations
  - Fitness function
- GP serves as a search heuristic
  - Others (random search, brute force, etc.) also work
- Similar in ways to search-based software engineering:
  - Regression tests to guide the search

Useful Insight #1 – Where To Fix

- In a large program, not every line is equally likely to contribute to the bug.
- **Fault localization**: given a bug, find its location in the program source.
- Insight: since we have the test cases, run them and collect coverage information.
- The bug is more likely to be found on lines visited when running the failed test case.
- The bug is less likely to be found on lines visited when running the passed test cases.

Useful Insight #2 – How To Fix

- Developers often use statements or lines of code as atomic units representing actions
- Insight: operate on statements or lines
  - Not on assembly ops or expressions
  - Factor of 10 reduction in search space each time
- Insight: do not invent new code
  - Instead, copy and modify existing statements
- We assume the program "contains the seeds of its own repair"
  - e.g., has another null check somewhere

Fault Localization Formalism

- We define a weighted path to be a list of <statement, weight> pairs.
- We use this weighted path:
  - The statements are those visited during the failed test case.
  - The weight for a statement S is
    - High (1.0) if S is not visited on a passed test
    - Low (0.0-0.1) if S is also visited on a passed test
  - (Other weight sources are possible: e.g., Cooperative Bug Isolation or Daikon predicates)

Genetic Programming for Program Repair: Mutation

- Population of Variants:
  - Each variant is an <AST, weighted path> pair
- Mutation:
  - To mutate a variant V = <AST_V, wp_V>, choose a statement S from wp_V biased by the weights
  - **Replacement**. Replace S with S1
  - **Insertion**. Replace S with { S2 ; S }
  - **Deletion**. Replace S with { }
  - Choose S1 and S2 from the entire AST
  - All variants retain weighted path length

Genetic Programming for Program Repair: Fitness

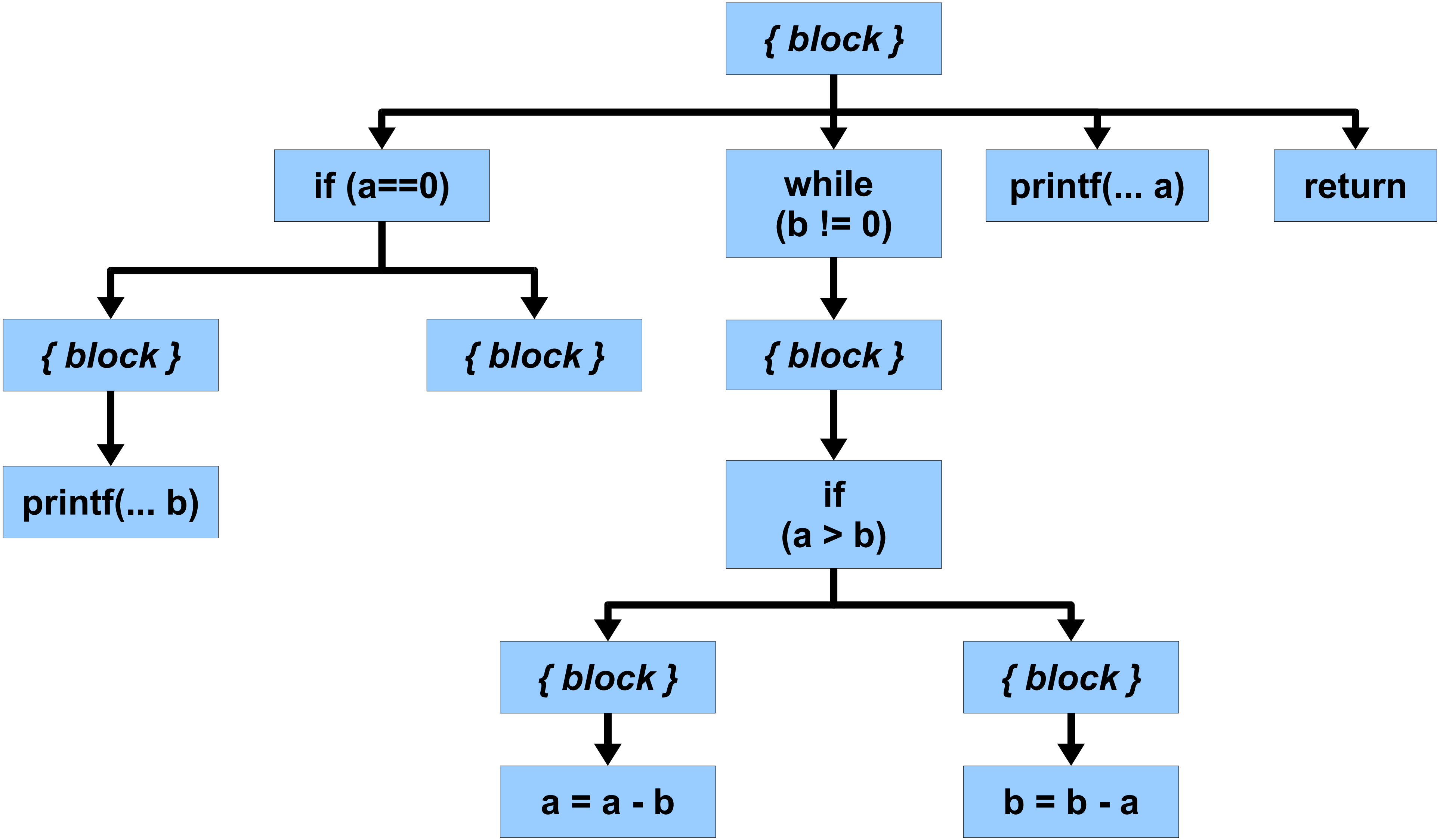
- Compile a variant
  - If it fails to compile, Fitness = 0
  - Otherwise, run it on the test cases
  - Fitness = number of test cases passed
  - Weighted: passing the bug test case is worth more
- Selection and Crossover
  - Higher fitness variants are retained and combined into the next generation
  - Tournament selection and one-point crossover
- Repeat until a solution is found

