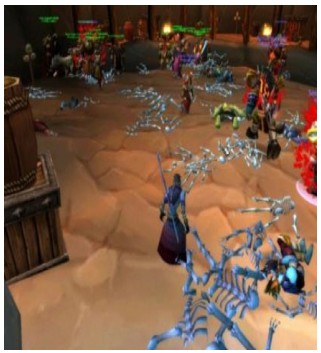
Automatic Program Repair using Evolutionary Computing

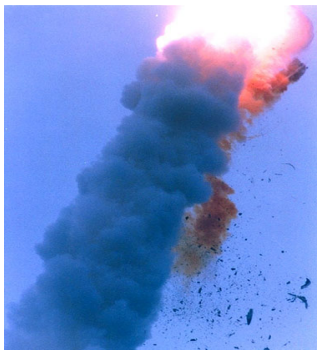
ThanhVu Nguyen

September 26, 2017

The Problem

SOFTWARE BUGS





The Cost

– Mozilla Developer

"Everyday, almost 300 bugs appear [..] far too many for only the Mozilla programmers to handle."

Software bugs annually cost 0.6% of the U.S GDP and $312 billion to the global economy

Average time to fix a security-critical error: 28 days

How do we repair bugs now?

- Ignore them
- Pay expensive programmers to fix them manually
- Develop tools to help the programmers
  - Debuggers, profilers, smart compilers
  - Type checkers
- Build mathematical models to analyze program correctness
  - Don’t scale up to production software

Stephanie Forrest

Claire Le Goues

Westley Weimer

GenProg: Evolutionary Program Repair

- A generic method for automated software repair
- Uses genetic algorithm
- Works with legacy code
- Does not assume formal specifications

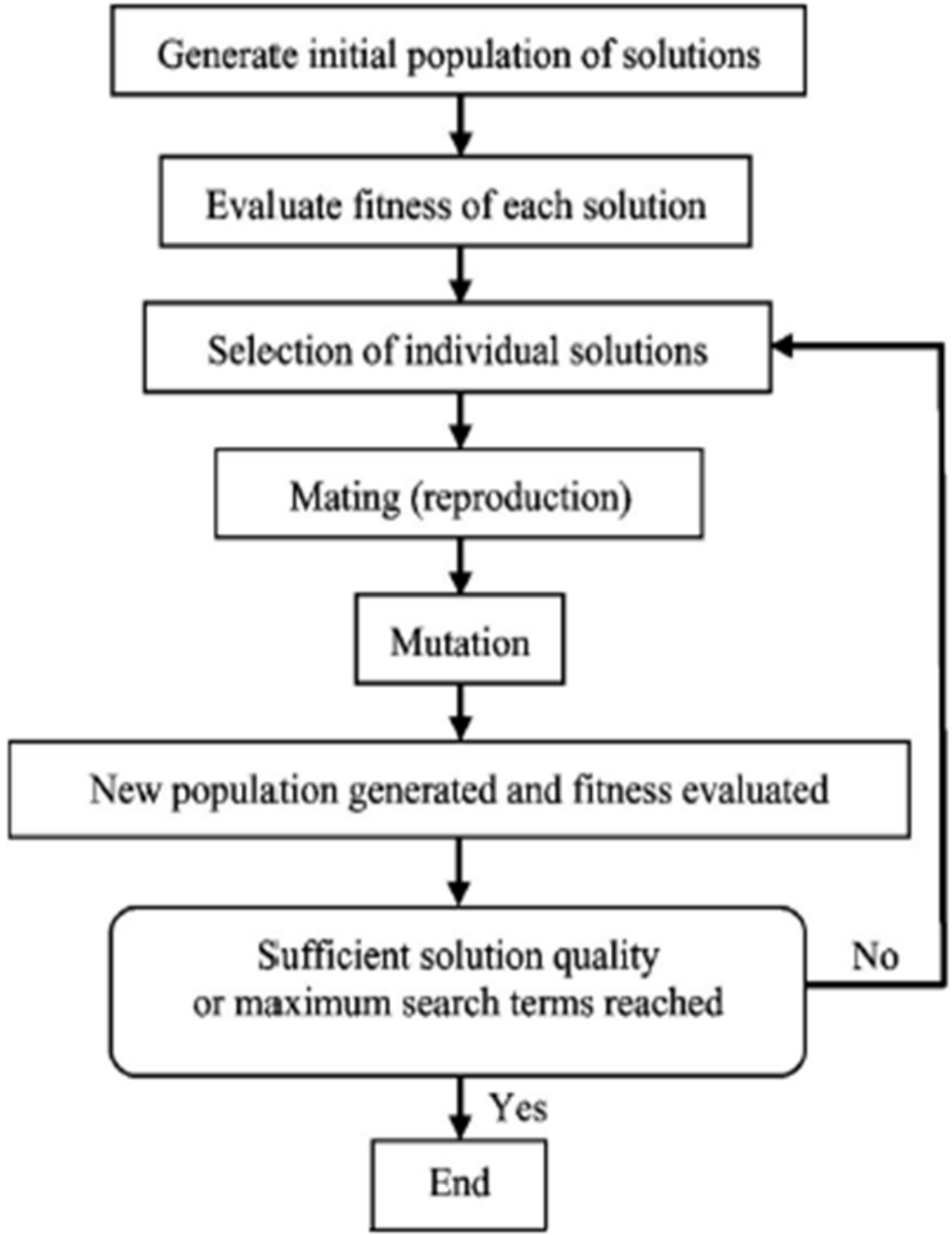
Darwinian Evolution in a Computer

Genetic Algorithm

- Developed by John Holland (1975)
- A search heuristic that mimics the process of natural evolution
- Uses concepts of *natural selection* and *genetic inheritance* (Darwin 1859)

Widely-used in business, science, and engineering

- Optimization and search problems
- Routing, scheduling, and timetabling
- Evolvable hardware, encryption, robotics


Generate initial population of solutions
Evaluate fitness of each solution
Selection of individual solutions
Mating (reproduction)
Mutation
New population generated and fitness evaluated
Sufficient solution quality
or maximum search terms reached
No
Yes
End

Example: maximize the number of ones in a bit string

1. Encoding of an individual: as a bit vector, e.g., $i = 1111010101$

Example: maximize the number of ones in a bit string

1. Encoding of an individual: as a bit vector, e.g., $i = 1111010101$
2. Initial population: create n *random* invididuals
   $pop = \{i_1 = 1111010101, i_2 = 0111000101, i_3 = 0010001000, \ldots, i_n = 0010110111\}$

