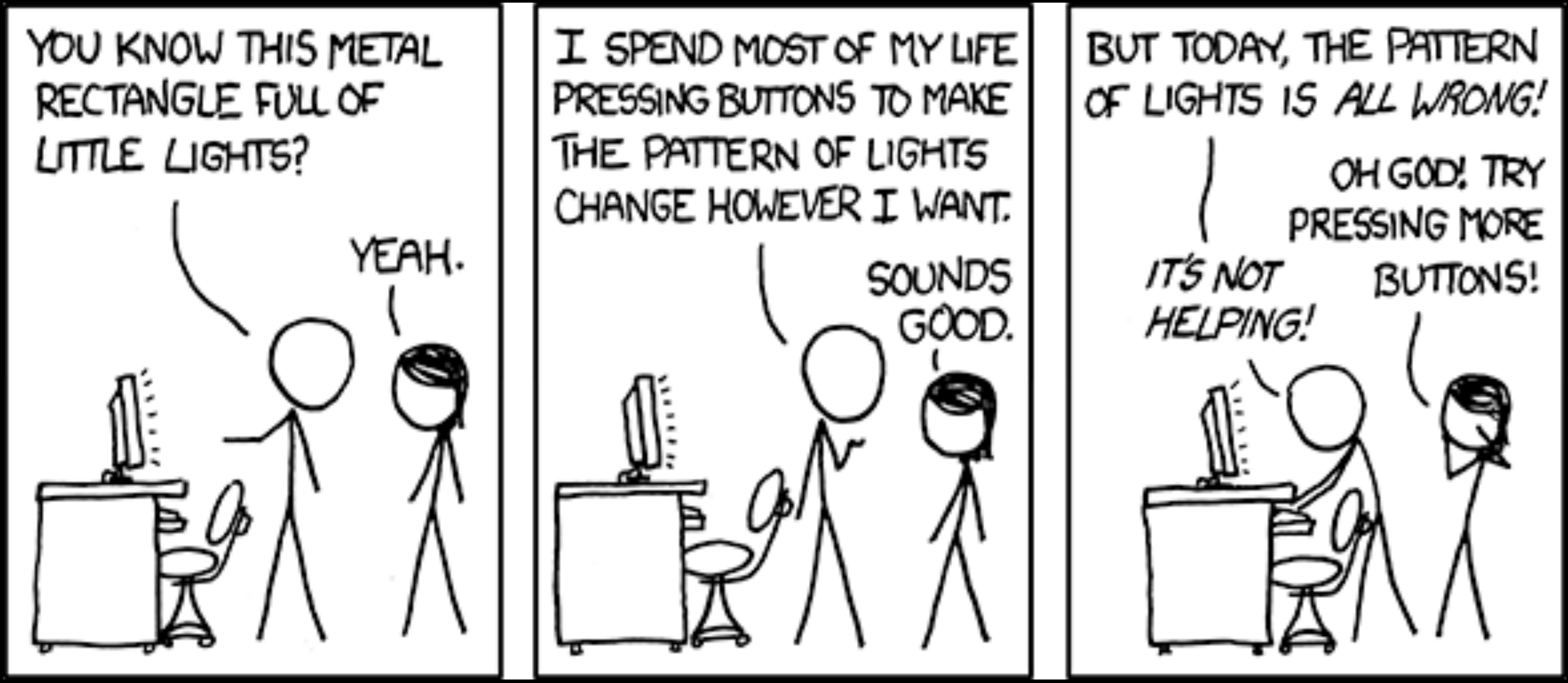
Mutational Robustness and Automatic Program Repair

Ethan Fast
SFI REU 2010

Mentor: Stephanie Forrest

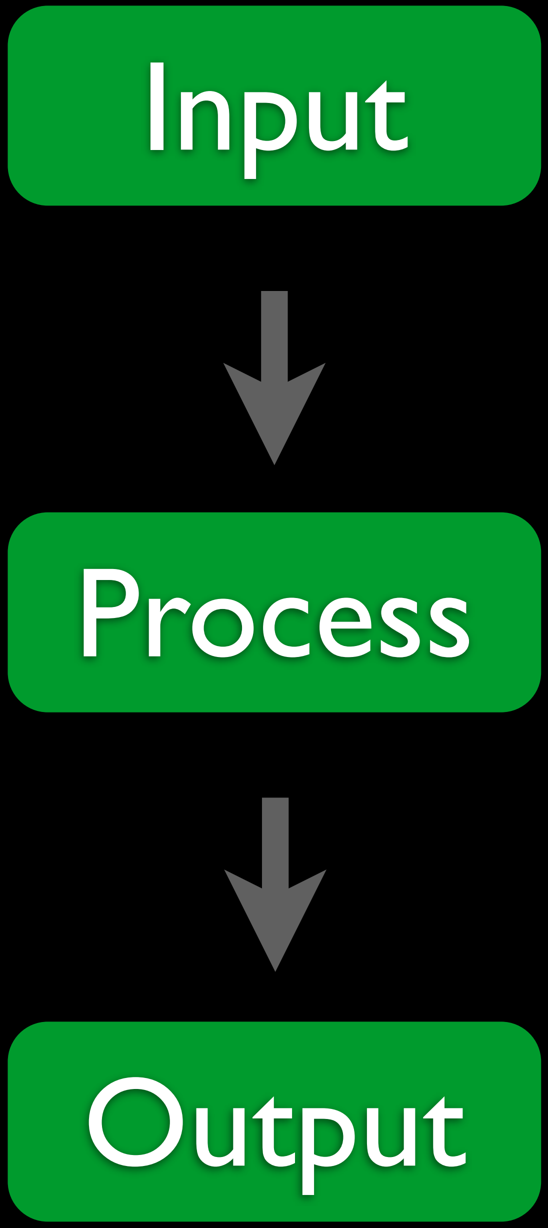

YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS *ALL WRONG!*
OH GOD! TRY PRESSING MORE BUTTONS!
IT'S NOT HELPING!