Example: GCD

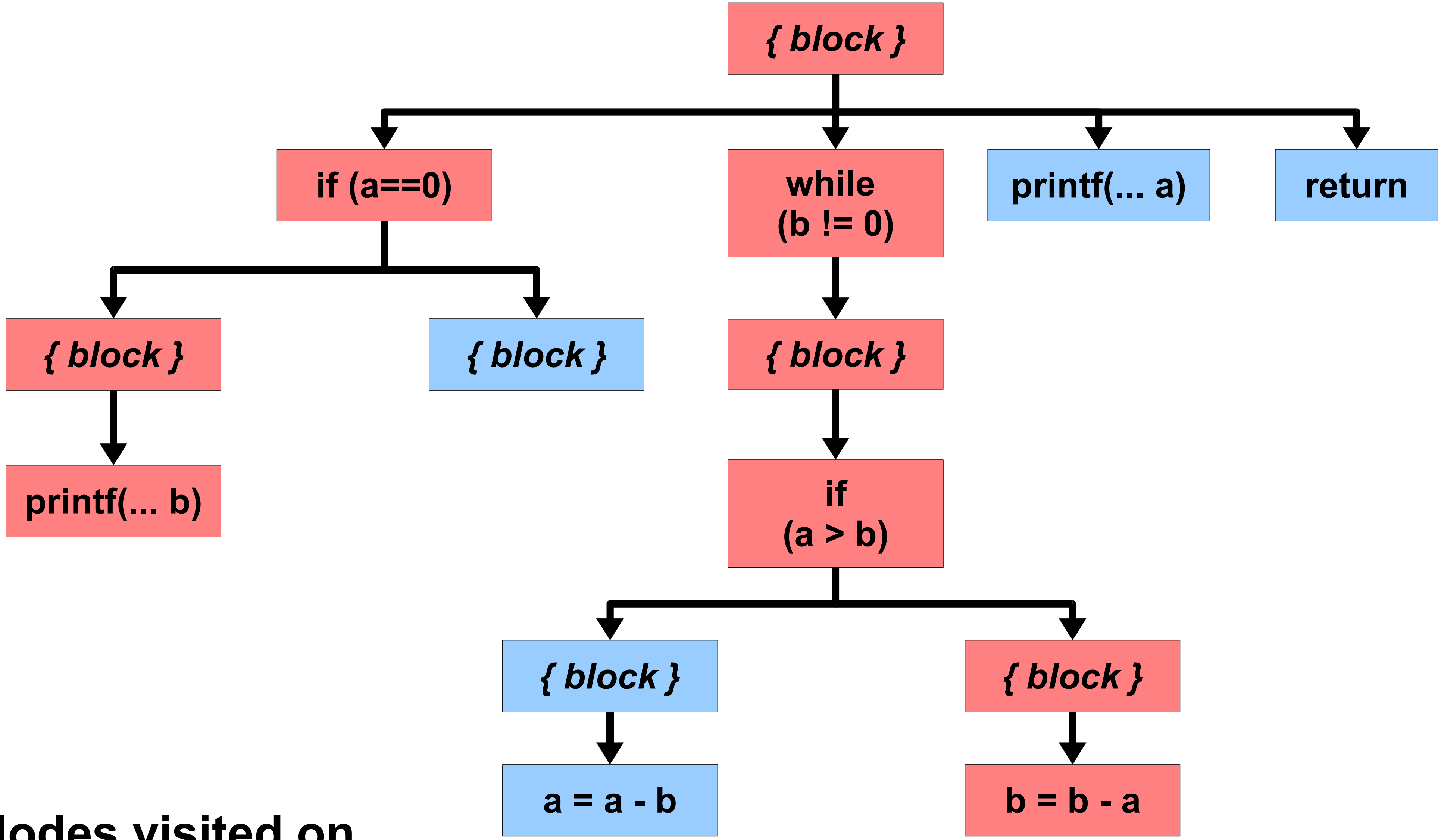
```
/* requires: a >= 0, b >= 0 */
void print_gcd(int a, int b) {
  if (a == 0)
    printf(“%d”, b);
  while (b != 0) {
    if (a > b)
      a = a – b;
    else
      b = b – a;
  }
  printf(“%d”, a);
  return;
}
```

Bug: when a==0 and b>0, it loops forever!