Example: maximize the number of ones in a bit string

1. Encoding of an individual: as a bit vector, e.g., $i = 1111010101$
2. Initial population: create n *random* invididuals
   $pop = \{i_1 = 1111010101, i_2 = 0111000101, i_3 = 0010001000, \ldots, i_n = 0010110111\}$
3. Fitness: the # of 1's in an individual
   $f(i_1) = 7, f(i_2) = 5, f(i_3) = 2, \ldots, f(i_n) = 6$

Example: maximize the number of ones in a bit string

1. Encoding of an individual: as a bit vector, e.g., $i = 1111010101$
2. Initial population: create n *random* invididuals
   $pop = \{i_1 = 1111010101, i_2 = 0111000101, i_3 = 0010001000, \ldots, i_n = 0010110111\}$
3. Fitness: the # of 1's in an individual
   $f(i_1) = 7, f(i_2) = 5, f(i_3) = 2, \ldots, f(i_n) = 6$
4. Selection: choose n individuals based on their fitnesses
   $pop' = \{i_1, i_n, i_1, i_2, \ldots\}$

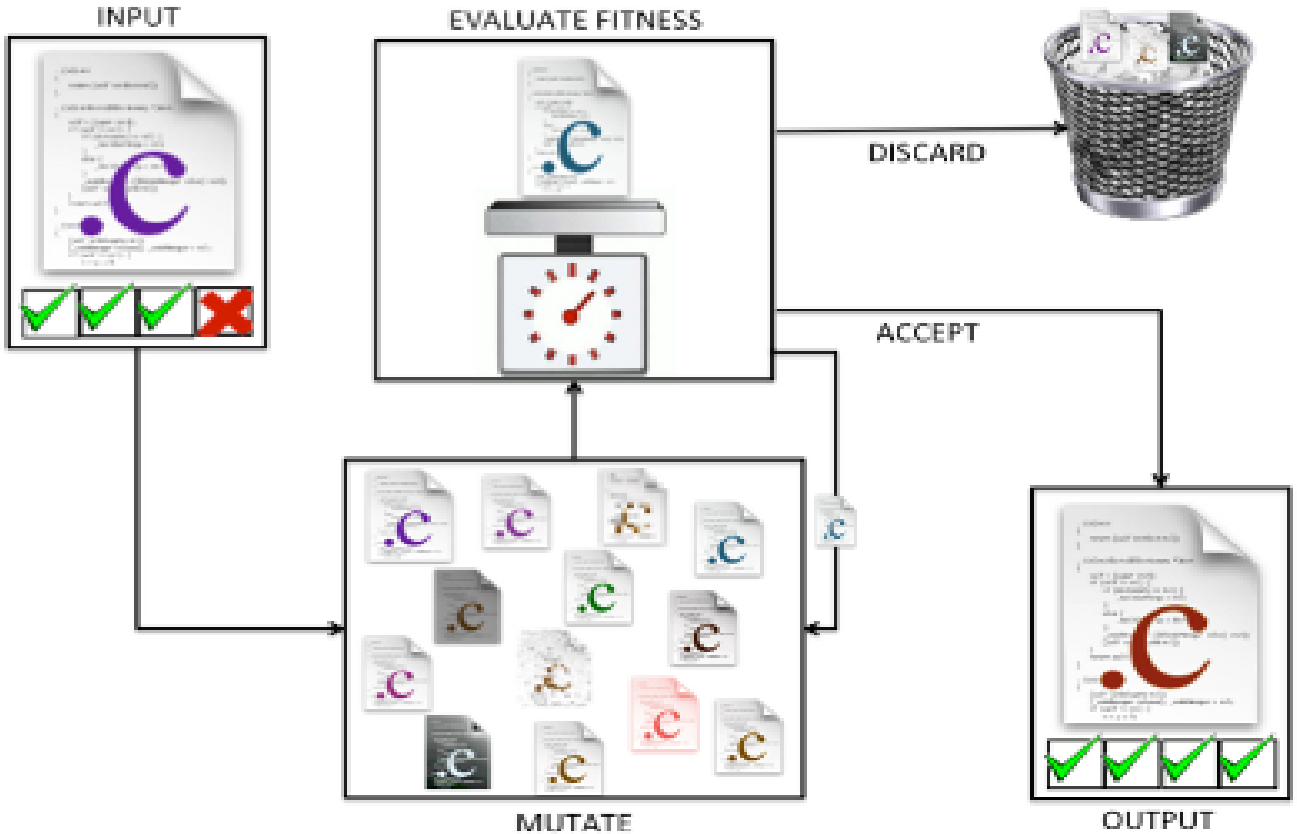
Example: maximize the number of ones in a bit string

1. Encoding of an individual: as a bit vector, e.g., $i = 1111010101$
2. Initial population: create n *random* invididuals
   $pop = \{i_1 = 1111010101, i_2 = 0111000101, i_3 = 0010001000, \ldots, i_n = 0010110111\}$
3. Fitness: the # of 1's in an individual
   $f(i_1) = 7, f(i_2) = 5, f(i_3) = 2, \ldots, f(i_n) = 6$
4. Selection: choose n individuals based on their fitnesses
   $pop' = \{i_1, i_n, i_1, i_2, \ldots\}$
5. Reproduction
   - Select each pair $m, n \in pop'$ with probability p_{xover}
   - Crossover m and n at a randomly selected point, i.e., 1-point xover
     1111010101 and 0010110111 give 1110110111 and 0011010101

Example: maximize the number of ones in a bit string

1. Encoding of an individual: as a bit vector, e.g., $i = 1111010101$
2. Initial population: create n *random* invididuals
   $pop = \{i_1 = 1111010101, i_2 = 0111000101, i_3 = 0010001000, \ldots, i_n = 0010110111\}$
3. Fitness: the # of 1's in an individual
   $f(i_1) = 7, f(i_2) = 5, f(i_3) = 2, \ldots, f(i_n) = 6$
4. Selection: choose n individuals based on their fitnesses
   $pop' = \{i_1, i_n, i_1, i_2, \ldots\}$
5. Reproduction
   - Select each pair $m, n \in pop'$ with probability p_{xover}
   - Crossover m and n at a randomly selected point, i.e., 1-point xover
     1111010101 and 0010110111 give 1110110111 and 0011010101
6. Mutation
   - select each individual i with probability p_{mut}
   - flip the bit at a randomly selected point
     $0010001000 \rightarrow 0011001000$