Automatic Program Repair via Genetic Programming

Weimer and Forrest

An optimization technique inspired by evolution

GP Program Repair



Input

program source code

regression tests

test case illustrating bug

Process

generate program variants

run them on test cases

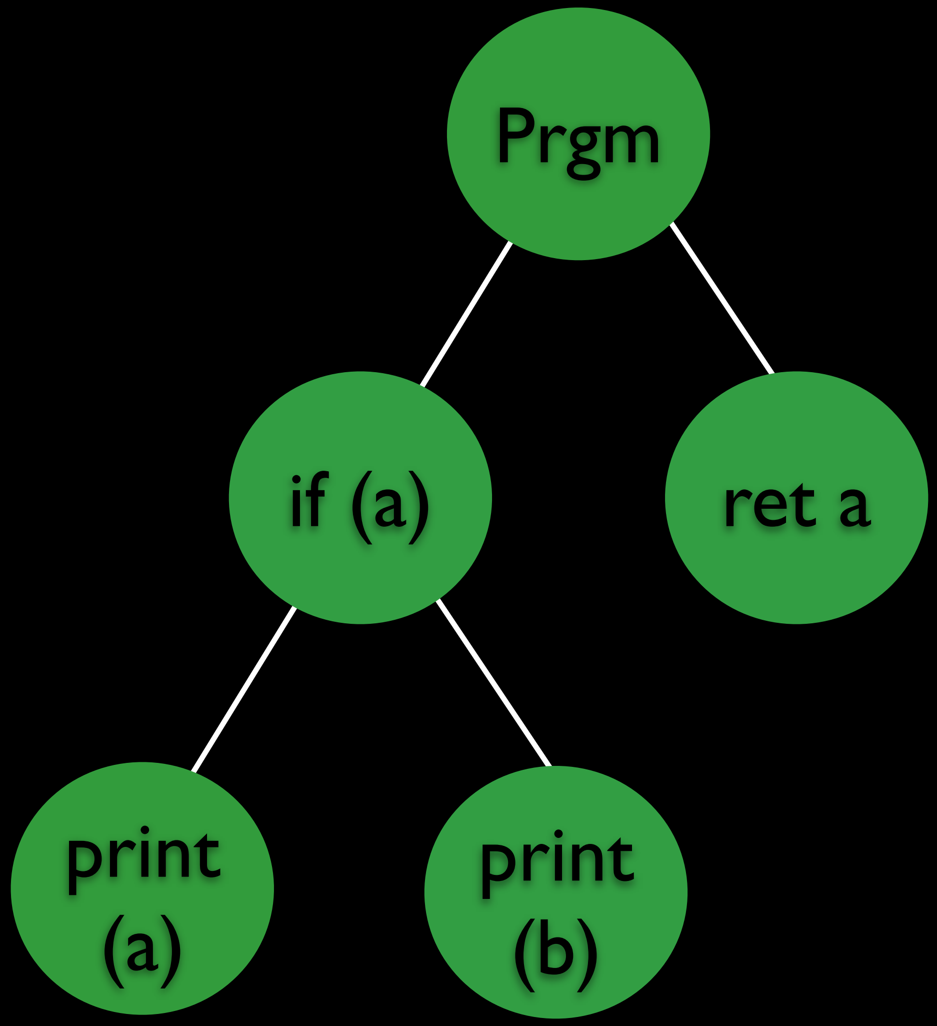
selection, crossover, mutation

Output

new program that passes tests

or, no solution

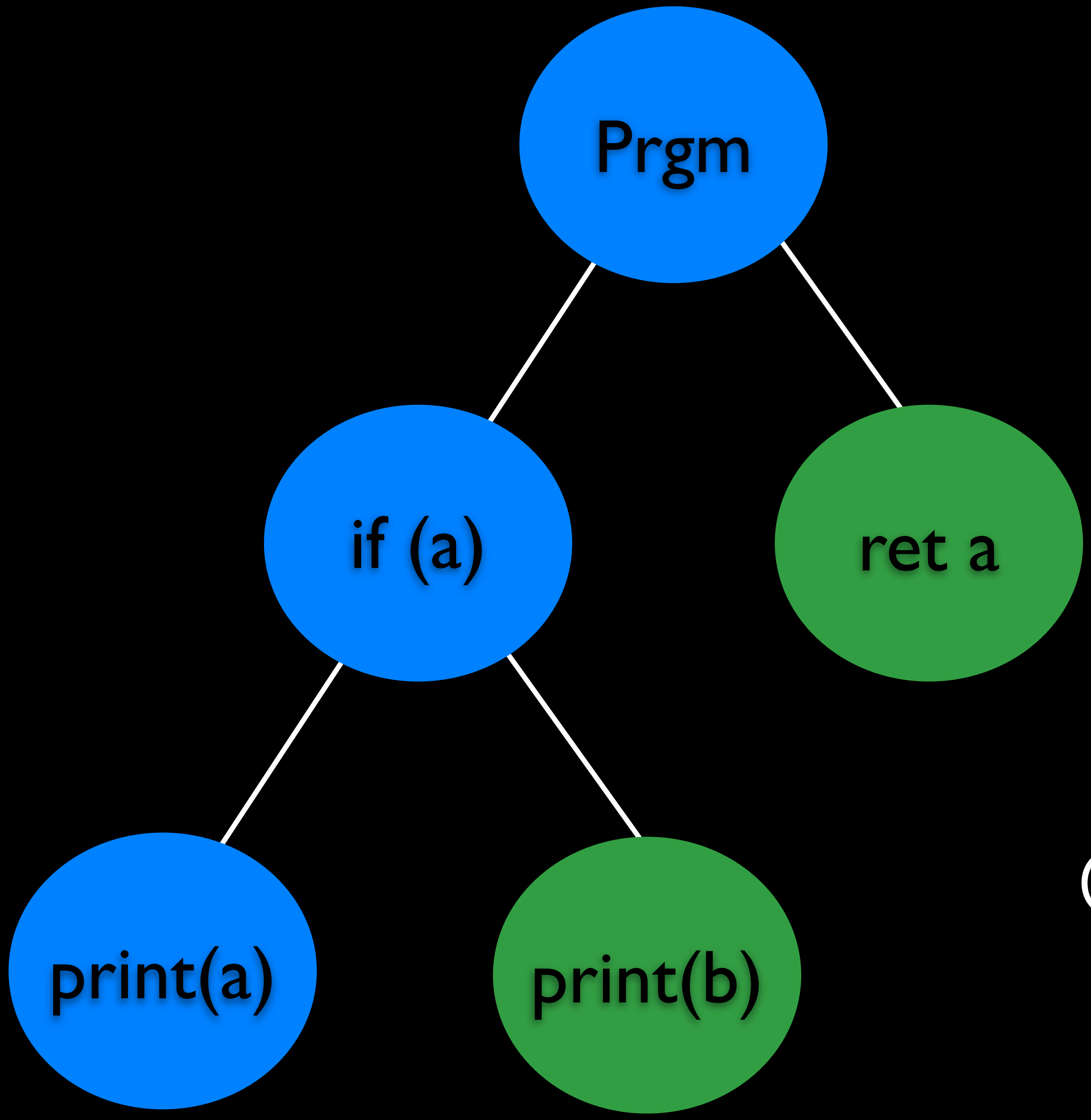
Representation



Individuals represented as ASTs

Weighted Path

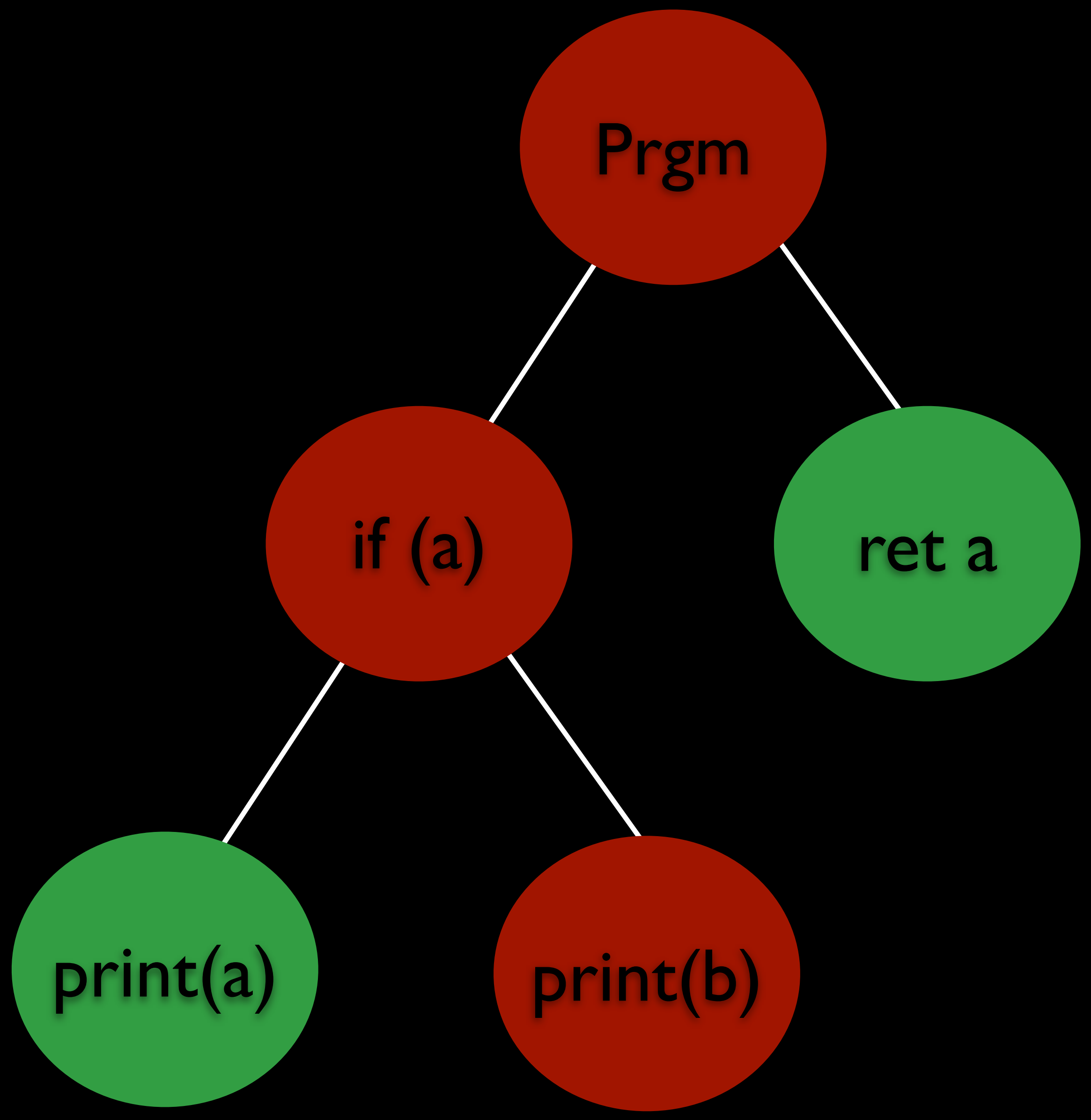
A means of fault localization



Good Path

Weighted Path