Example: Abstract Syntax Tree

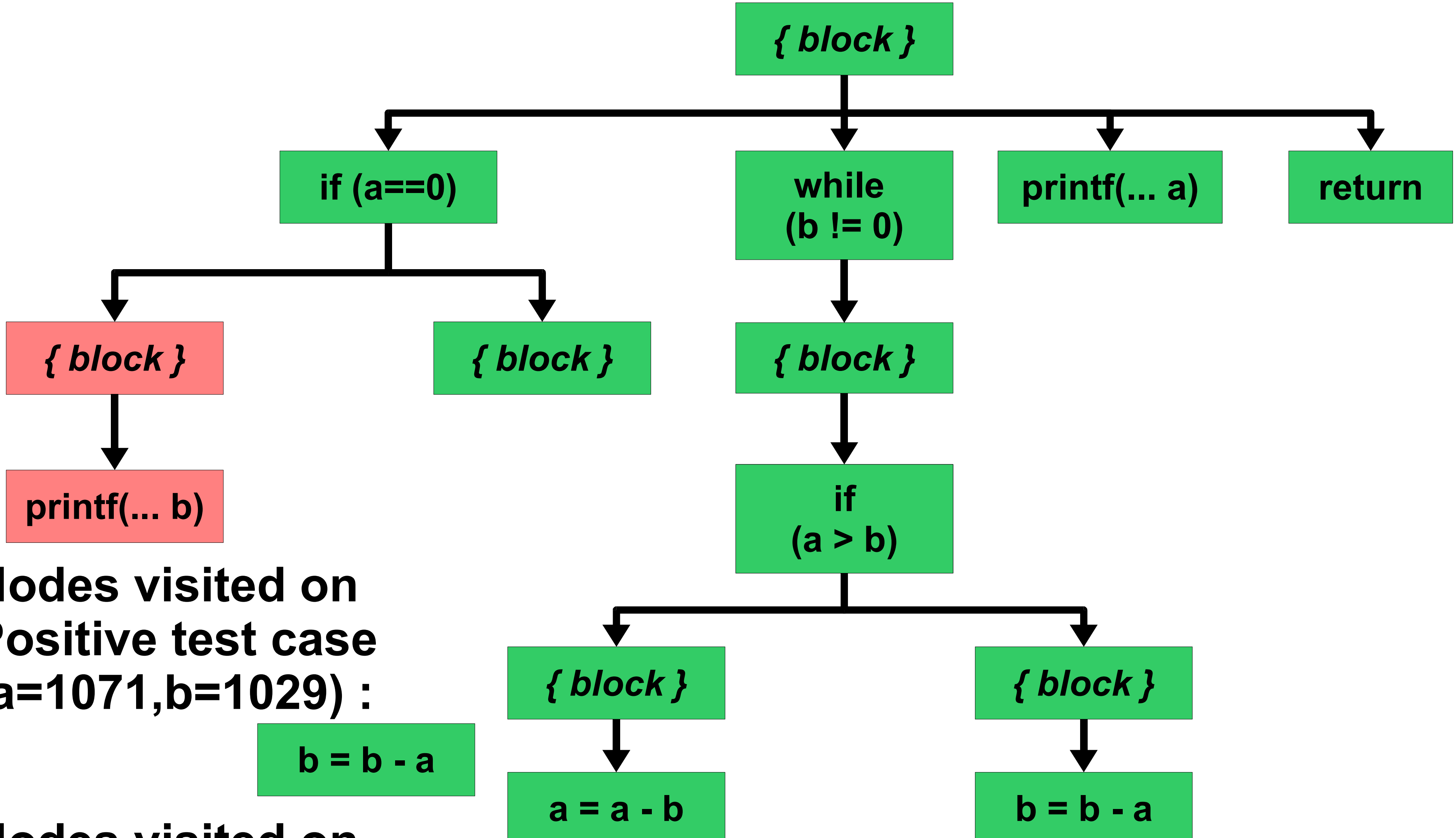


Example: Weighted Path (1/3)



Nodes visited on Negative test case (a=0,b=55) : (printf ...b)

Example: Weighted Path (2/3)



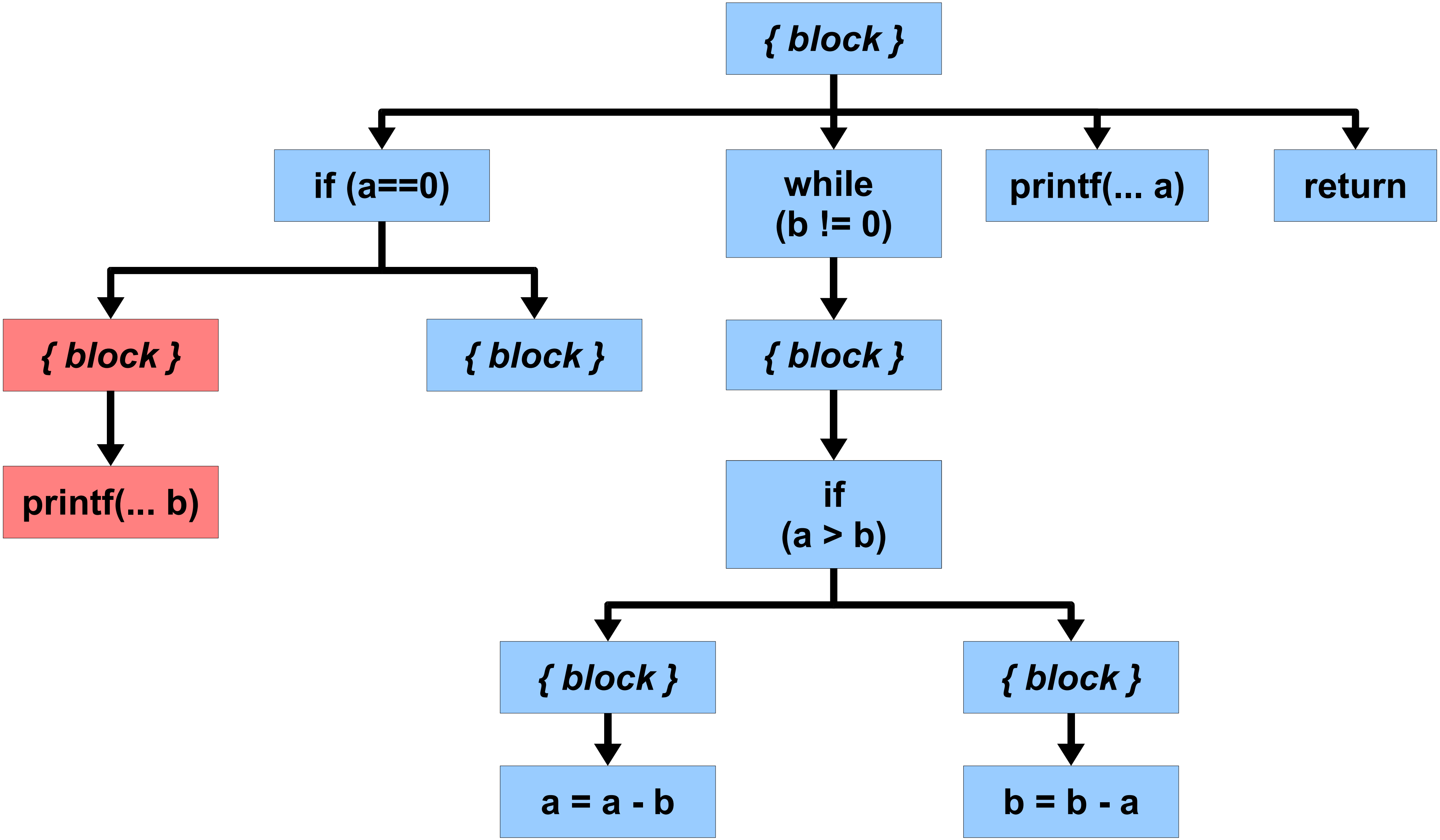
Nodes visited on Positive test case (a=1071,b=1029) :