GenProg



- Given a program
  - C source code
- .. and evidence of a bug
  - test suite consisting of passed (positive) and failed (negative) test cases
- .. fix that bug using genetic algorithm
  - returns a textual patch

Fault Localization

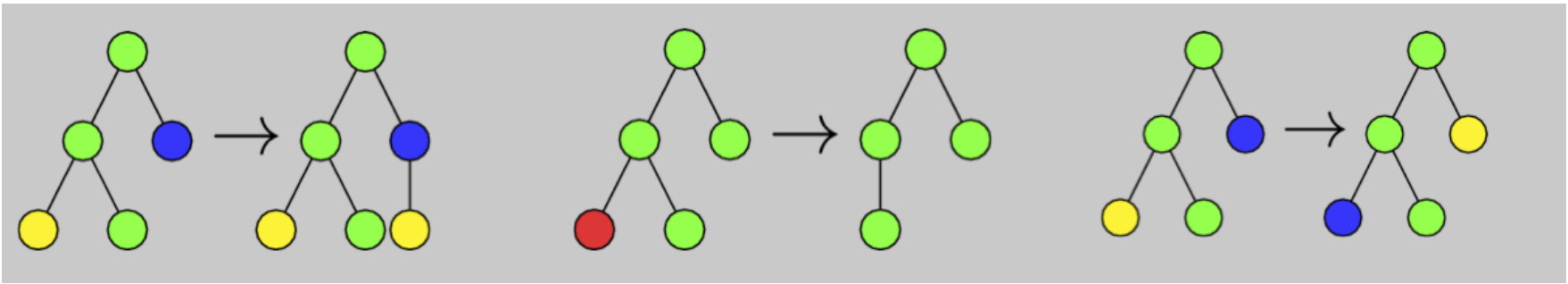
- In a large program, not every line is equally likely to contribute to the bug
- Insight: since we have the test cases, run them and collect coverage information
- The bug is more likely to be found on lines visited when running the failed test case.
- The bug is less likely to be found on lines visited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

Weighted Path

- Define a weighted path to be a list of $< statement, weight >$ pairs
- Statements in 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 (0.1) if S is also visited on a passed test

Genetic Representation and Operations

- Population: each invidiual is an *AST* of the program
- Mutation: select a statement $S \in AST$ biased by the weight of S
  - Insert $S1$ after S, delete S, replace S with $S2$
  - Choose $S1$ and $S2$ from the entire AST



insert/copy delete replace/swap

- Crossover: 1-point crossover
- Insight
  - don't invent new code
  - assume program contains the seeds of its own repair (e.g., has another null check elsewhere)

Fitness Evaluation

- Take in a program source P to be evaluated
- Compile P to an executable program P'
  - If cannot compile, assign fitness 0
- Fitness score of P' : weighted sum of test cases that P' passes
  - $f(P') = \#\ pos\ pass * W_{pos} + \#\ neg\ pass * W_{neg}$
  - $W_{pos} = 1$ and $W_{neg} = 10$
- If P' passes all test cases, then P is a solution candidate
- Note: the original (buggy) program passes all positive test cases

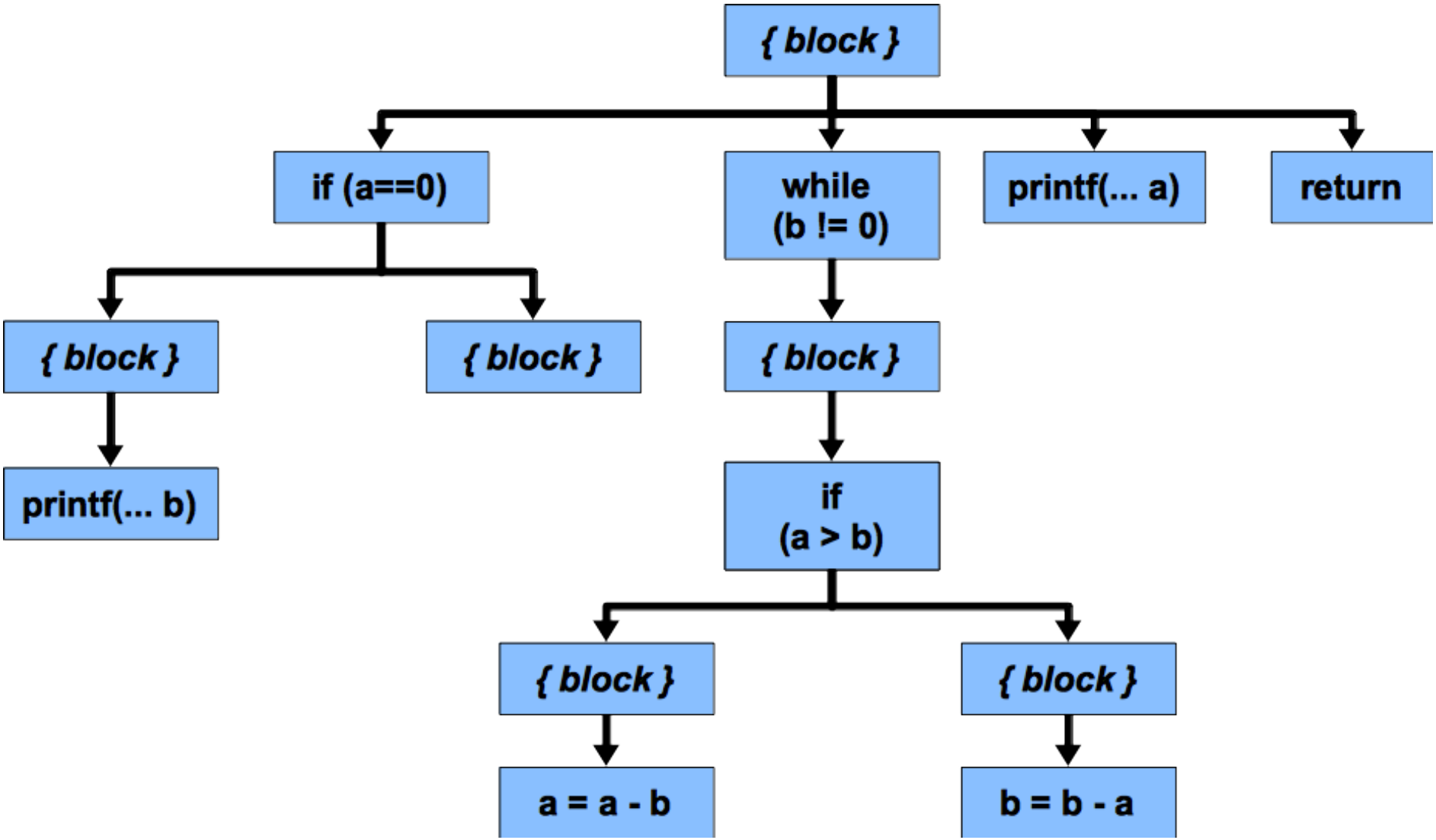
Example: GCD

```
/* 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);
  exit(0);
}
```