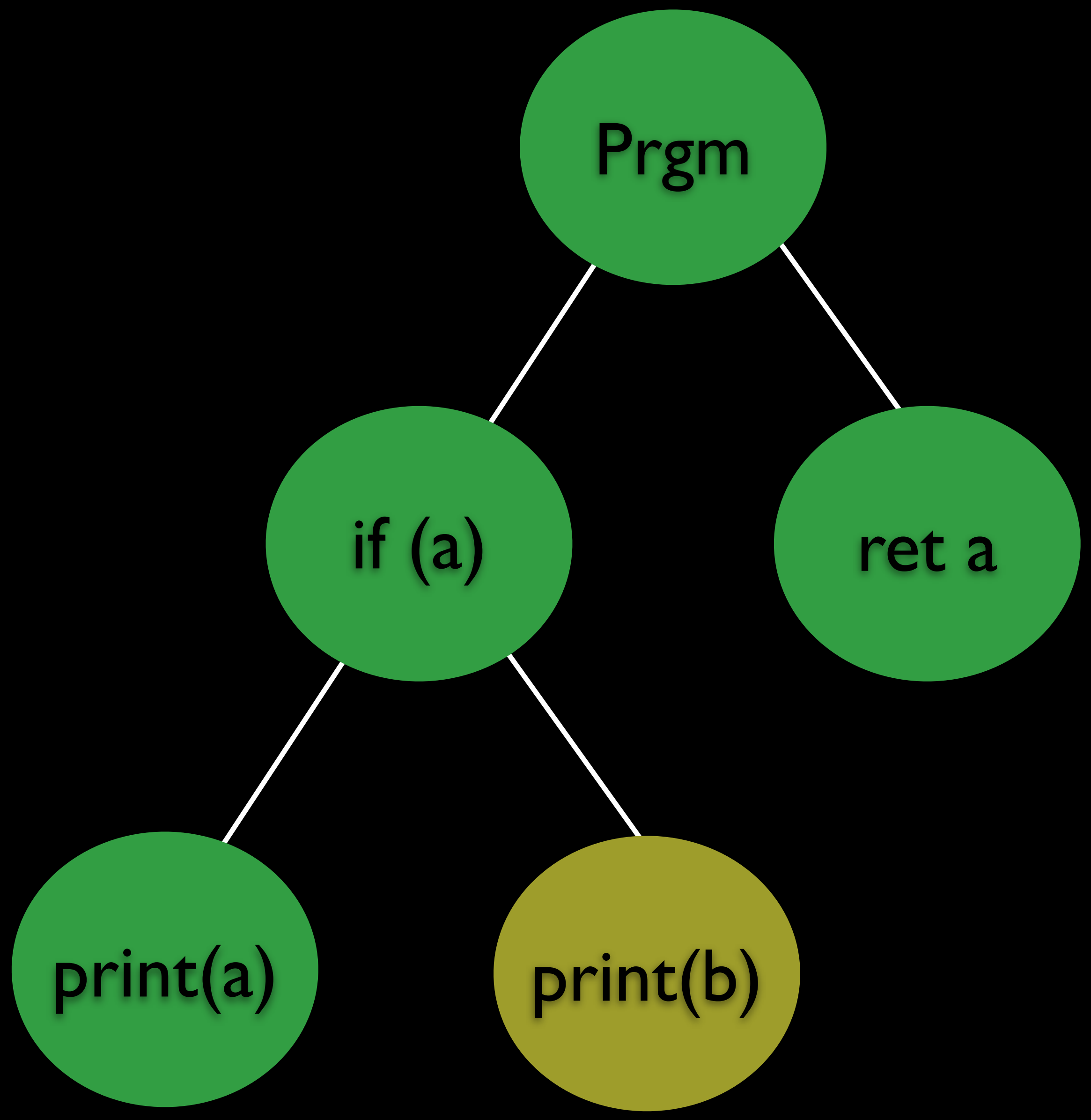
A means of fault localization



Bad Path

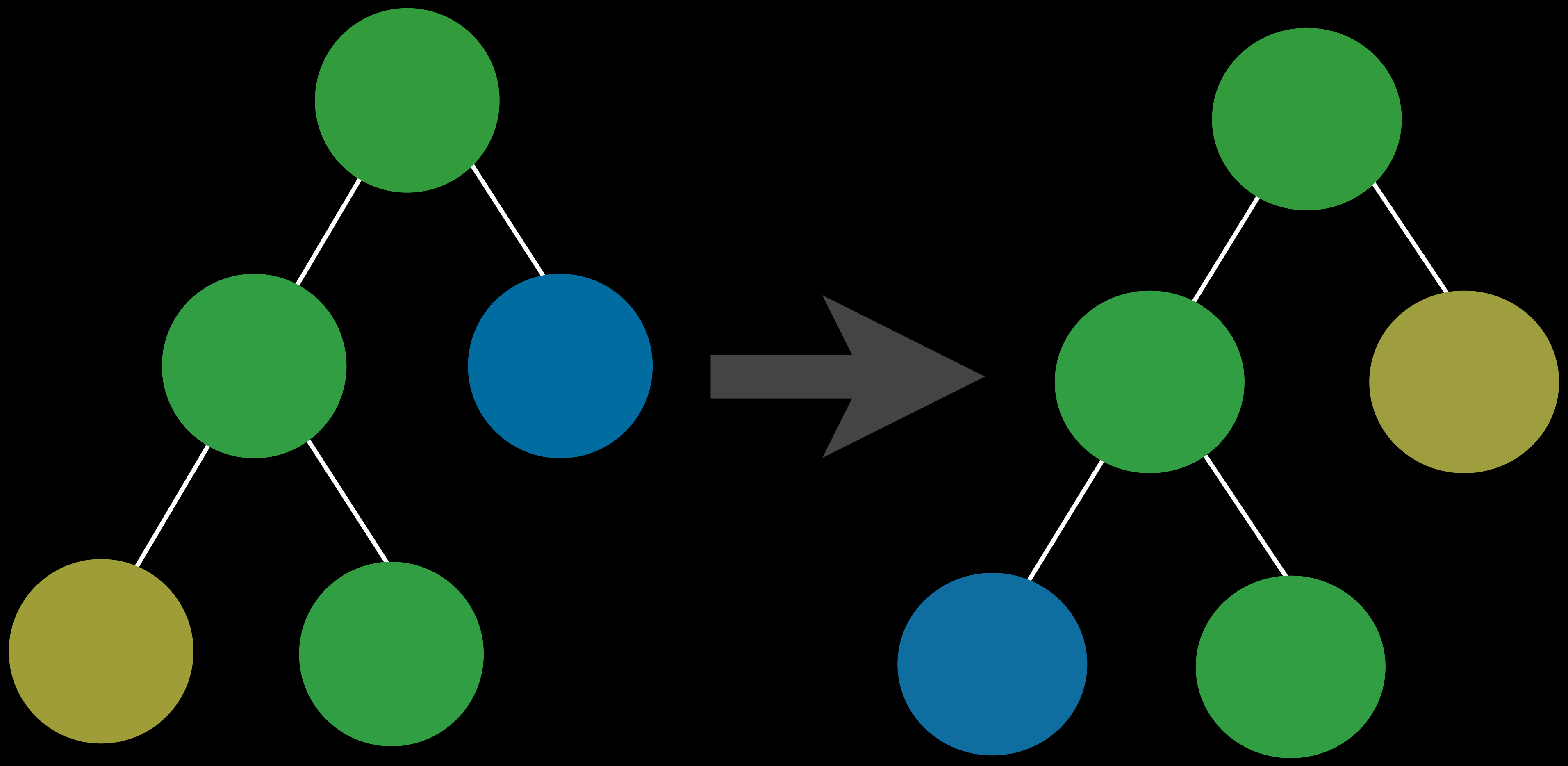
Weighted Path

A means of fault localization



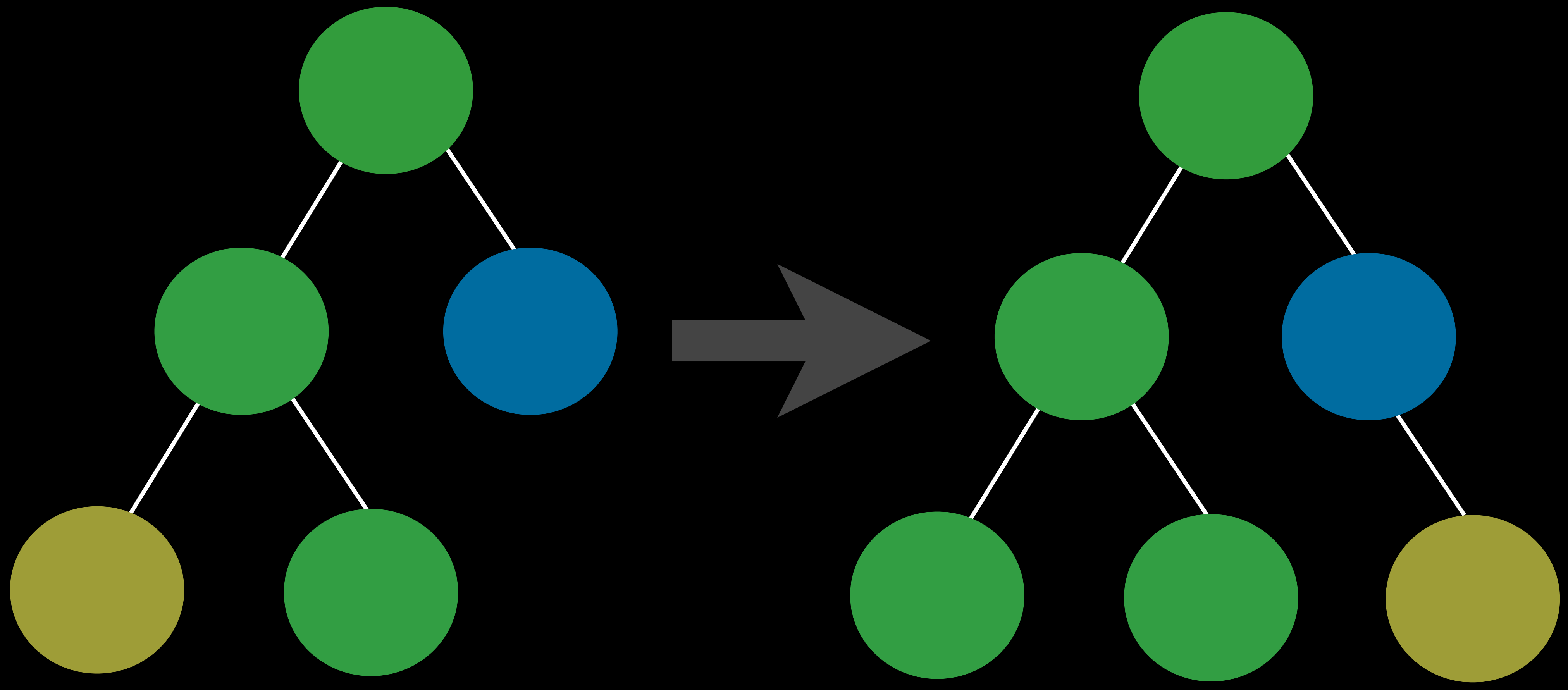
Final Path

Mutation: Swap



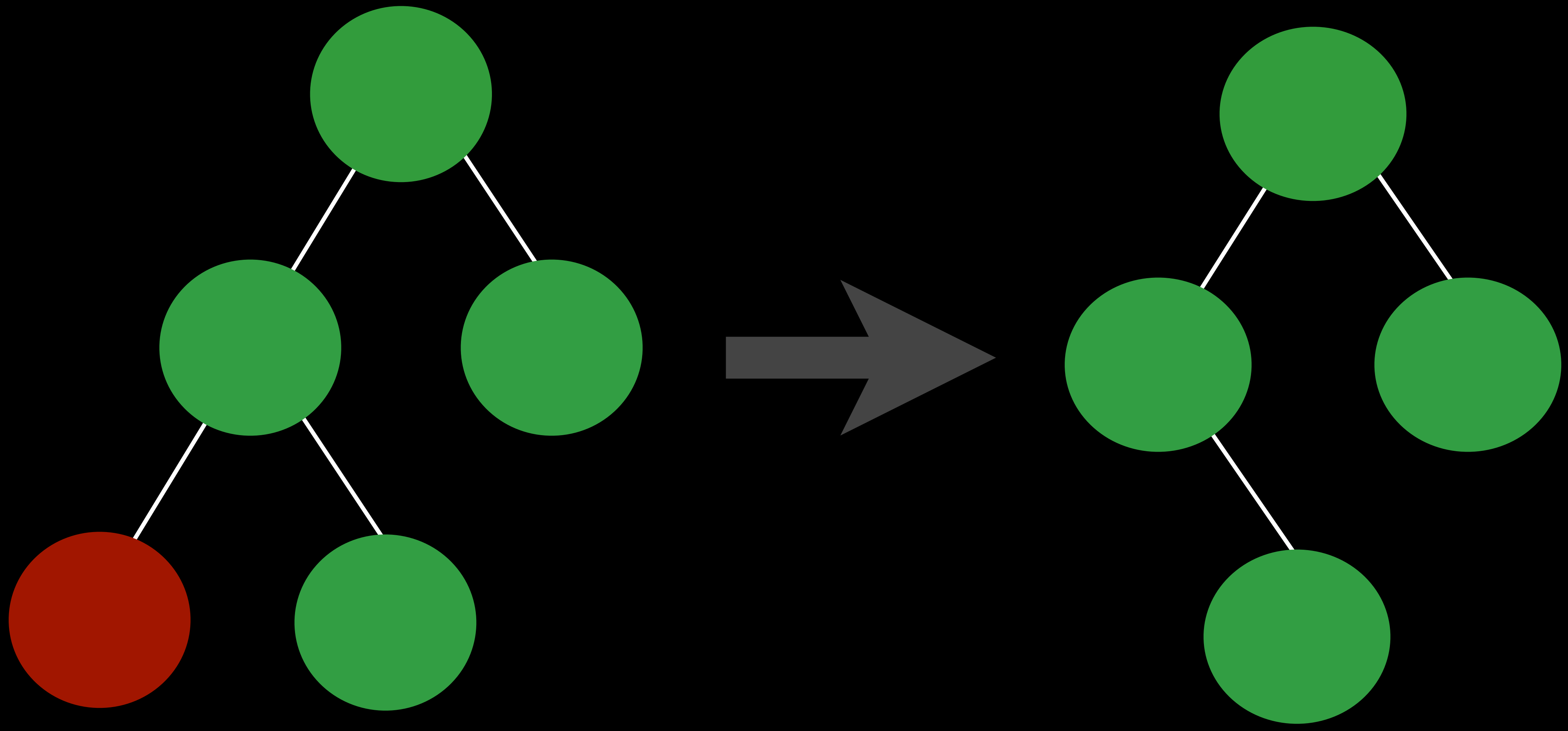
Exchange two nodes on the tree

Mutation: Append



Copy a node to elsewhere on the tree

Mutation: Delete



Delete a node from the tree

GP Program Repair Details

To compute fitness, compile a variant

If it fails to compile, then fitness = 0

Otherwise, run test cases

Now, fitness = # tests passed

Negative test case(s) more heavily weighted

Does it actually work?

deroff	gcd	look
indent	uniq	zune
atris	leukocyte	imagemagick
tiff	nullhttpd	python
php	lighttpd	openldap

A few repaired programs

So what about robustness?

Some Definitions

mutational robustness: the probability of a change in genotype affecting a change in phenotype

neutral fitness landscape: described by region of differing genotypes assigned the same fitness value

Motivation

High mutational robustness seems to support the idea of evolving software

Robustness and neutral fitness may be key ideas for repairing more complicated bugs