b = b - a

Nodes visited on Negative test case (a=0,b=55) :

(printf ...b)

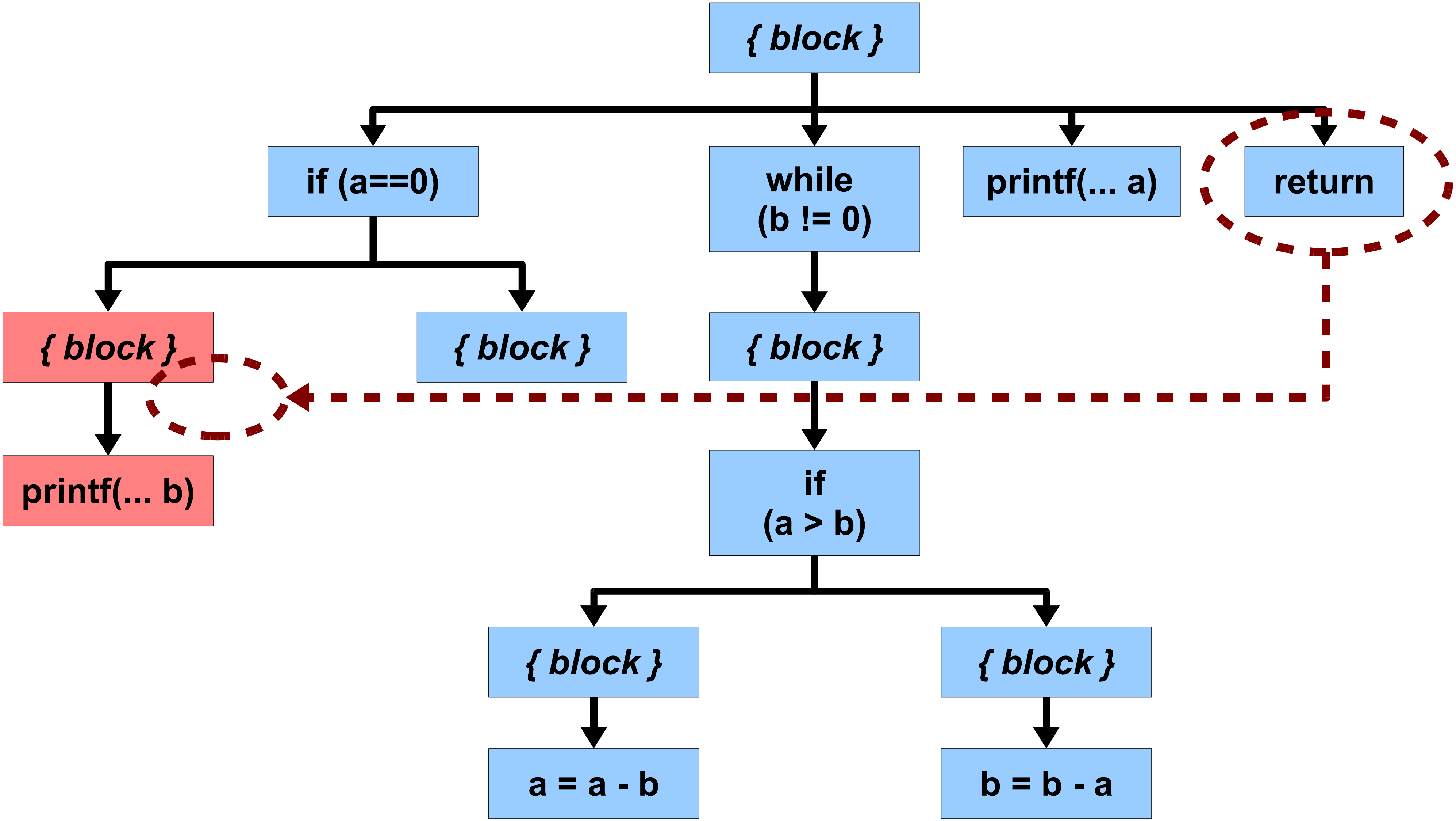
Example: Weighted Path (3/3)



Weighted Path:

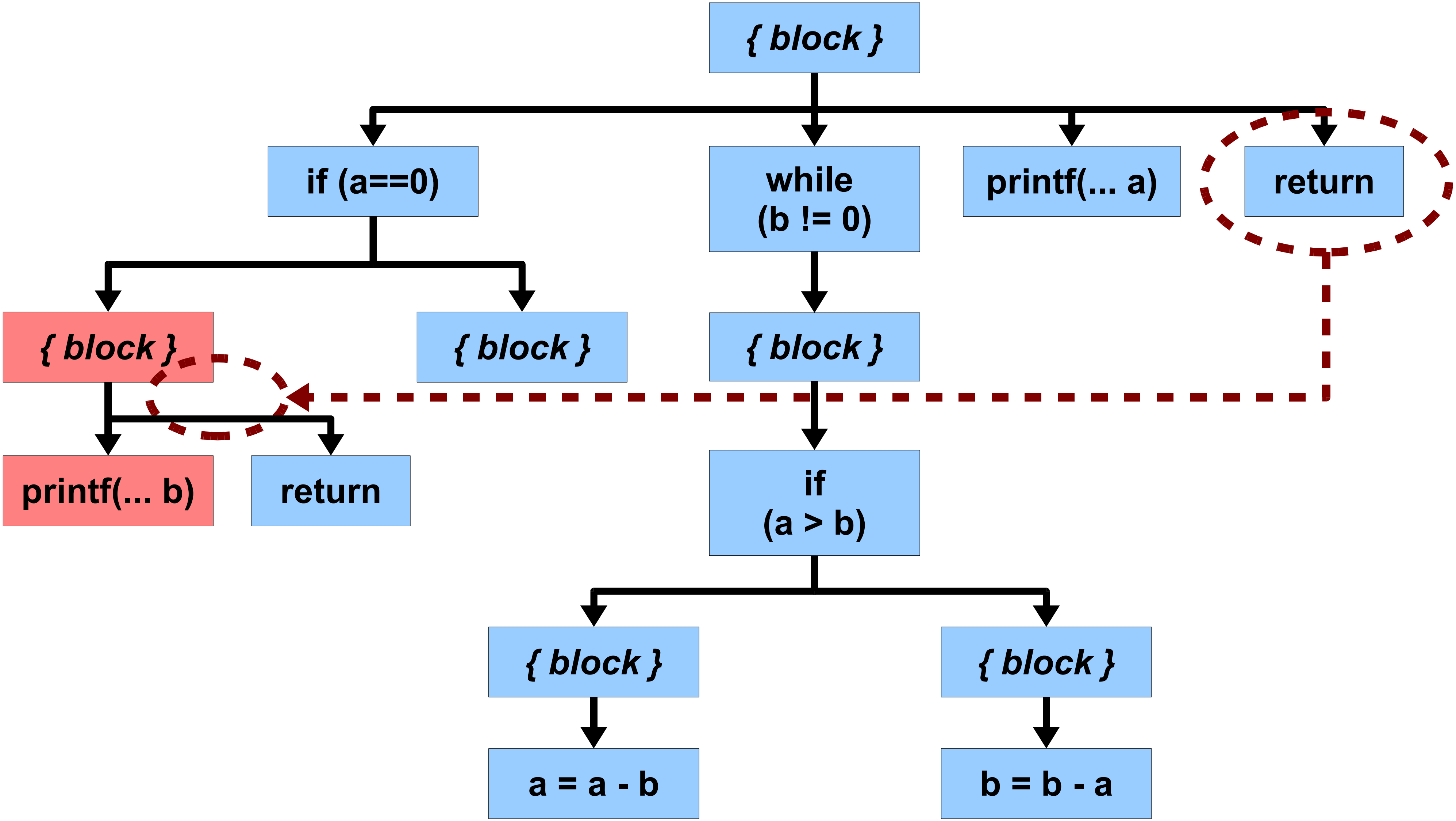
(printf ...b)

Example: Mutation (1/2)



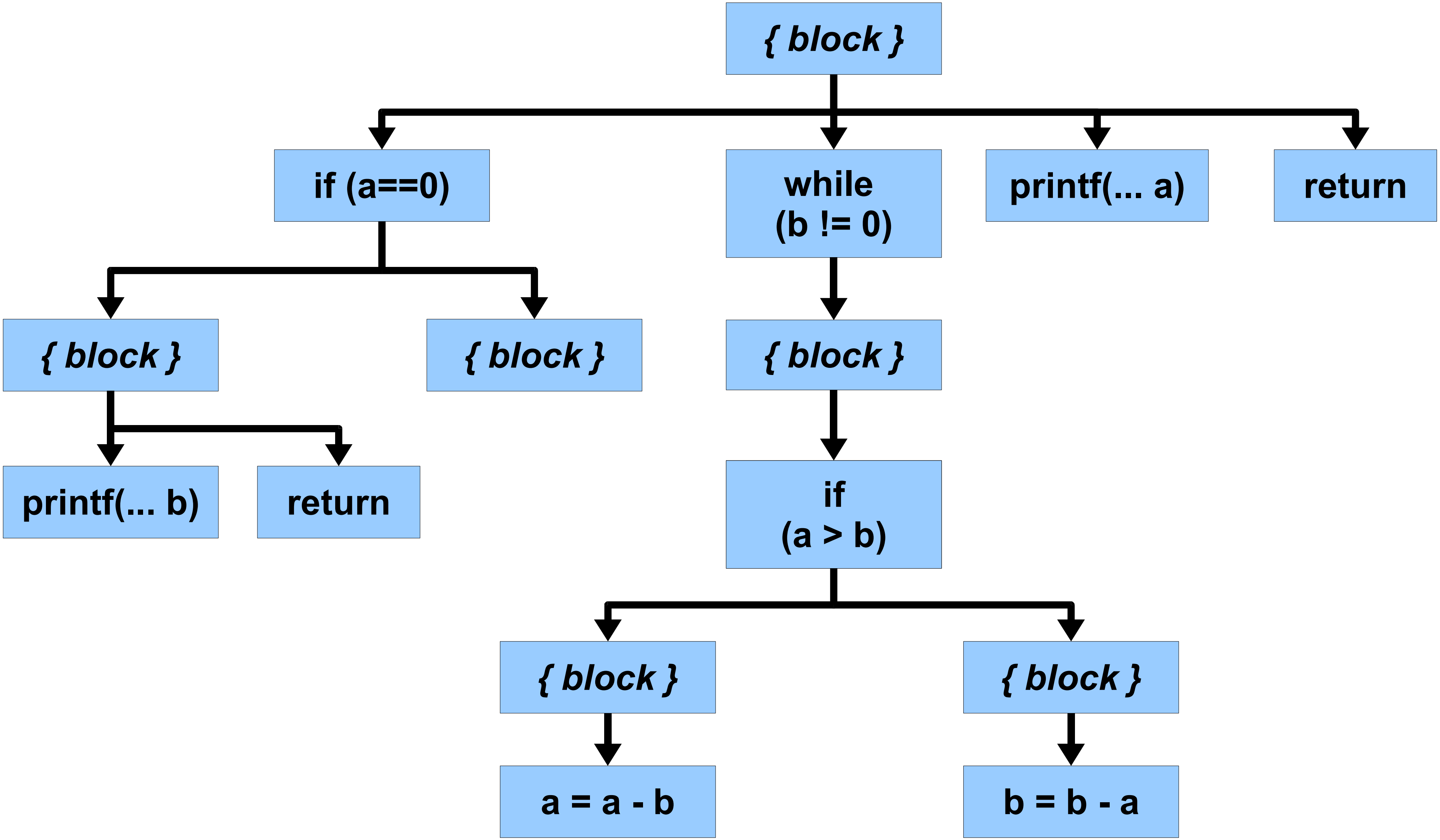
Mutation Source: Anywhere in AST
Mutation Destination: Weighted Path