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

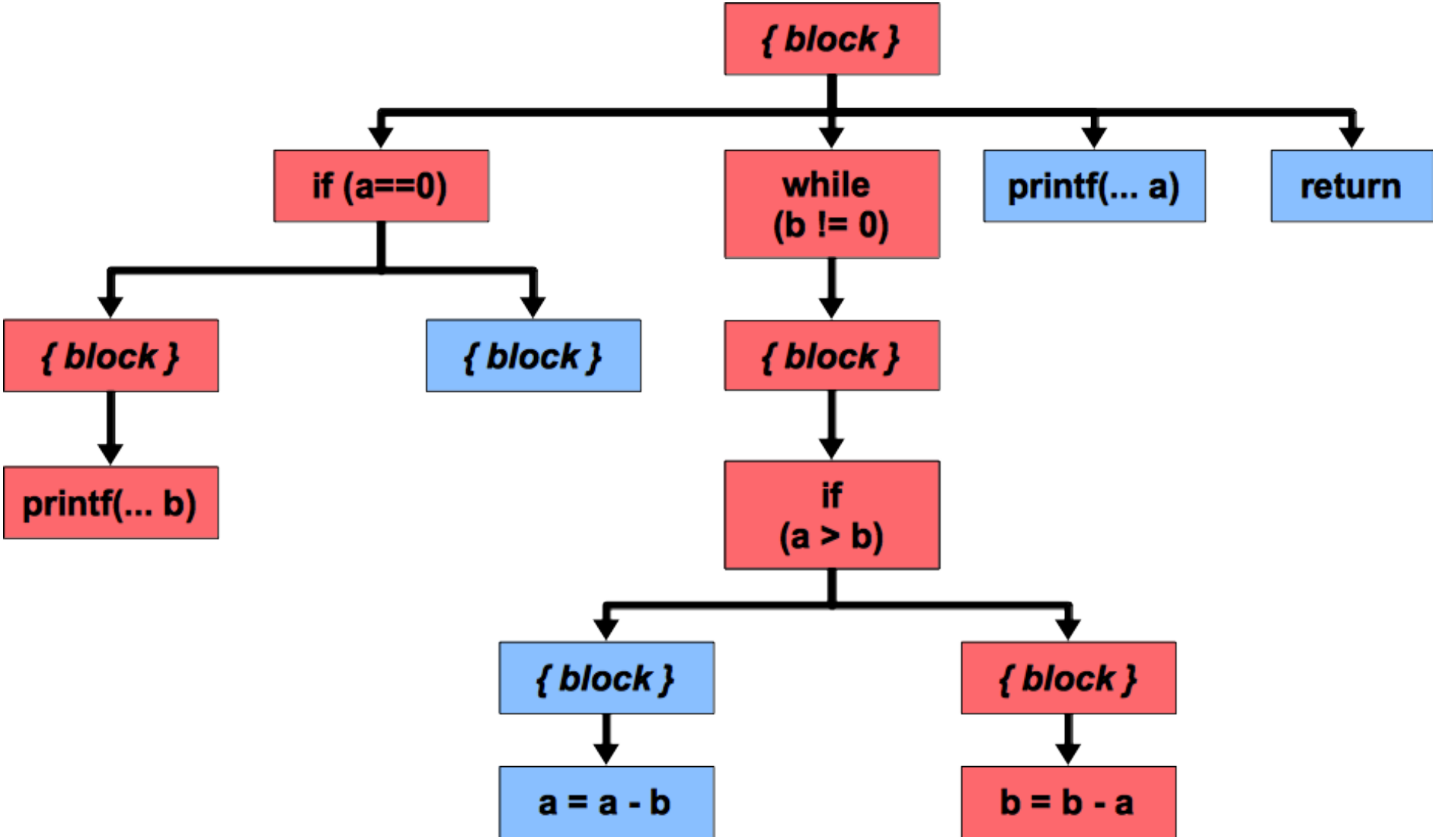
e.g., `a=0,b=55`

Abstract Syntax Tree



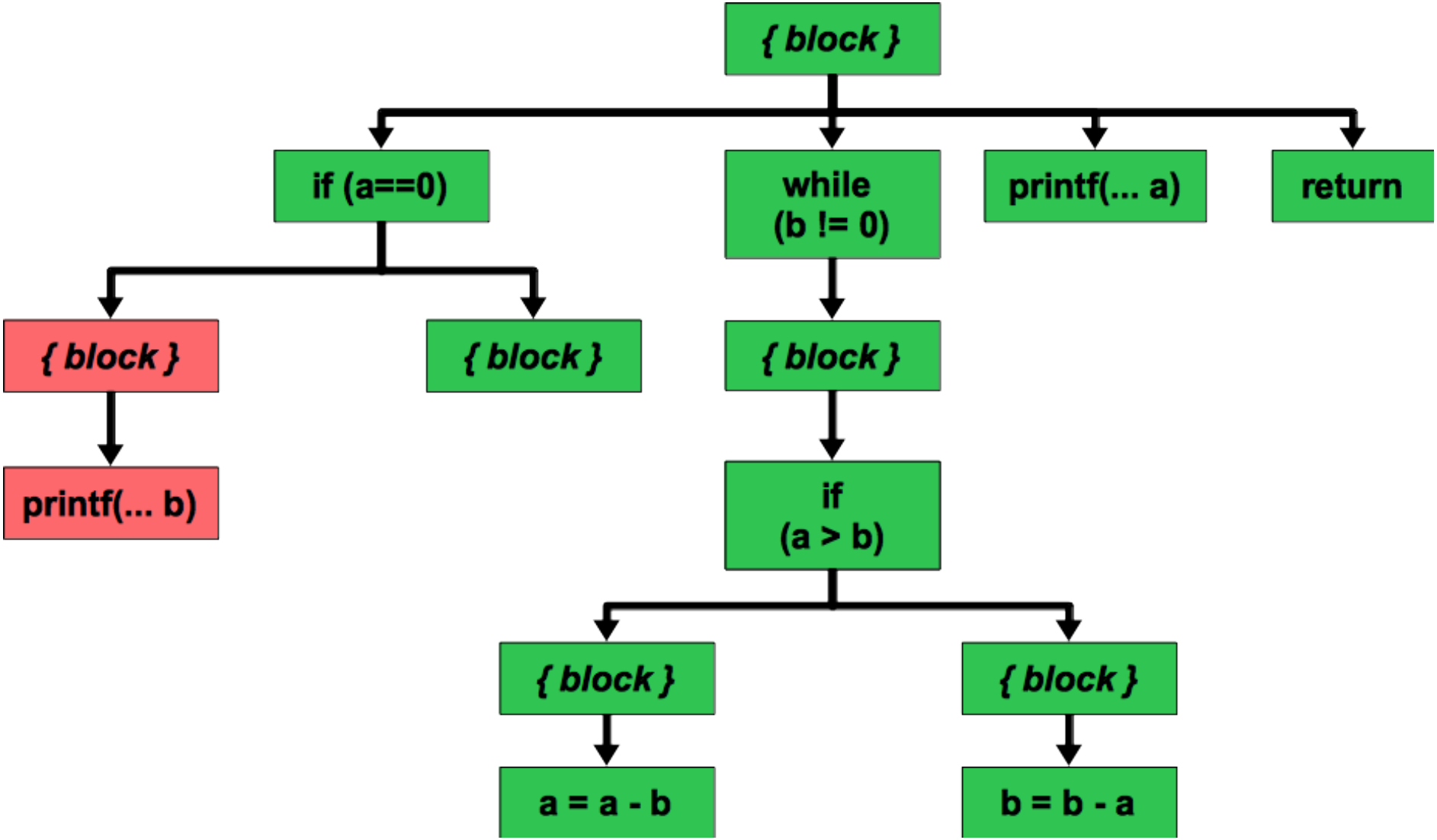