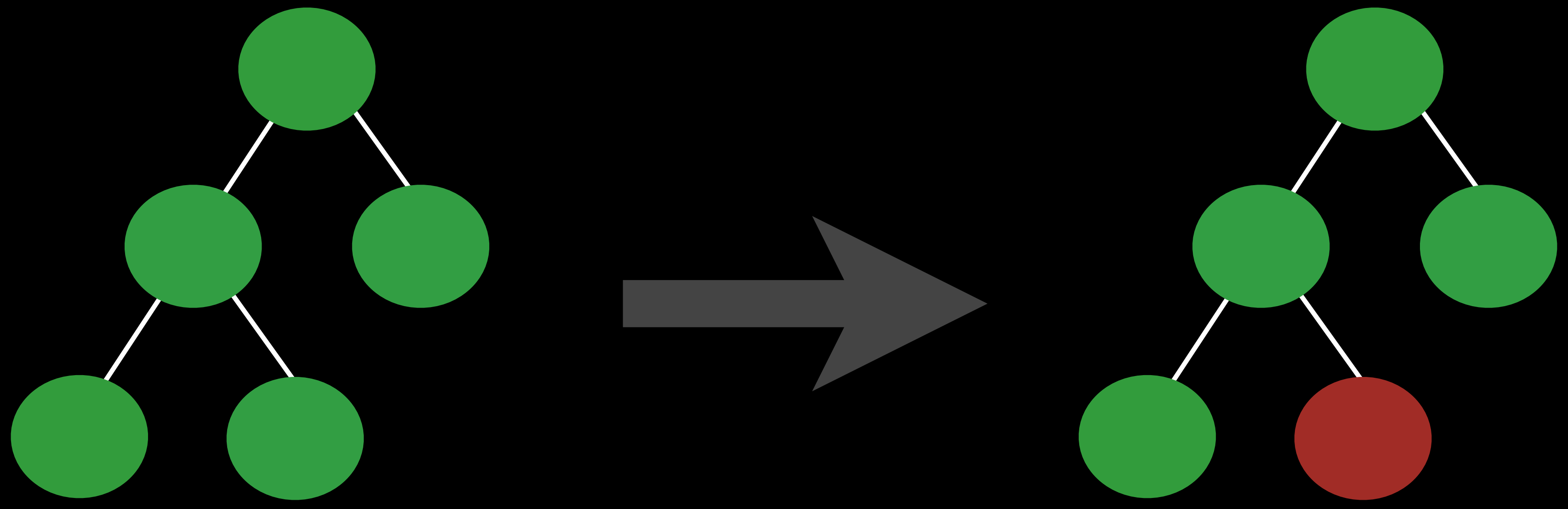
Questions

How do we measure robustness?

Given a metric, how mutationally robust are typical programs?

How does robustness affect automatic program repair?

Measuring Robustness



Original Program

Apply Mutation (x1000)

Metrics:

Average distance in fitness

Percent of mutations that are neutral

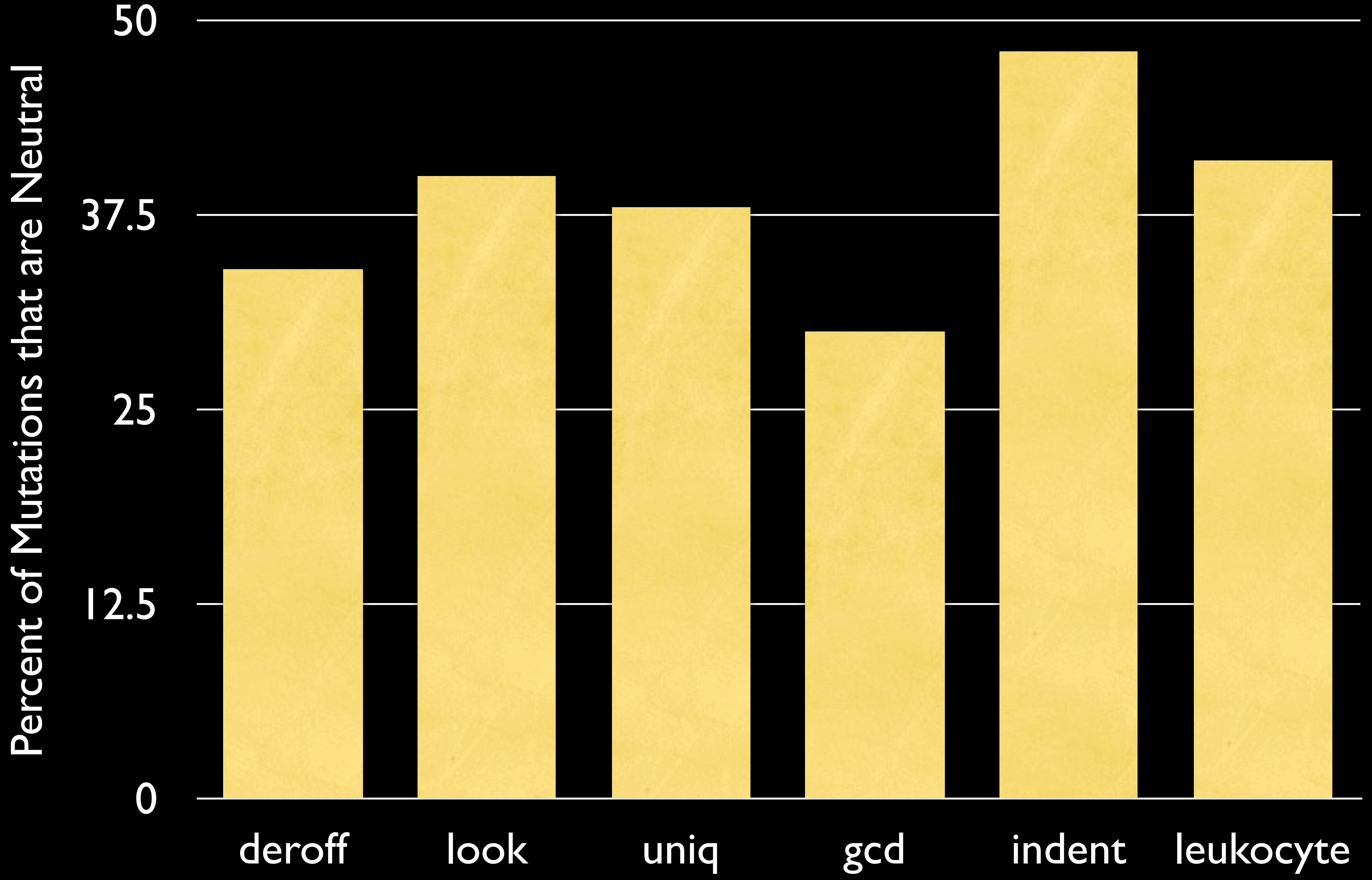
Your Intuition

(A walk down the garden path)

Suppose that we make a single mutation to some arbitrary program.

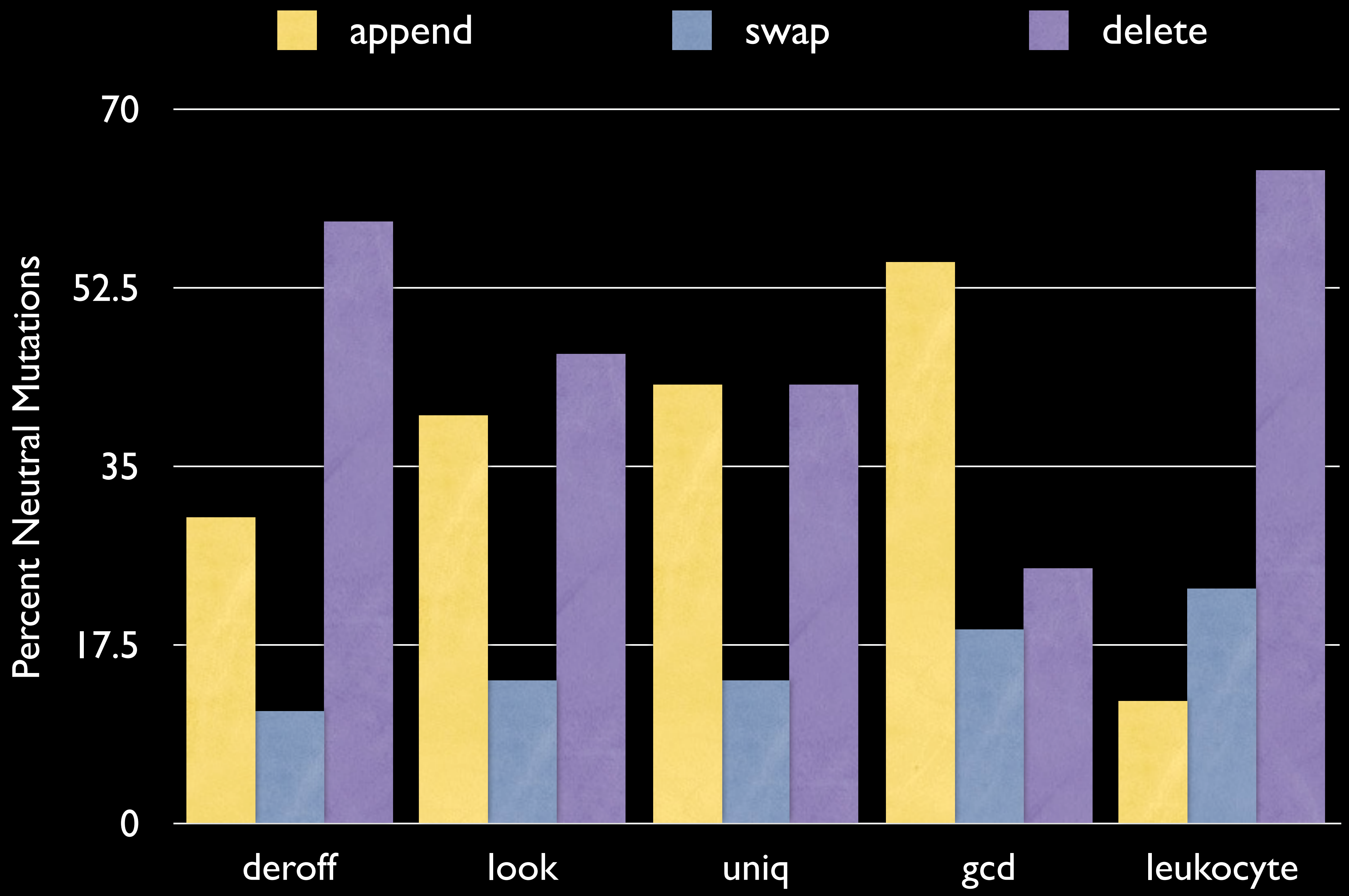
How often will its behavior change?