Example: Mutation (2/2)



Mutation Source: Anywhere in AST
Mutation Destination: Weighted Path

Example: Final Repair



Minimize The Repair

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

Experimental Results: 20 Repairs

Program	Lines of Code		Program Description	Defect Repaired	Time	# of
	Repaired	Total				Fitness
gcd	22	-	Euclid's algorithm	infinite loop	276 s	909
zune	28	-	MS Zune example	infinite loop	78 s	460
tiff	1084	84067	image processing	segfault	45 s	280
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	-	threaded webserver	heap buff. overflow †	1394 s	1800
openldap	6159	308777	authentication server	non-overflow D.O.S. †	665 s	8
ccrypt	6413	7515	Rijndael cryptography	segfault	330 s	32
leukocyte	6718	-	computational biology	segfault	544 s	12
indent	9906	-	pretty printing	infinite loop	7614 s	13628
python	11227	496527	web app. interpreter	overflow error	1393 s	23222
lighttpd	13984	51895	webserver CGI	heap buff. overflow †	395 s	52
imagemagick	16851	450416	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	26044	797749	web app. interpreter	integer overflow †	2678 s	18081
wu-ftpd	35109	-	FTP server	format string vuln. †	5397 s	74
total or avg	186,866	2,302,050	(20 distinct programs)	(20 defects in 8 classes)	1428 s	3903

Many defects from "black hat" lists; avg minimization time: 12 seconds.

The Story Thus Far

- How does the approach work?
  - Create programs in a **restricted** search space
- Can it produce repairs?
  - Yes, for many types of programs and defects

- **Can I afford to use it?**
  - Are the repairs trustworthy?
  - Does the approach scale?

Repair Quality

- Repairs are typically *not* what a human would have done
  - Example: our technique adds bounds checks to one particular network read, rather than refactoring to use a safe abstract string class in multiple places
- Recall: any proposed repair must pass **all** regression test cases
  - When POST test is omitted from nullhttpd, the generated repair eliminates POST functionality
  - Tests ensure we do not sacrifice functionality
  - Minimization prevents gratuitous deletions
  - Adding more tests helps rather than hurting

Repair Quality Experiment

- A high-quality repair ...
  - Retains required functionality
  - Does not introduce new bugs
  - Is not a “fragile memorization” of the buggy input
  - Works as part of an entire system
- If humans are present, they can inspect it
- Let's consider a human-free situation, such as:
  - A long-running server with an anomaly intrusion detection system that will generate and deploy repairs for all detected anomalies.

Repair Quality Benchmarks

- Two webservers with buffer overflows
  - **`nullhttpd`** (simple, multithreaded)
  - **`lighttpd`** (used by Wikimedia, etc.)
  - 138,226 requests from 12,743 distinct client IP addresses (held out; one day of data)
- One web application language interpreter
  - **`php`** (integer overflow vulnerability)
  - 15kloc secure reservation system web app
  - 12,375 requests (held out; one day of data)

Repair Quality Experimental Setup

- Apply indicative workloads to vanilla servers
  - Record result contents and times
- Send attack input
  - Caught by anomaly intrusion detection system
- Generate and deploy repair
  - Using attack input and six test cases
- Apply indicative workload to patched server
  - Each request must yield exactly the same output (bit-per-bit) in the same time or less!

Closed-Loop Outcomes

Case	Anomaly Detected?	Successful Repair?	Result
1	True Neg.	N/A	Legitimate request handled correctly; no repair
2	False Neg.	N/A	Attack succeeds; Repair not attempted
3	True Pos.	Yes	Attack stopped and bug fixed. Later requests could be lost if repair breaks functionality
4	True Pos.	No	Attack detected; bug not repaired
5	False Pos.	No	Legitimate request dropped; repair not found
6	False Pos.	Yes	Legitimate request dropped; later requests may be harmed if “repair” is incorrect

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

Repair Quality Conclusions

- It is possible to create repairs that
  - Retain required functionality
  - Do not introduce new bugs
  - Are not a fragile memorizations
  - Work as part of an entire system
- This reduces to the problem of supplying a good test suite
- For webservers and php, a few indicative end-to-end system tests suffice
  - But in general we may need more test cases ...

Algorithm Scalability

- We want to quickly produce high-quality repairs for complicated defects in large programs with arbitrary test suites
- GP is a heuristic search strategy
- Worst-case run time is effectively
  - Size of search space multipled by
  - Time to evaluate a point in the search space
- Examine fitness cost first, then search space

Fitness Scalability

- 1000 fitness evaluations means 1000 complete runs of your test suite
  - This task can be done in parallel (two ways)
- We view it as an advantage that we can repair programs with only a few test cases
- But we want to scale to more larger test suites
- For both performance and correctness
  - Test cases encode required behavior!