AST of `print_gcd`

Weighted Path: Negative Test case



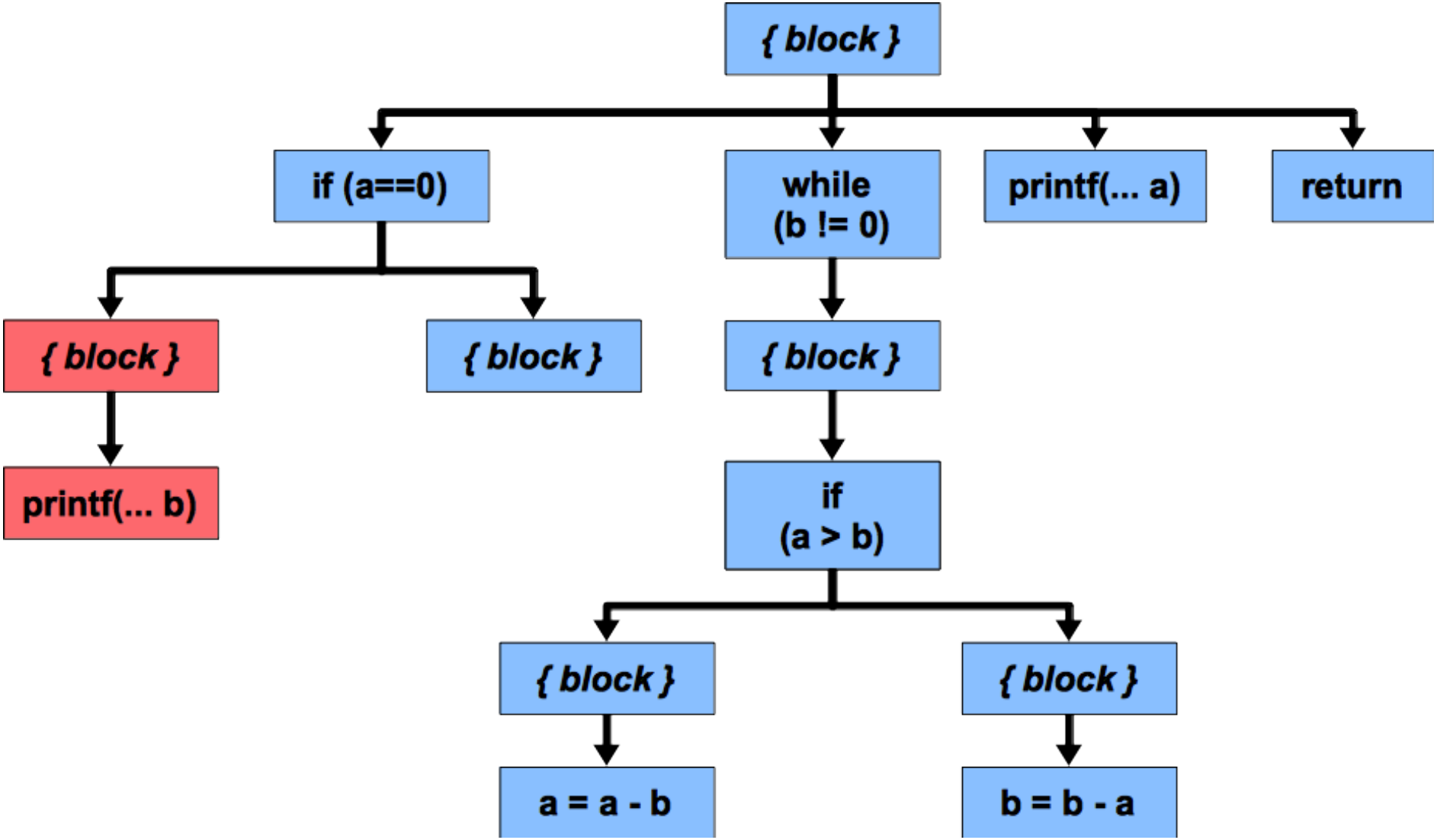
Nodes (stmts) visited on a negative test case, e.g., `a=0, b=55`