Neutral Mutations



What mutation operators are likely to result in neutral mutations?

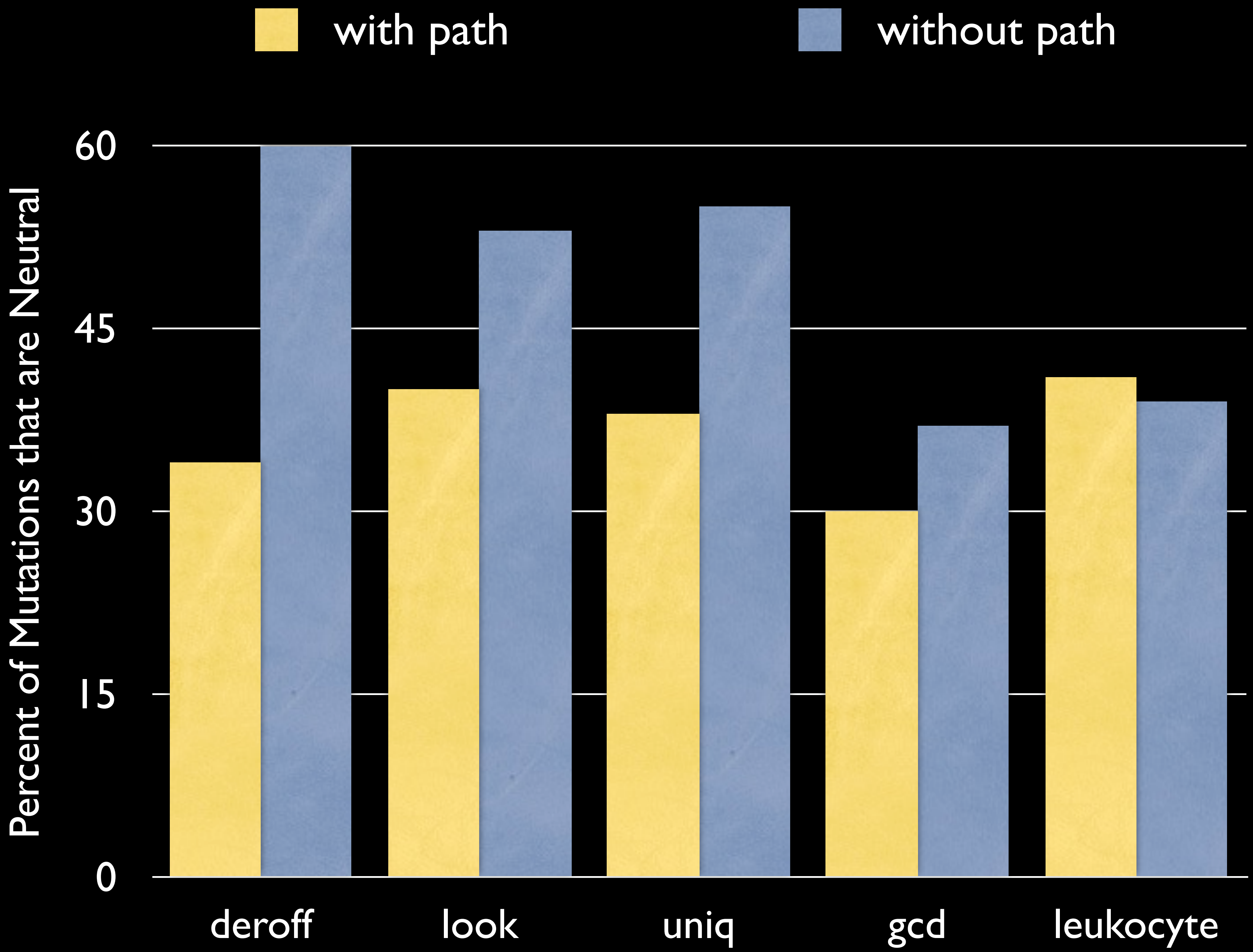
By Mutation Operators



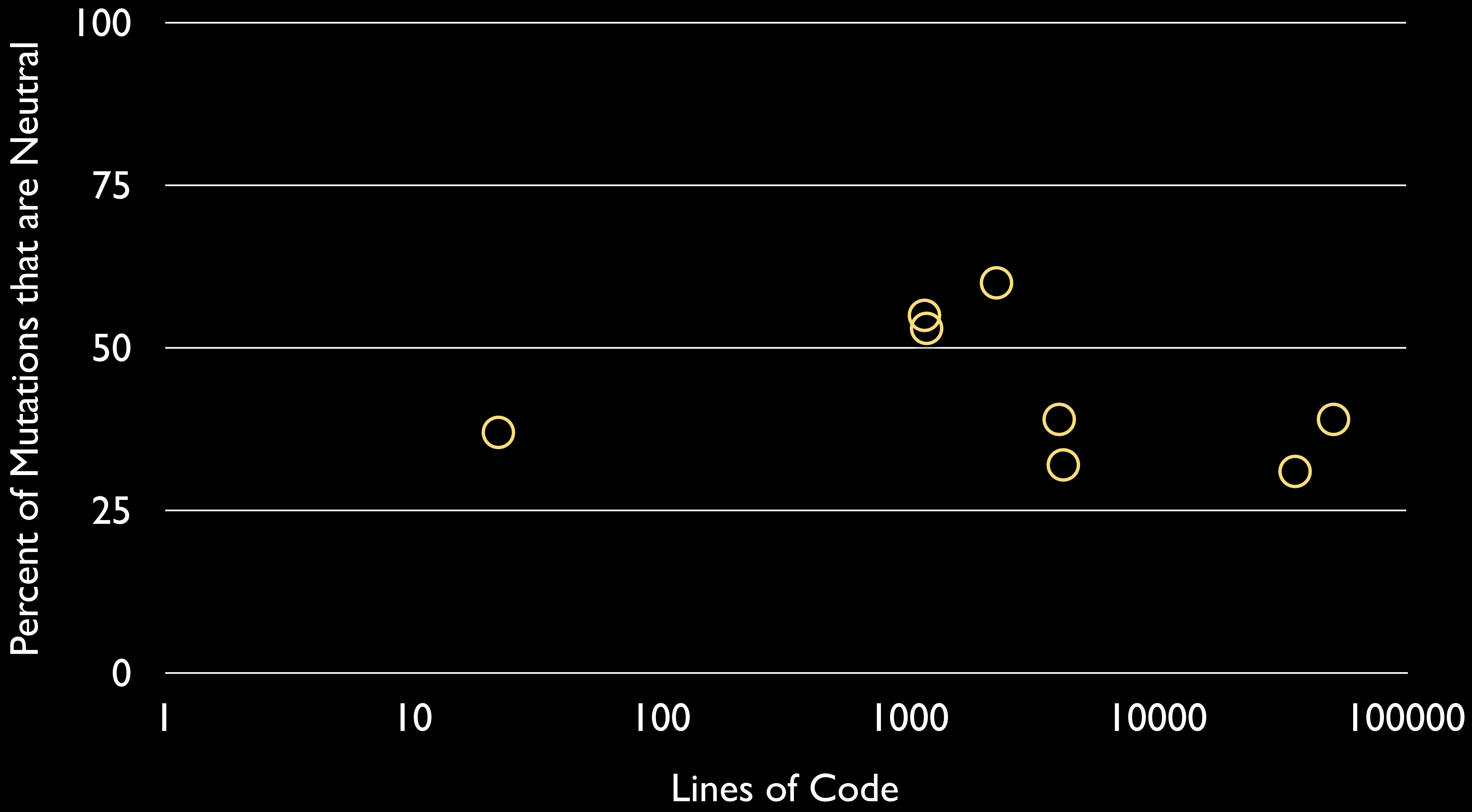
So what about that weighted path?