Test Suite Purposes

- Thus far, the full test suite determines:
  - Do we keep this variant in the next generation?
  - Is this a candidate repair that passes all tests?
- Insight: split these tasks
  - Use a small subset of tests to decide “keep/drop”
  - GP structure allows noise
  - Use the full suite to evaluate candidate repairs

Test Suite Selection

- Can choose subset at random or by some other metric (e.g., max coverage, min time)
- In intermediate steps, poor test selection:
  - Retains variants that should be dropped
  - Drops variants that should be kept (rare)
  - Distorts the view of the fitness landscape
- Thus requiring more generations
- Does the time saved in fitness evaluations exceed the cost of being “led astray” ?

Tests Suite Selection Algorithms

- Random Subset
  - Pick next test at random without replacement
- Time-Aware Test Suite Prioritization
  - Includes test time and test coverage

    (Walcott, Soffa, Kapfhammer, Roos. ISSTA'06)
- Greedy Coverage
  - Pick next test to maximize coverage gains

Test Suite Selection Results

- 10 programs, each with 100+ test cases

Program	Total LOC	Module LOC	# Tests	Test Suite Description	Defect Description
`deroff` utx 4.3	2236	2236	100	fuzz input (as typesetting directives)	segfault
`look` utx 4.3	1169	1169	100	fuzz input (for both needle and haystack strings)	segfault
`uniq` utx 4.3	1146	1146	100	fuzz input (as duplicate-containing text file)	segfault
`zune`	28	28	100	fuzz input (as days since 1 January 1980)	infinite loop
`gcd`	22	22	100	fuzz input (as pairs of integers)	infinite loop
`lighttpd` 1.4.15	51895	3829	200	HTTP requests from `cs.virginia.edu`	buffer overrun
`nullhttpd` 0.5.0	5575	5575	200	HTTP requests from `cs.virginia.edu`	buffer overrun
`leukocyte`	6718	6718	100	video microscopy images	segfault
`tiff` 3.8.2	84067	1084	106	image suite bundled with `tifflib`	segfault
`imagemagick` 6.5.2	450416	5858	100	images from `pngsuite` test suite	wrong output
total	603272	27665	1206		

- Selection reduces time-to-repair by 81%
  - Yields equivalent-quality repairs
  - leukocyte with 100 tests: 90 mins to 6 mins
  - imagemagick with 100 tests: 36 mins to 3 mins