Weighted Path: Positive Testcase



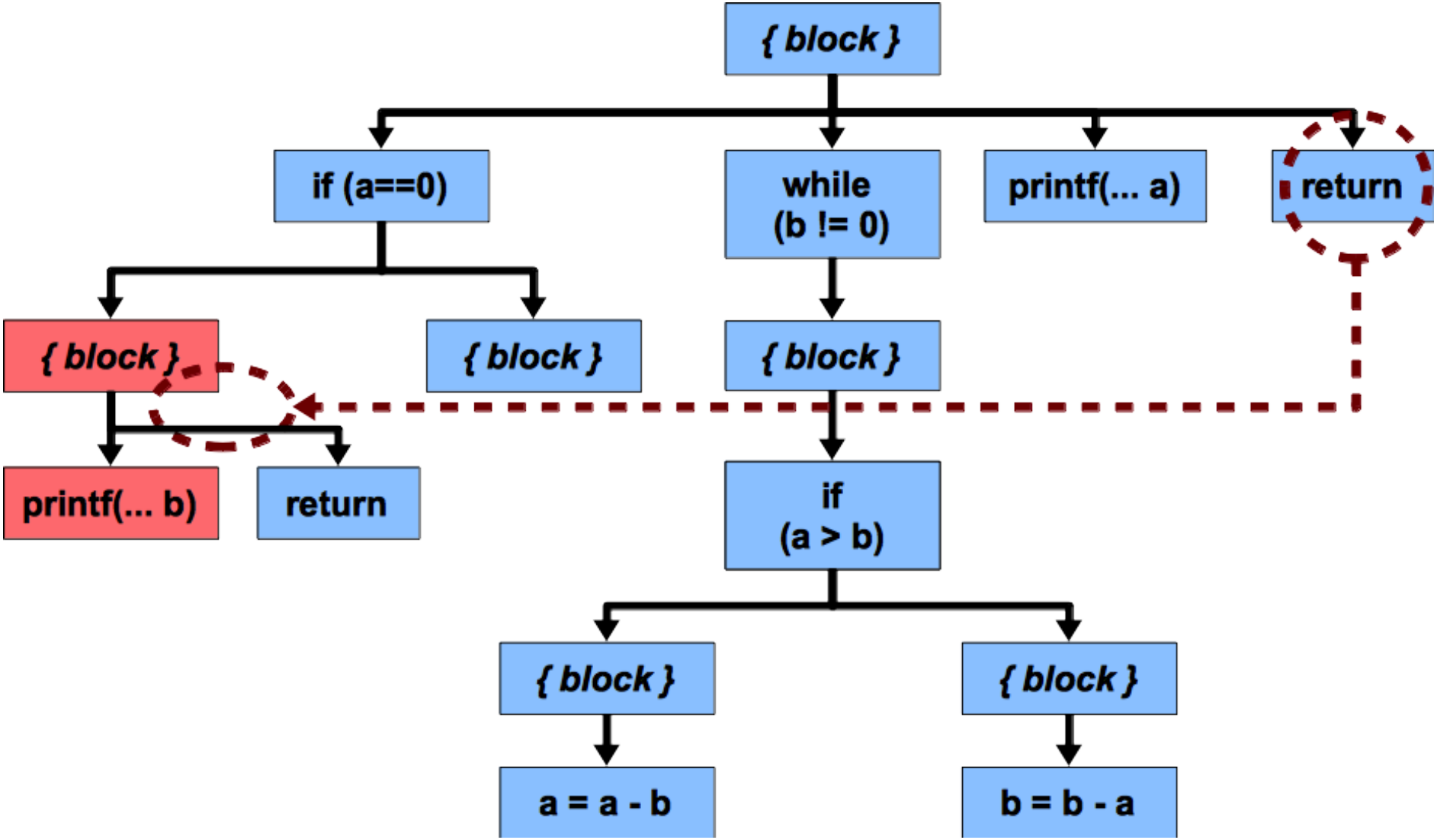
Nodes visited on a positive test case (`a=5,b=15`)

Weighted Path



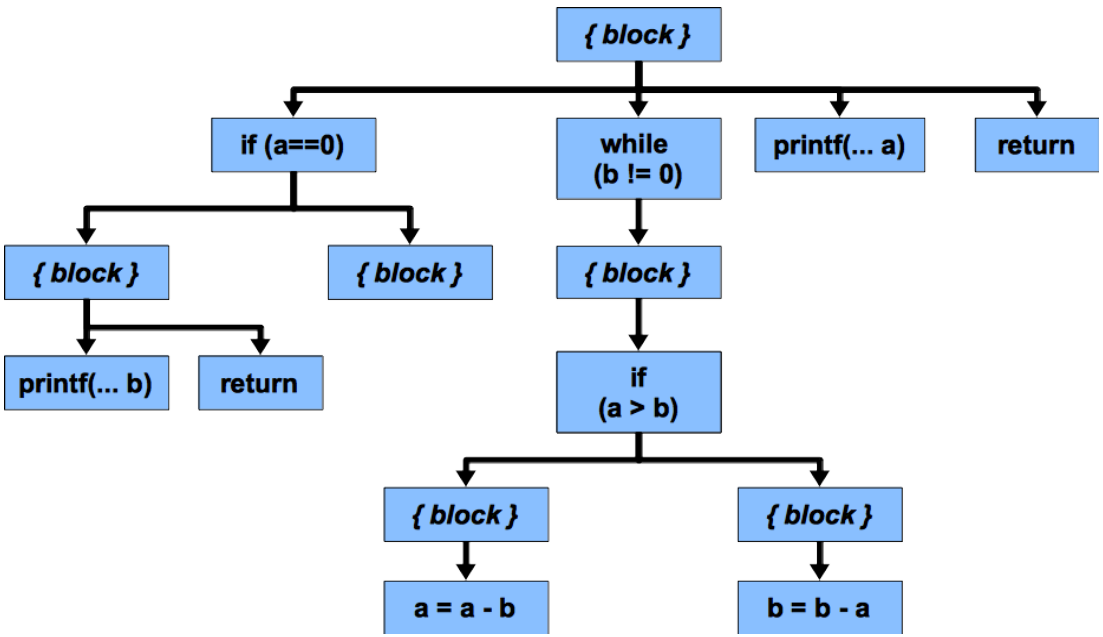
Concentreate on nodes visisted on negative, but not, positive test cases