Shouldn’t one look at programs more generally?

With and Without Path



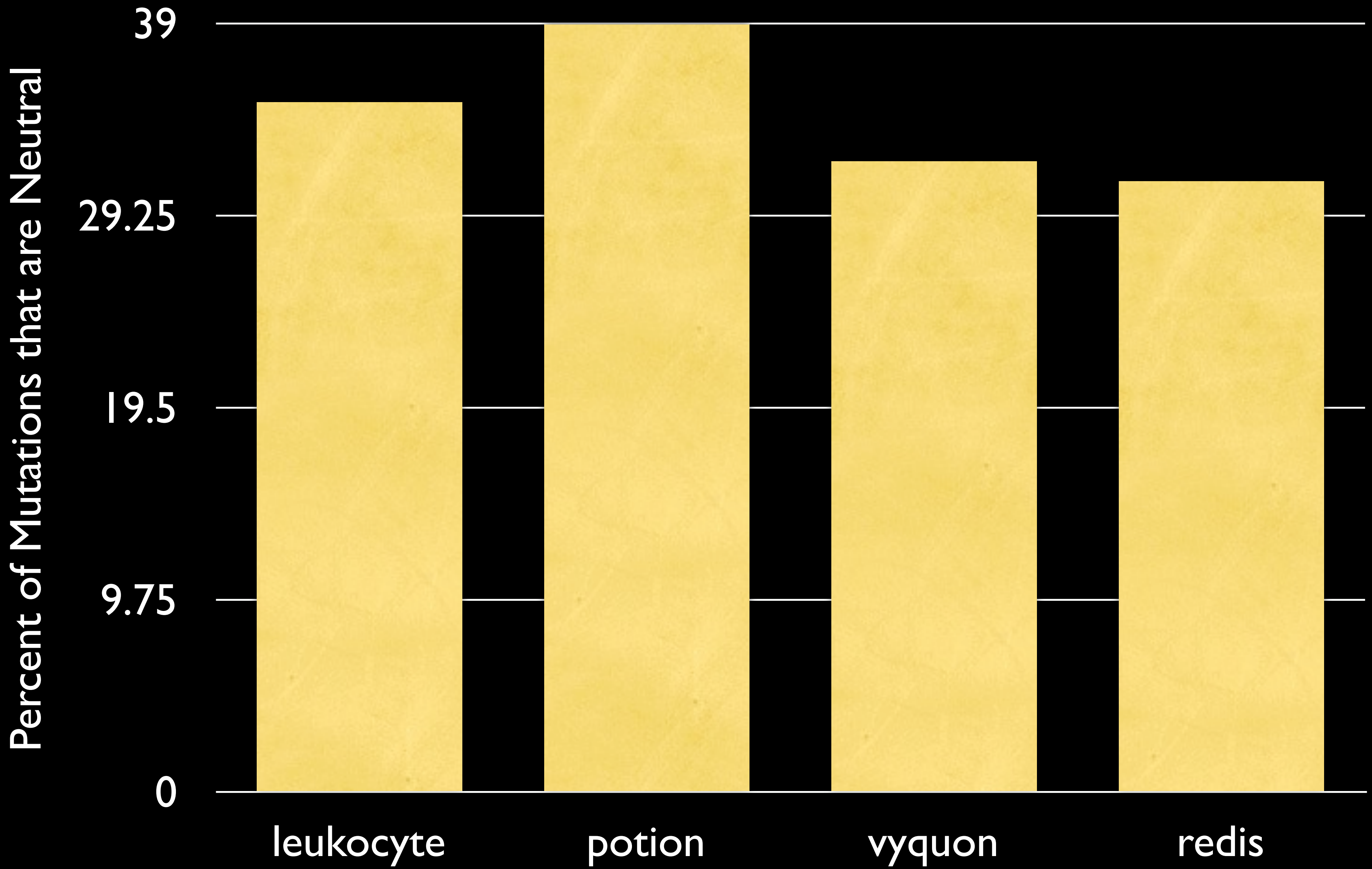
Robustness vs. Code Size



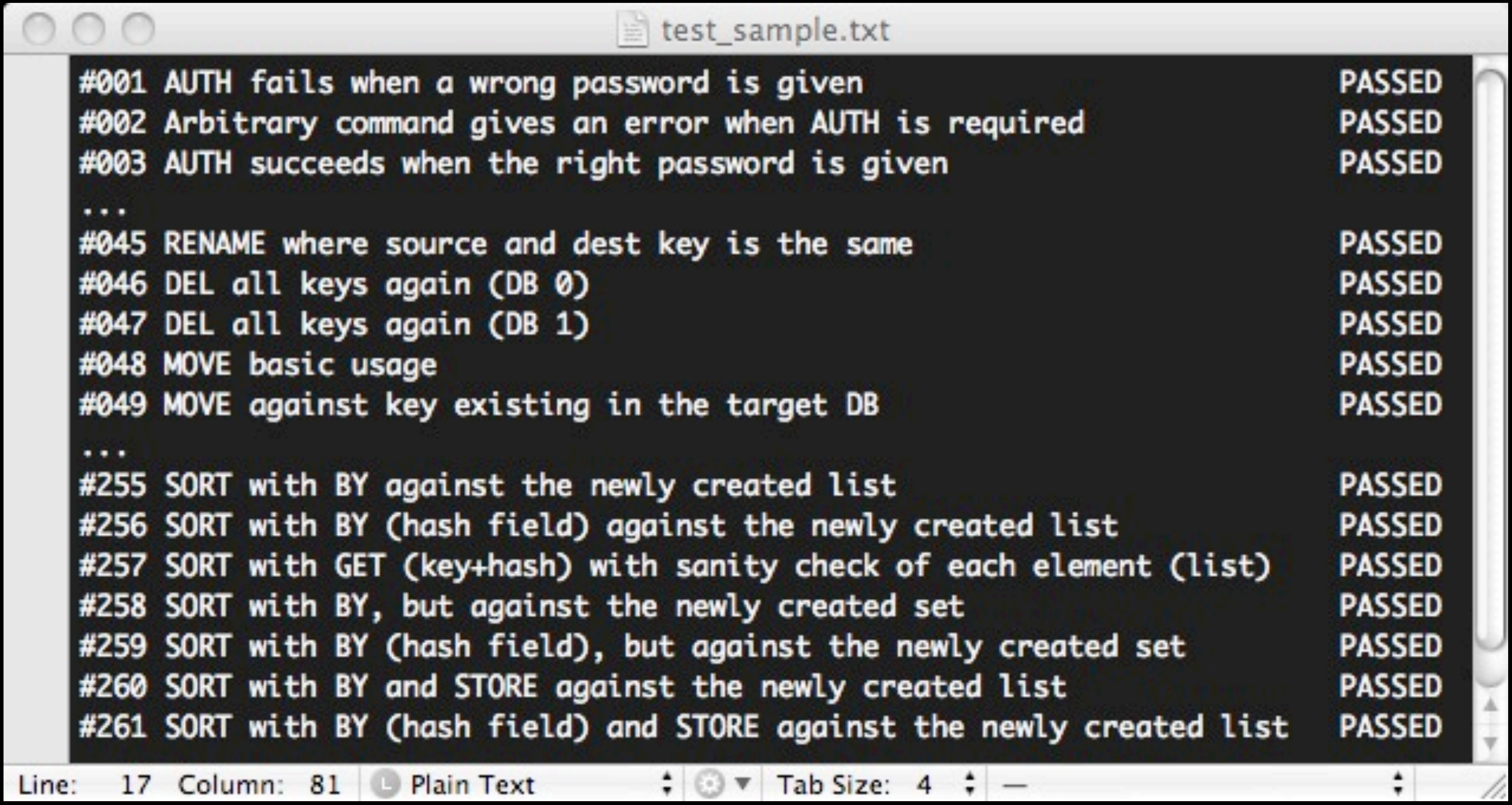
But perhaps my tests suites are simply quite terrible?

Do these results actually generalize?