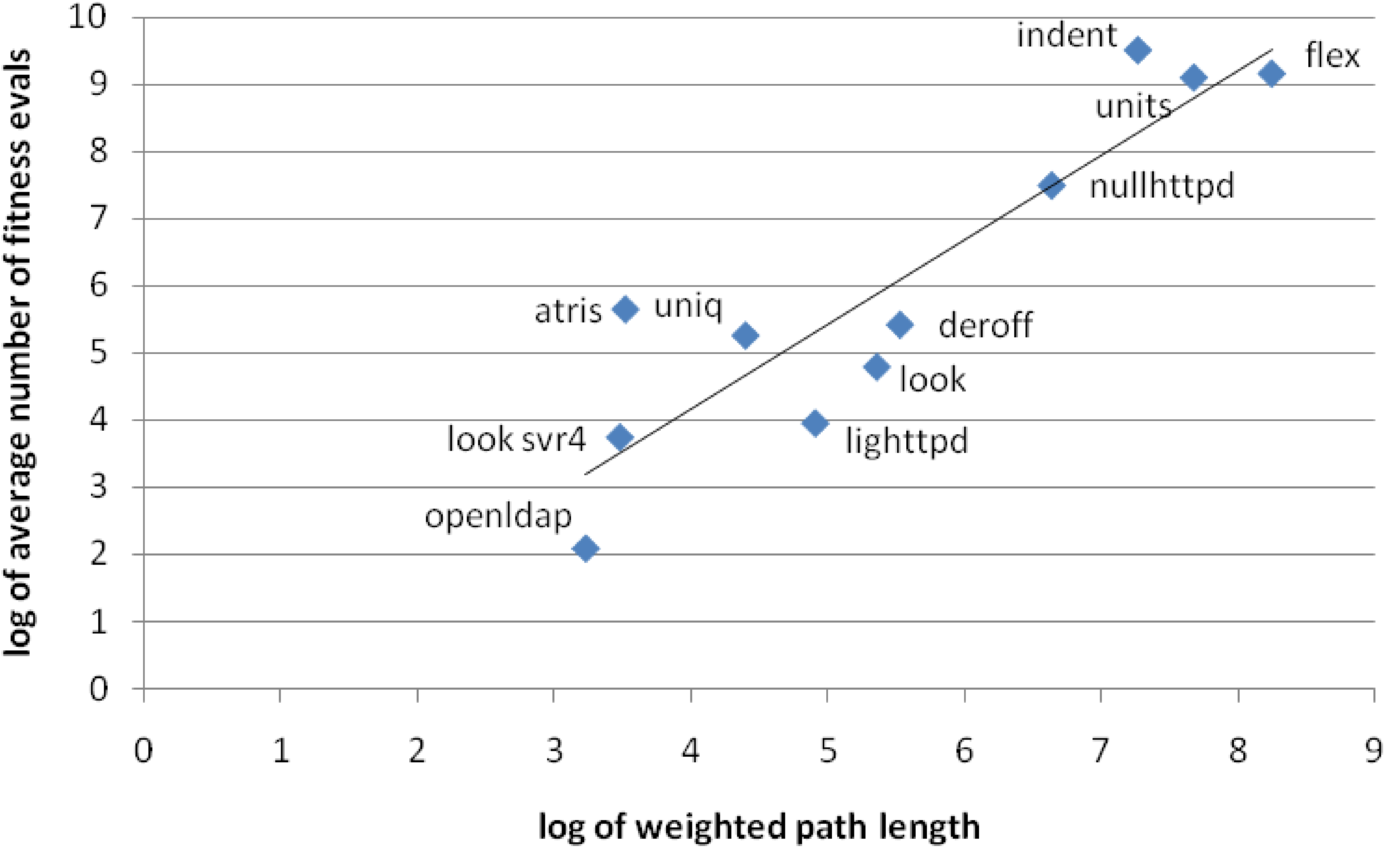
Test Suite Selection Explained

- Small changes between variants mean most variants have similar test case behavior
  - True. However, an optimal safe impact analysis could only reduce time-to-repair by 29% (cf. 81%)
- The test cases are all dependent, and thus running one is as good as running another
  - False. There was only a 3% performance increase between high-overlap and low-overlap suites.
- The fitness function can tolerate noise
  - True. Test suite selection on gcd distorts the fitness function by 27%. (cf. Fitness Distance Correlation)

Search Space Scalability

- So each variant can be evaluated rapidly
- The other factor in cost is the number of variants examined
  - i.e., the size of the search space
- This is related to fault localization precision, not overall program size
  - Since we only mutate and crossover statements along the weighted path

Search Space vs. Fault Localization



Outline

- Fixing Real Bugs In Real Programs
  - Representation and Operations
- The Quality of Automated Repairs
  - Self-Healing Systems and Metrics
- Test Suite Selection
  - Success and Explanations
- **Open Questions in Automated Repair**

Can Formal Specifications Be Used?

- Use Local Annotations in Mutation
- Typestate Repair
  - Algorithms for repairing programs with respect to a temporal safety policy
  - Provably safe with respect to that one policy
- Synthesis
  - Use GP to identify regions, not to copy statements
- Refactoring for Formal Verification
  - If repair is correct but cannot easily be verified

What Fault Localization is Possible?

- Standard approaches (e.g., Tarantula)
- Cooperative Bug Isolation
  - Instrument program with Daikon-style predicates
  - Measure which are false on normal runs but true on failing runs (etc.)
  - Have repaired a program using a weighted path induced from CBI information
- Impact on mutation operator:
  - Guide changes to flip predicates

What Mutations are Possible?

- Goal: increase "expressive power"
- Expression-level Mutation
  - Increases size of search space
  - In practice, reduces time-to-repair by 30%
    - "x=3;" vs. "x=0; x++; x++; x++;"
- "Typed" Repair Templates
  - "if (local_ptr != NULL) { local_stmt(local_ptr); }"
  - Manually crafted or mined automatically
    - Distance metric on changes

Can We Handle Threads?

- Currently we assume a deterministic fitness function and the ability to localize faults.
- VM Integration
  - Add scheduler constraints to the representation
  - Repair = code changes plus scheduler directives
- Context-Bounded
  - Existing tools can prove the presence or absence of race conditions assuming at most *k* thread interleavings

Can We Do More Than Repair?

- Evolutionary approaches are traditionally strong at small optimization and synthesis
- Vertex and Pixel Shaders
  - Small C programs used by modern graphics cards
  - Optimize for space or speed
  - Can be "10% blurrier" than original
  - What is "fault localization" here?

Is Our Fitness Function Reasonable?

- A good fitness function increases for more desirable variants
- Ours is accurate at the extremes
  - But weak in the middle
- Correlation with an “optimal” tree structured distance metric for known repairs is ~0.3
- Can we combine test case counts with ...
  - Invariants retained, anomaly detection signals, ...

Is Competitive Coevolution Possible?

- In the security domain, white hats and black hats both want to identify the next attack as quickly as possible
- Can we simulate this “arms race”?
  - Many exploits are themselves C programs
  - For a small exploitable program we have evolved a repair, then evolved the exploit to work again, then evolved a second repair
- Automated Hardening and Synthetic Diversity
  - Repair old programs against various signals
  - Does it defeat attacks that came out later?

Conclusions

- We can automatically and efficiently repair certain classes of bugs in off-the-shelf legacy programs.
    - 20 programs totaling 186kloc in about 5 minutes each, on average
- We use regression tests to encode desired behavior.
    - Existing tests encode required behavior
- The genetic programming search focuses attention on parts of the program visited during the bug but not visited during passed test cases.

Questions

- I encourage difficult questions.

Bonus Slide: Test Cases

```
1 #!/bin/sh
2 # Positive Test Case for nullhttpd (POST data)
3 ulimit -t 5
4 /usr/bin/wget --tries=1 --post-data 'name=my_name&submit=submit'
5   "http://localhost:\$PORT/cgi-bin/hello.pl"
6 if diff hello.pl ../known-good-hello.pl-result ; then
7   # if the current output matches the known-good output
8   echo "passed hello.pl test case" >> ../list-of-tests-passed
9 fi
```

Figure 2: Positive test case for `nullhttpd`. `wget` is a command-line HTTP client; `ulimit` cuts the test off after five seconds. The test assumes that the sandboxed webserver is accepting connections on `PORT` and has its own copy of `htdocs`, including `cgi-bin/hello.pl`. Note the oracle comparison using `diff` against `known-good-hello.pl-result` on line 6.

```
1 #!/bin/sh
2 # Negative Test Case for nullhttpd
3 ulimit -t 5
4 ../nullhttpd-exploit -h localhost -p $PORT -t2
5 /usr/bin/wget --tries=1 "http://localhost:$PORT/index.html"
6 if diff index.html ../known-good-index.html-result ; then
7   # if the current output matches the known-good output
8   echo "passed exploit test case" >> ../list-of-tests-passed
9 fi
```

Figure 3: Negative test case for `nullhttpd`. If the exploit (line 4) disables the webserver then the request for `index.html` (line 5) will fail.

Evolution of Zune Repair

(5 normal test cases weighing 1 each,
2 buggy test cases weighing 10 each)