Insertion



Randomly pick a node and insert after another node

Final Repair

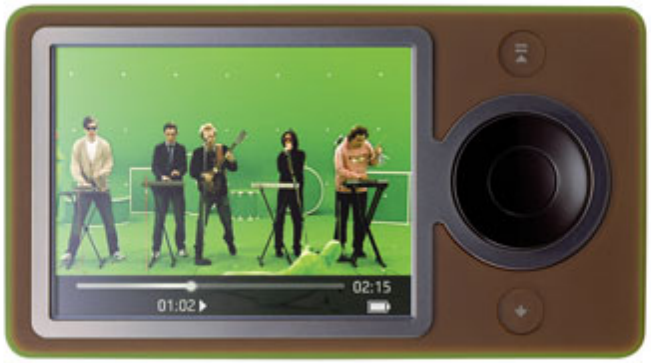



```
void print_gcd(int a, int b)
  if (a == 0) {
    printf("%d", b);
    return; //repair insert
  }
  while (b != 0)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  printf("%d", a);
  return;
}
```

Minimize 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.

Example: Zunebug



- Dec. 31, 2008. Microsoft Zune players freeze up
- Bug: infinite loop when input is last day of a leap year
- Negative test case: 10593 (Dec 31, 2008)
- Repair is not trivial

```
int zunebug(int days){
  int year = 1980;
  while (days > 365){
    if (isLeapYear(year)){
      if (days > 366){
        days -= 366;
        year += 1;
      }
    }
    else {
      days -= 365;
      year += 1;
    }
  }
  return year;
}
```

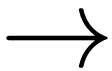
Evolving the Repair

```
if (days > 366)   {
  days -= 366;
  if (days > 366){   // insert #1
    days -= 366;     // insert #1
    year += 1;       // insert #1
  }                  // insert #1
  year += 1;
}
days -= 366;         // insert #2
```

Pass negative test cases, but fail some positive test cases.

Final Repair

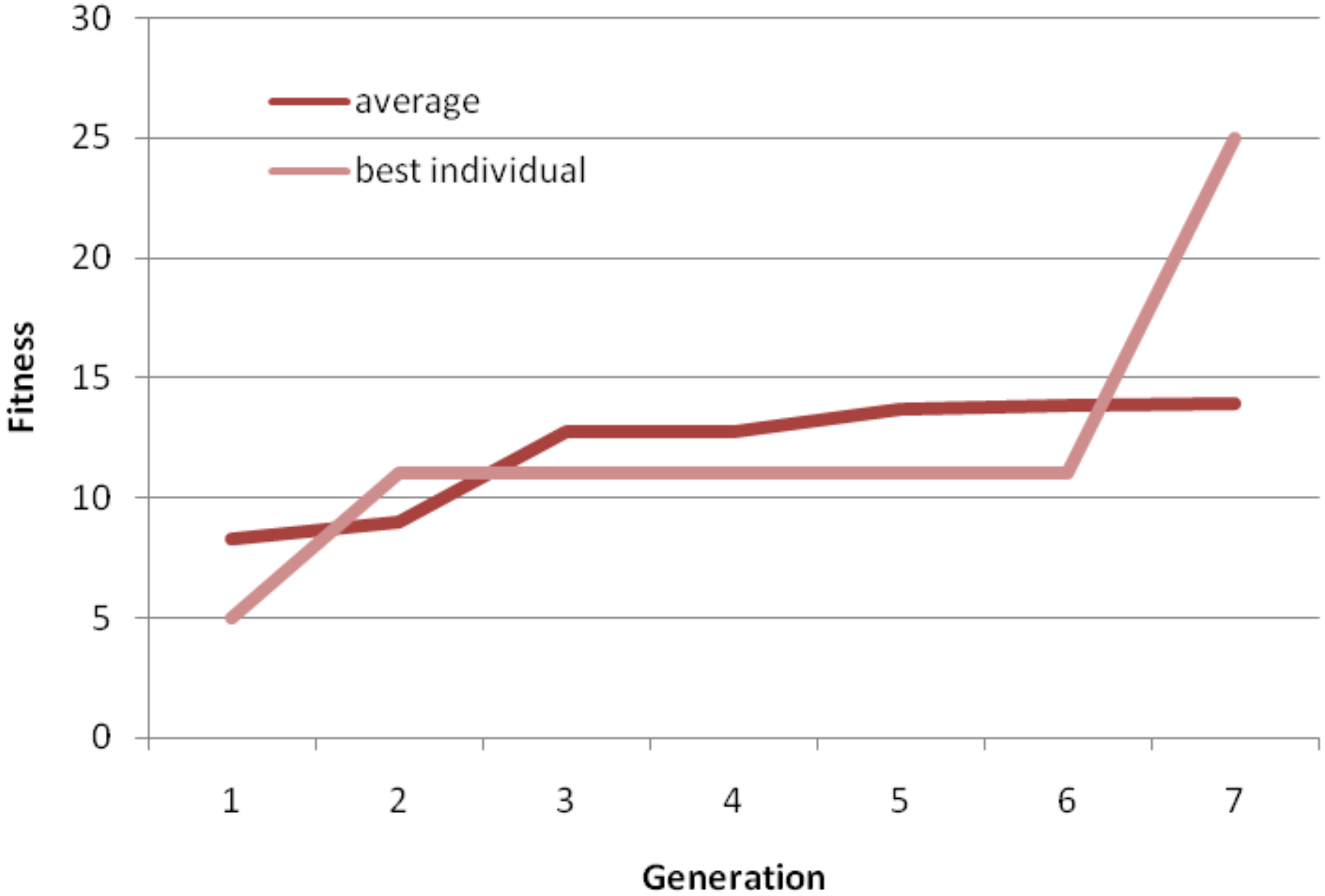
```
int zunebug(int days){
  int year = 1980;
  while (days > 365){
    if (isLeapYear(year)){
      if (days > 366){
        days -= 366;
        year += 1;
      }
    }
    else {
      days -= 365;
      year += 1;
    }
  }
  return year;
}
```



```
int zunebug_repair(int days)
  int year = 1980;
  while (days > 365){
    if (isLeapYear(year)){
      if (days > 366) {
        // repair deletes
        // days -= 366;
        year += 1;
      }
      // repair inserts
      days -= 366;
    } else {
      days -= 365;
      year += 1;
    }
  }
  return year;
}
```

- Final repair produced in 42 seconds
- One of the several possible repairs

Evolution of Repair



Experimental Results

Program	Description	LoC
gcd	example	22
uniq	duplicate text processing	1146
ultrix look	dictionary lookup	1169
svr4 look	dictionary lookup	1363
units	metric onversion	1504
deroff	document processing	2236
nullhttpd	web server	5575
indent	source code processing	9906
flex	lex analyzer generator	18775
atris	graphical Tetris game	21553
average		

Experimental Results

Program	Description	LoC	WPath	Time(s)	Success	Bug
gcd	example	22	1.3	149	54%	inf loop
uniq	duplicate text processing	1146	81.5	32	100%	segfault
ultrix look	dictionary lookup	1169	213.0	42	99%	segfault
svr4 look	dictionary lookup	1363	32.4	51	100%	inf loop
units	metric onversion	1504	2159.7	107	7%	segfault
deroff	document processing	2236	251.4	129	97%	segfault
nullhttpd	web server	5575	768.5	502	36%	buffer overrun
indent	source code processing	9906	1435.9	533	7%	inf loop
flex	lex analyzer generator	18775	3836.6	233	5%	segfault
atris	graphical Tetris game	21553	34.0	69	82%	buffer overrun
average			881.4	184.7	58.7%	

Experimental Results

Program	Description	LoC	WPath	Time(s)	Success	Bug
gcd	example	22	1.3	149	54%	inf loop
uniq	duplicate text processing	1146	81.5	32	100%	segfault
ultrix look	dictionary lookup	1169	213.0	42	99%	segfault
svr4 look	dictionary lookup	1363	32.4	51	100%	inf loop
units	metric onversion	1504	2159.7	107	7%	segfault
deroff	document processing	2236	251.4	129	97%	segfault
nullhttpd	web server	5575	768.5	502	36%	buffer overrun
indent	source code processing	9906	1435.9	533	7%	inf loop
flex	lex analyzer generator	18775	3836.6	233	5%	segfault
atris	graphical Tetris game	21553	34.0	69	82%	buffer overrun
average			881.4	184.7	58.7%	

- Initial result (ICSE '09): over 63,000 lines of code, 10 bugs

Experimental Results

Program	Description	LoC	WPath	Time(s)	Success	Bug
gcd	example	22	1.3	149	54%	inf loop
uniq	duplicate text processing	1146	81.5	32	100%	segfault
ultrix look	dictionary lookup	1169	213.0	42	99%	segfault
svr4 look	dictionary lookup	1363	32.4	51	100%	inf loop
units	metric onversion	1504	2159.7	107	7%	segfault
deroff	document processing	2236	251.4	129	97%	segfault
nullhttpd	web server	5575	768.5	502	36%	buffer overrun
indent	source code processing	9906	1435.9	533	7%	inf loop
flex	lex analyzer generator	18775	3836.6	233	5%	segfault
atris	graphical Tetris game	21553	34.0	69	82%	buffer overrun
average			881.4	184.7	58.7%	

- Initial result (ICSE '09): over 63,000 lines of code, 10 bugs
- Current result: over 5 Million LoC, 100+ bugs in php, python, gzip, etc

Why does it succeed ?

- Most bugs are small
- Only reuses existing statements
- Algorithmic innovations
  - Start with an (almost) working program
  - Weighted path greatly reduces search space
  - Minimization eliminates unnecessary fixes

Repair Quality

- Repairs are typically *not* what a human would have done
  - Example: GenProg 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 test cases
  - When POST test is omitted from `nullhttpd`, the generated repair eliminates POST functionality
  - Tests ensure GenProg do not sacrifice functionality
  - Minimization prevents gratuitous deletions
  - Adding more tests helps rather than hurting

Limitations

- May not handle nondeterministic faults
  - Difficult to test for race conditions, etc.
  - Long term: put scheduler constraints into the individual representation.
- Assumes bug test case visits different lines than normal test cases
- Assumes existing statements can form repair
  - Current work: repair templates
  - Hand-crafted and mined patterns and specifications from CVS repositories
- No formal correct guarantee (may delete functions not specified in test suites)

The growth of automatic program repair

- 2009 (a banner year): 15 papers on auto program repair
  - GenProg: evolves source code until it passes the rest of a test suite. [Weimer, Nguyen, Le Goues, Forrest]
  - ClearView: detects normal workload invariants and anomalies, deploying binary repairs to restore invariants. [Perkins, Kim, et al.]
  - Pachika: summarizes test executions to behavior models, generating fixes based on the differences. [Dallmeier et al.]
- 2012: 30 papers on auto program repair
  - At least 20+ different techniques, 3+ best paper awards, etc.
  - Two main approaches
    - uses random mutation to create *multiple* repair candidates and validate them
    - uses constraint solving to create a *single* repair that is correct-by-construction
- 2013: ICSE has a Program Repair session
- 2017: Program repair sessions/papers appear in many SE/PL conferences

Invariant Generation and Template-based Synthesis

Invariant Generation

```
def intdiv(x, y):
  q = 0
  r = x
  while r ≥ y:
    a = 1
    b = y
    while [??] r ≥ 2b:
      a = 2a
      b = 2b
    r = r - b
    q = q + a
    [??]
    return q
```

- Discovers invariant properties at certain program locations
- Answers the question *"what does this program do ?"*

Invariant Generation and Template-based Synthesis

Invariant Generation

```
def intdiv(x, y):
  q = 0
  r = x
  while r ≥ y:
    a = 1
    b = y
    while [??] r ≥ 2b:
      a = 2a
      b = 2b
    r = r - b
    q = q + a
    [??]
    return q
```

- Discovers invariant properties at certain program locations
- Answers the question *"what does this program do ?"*

Template-based Synthesis

```
def intdiv(x, y):
  q = 0
  r = x
  while r [**] y:
    a = 1
    b = [**]
    while r ≥ 2b:
      a = [**]
      b = 2b
    r = r - b
    q = q + a

  return [**]
```

- Creates code under specific templates from partially completed programs
- Can be used for automatic program repair

Summing Up

- Software **is** the problem, and industry is already paying untrusted strangers
- Automated Program Repair is a hot research area with rapid growth in the last few years
- GenProg: does not rely on formal specification, no pre-specification of bug type or repair template
- Challenges and Opportunities:
  - Invariant Generation
  - Program Synthesis