Neutral Mutations on Large Suites

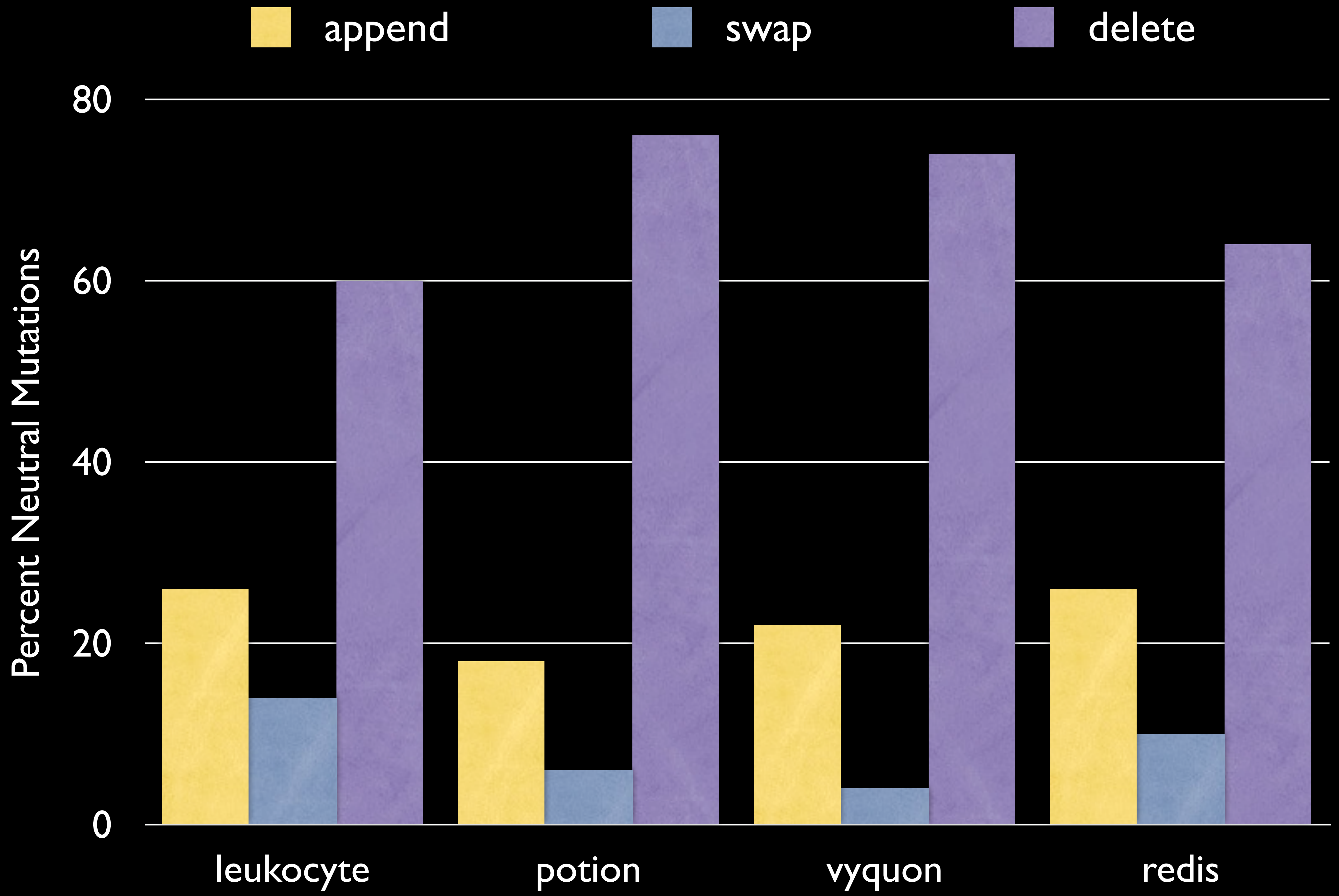


A Non-Trivial Test Suite



```
#001 AUTH fails when a wrong password is given                           PASSED
#002 Arbitrary command gives an error when AUTH is required              PASSED
#003 AUTH succeeds when the right password is given                      PASSED
...
#045 RENAME where source and dest key is the same                        PASSED
#046 DEL all keys again (DB 0)                                           PASSED
#047 DEL all keys again (DB 1)                                           PASSED
#048 MOVE basic usage                                                    PASSED
#049 MOVE against key existing in the target DB                          PASSED
...
#255 SORT with BY against the newly created list                         PASSED
#256 SORT with BY (hash field) against the newly created list            PASSED
#257 SORT with GET (key+hash) with sanity check of each element (list)   PASSED
#258 SORT with BY, but against the newly created set                     PASSED
#259 SORT with BY (hash field), but against the newly created set        PASSED
#260 SORT with BY and STORE against the newly created list               PASSED
#261 SORT with BY (hash field) and STORE against the newly created list  PASSED
```

By Mutation Operators



Stepping Back

Surprising to see such high levels of mutational robustness, at this level of representation

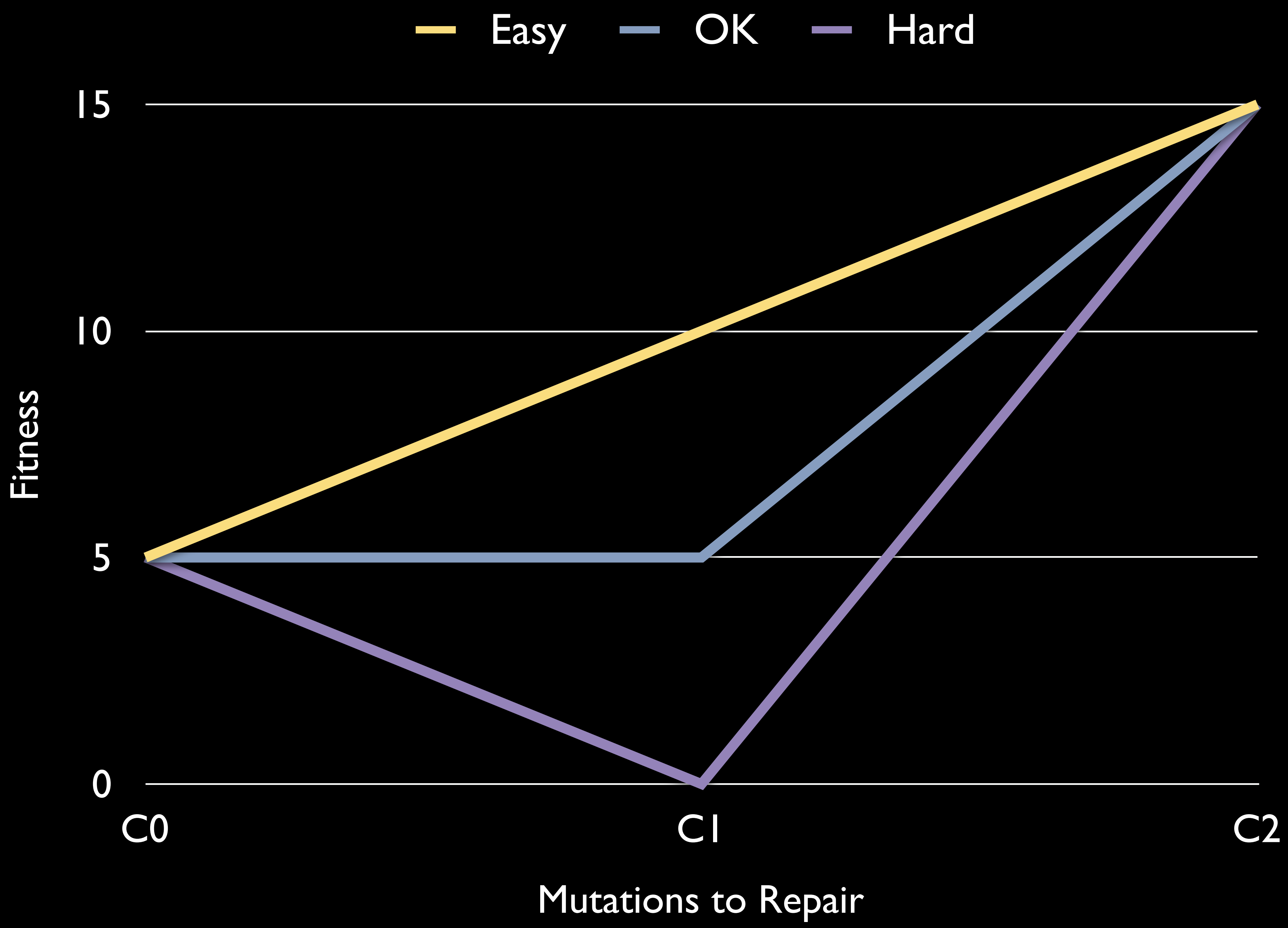
Possibly contributes * to the success of Program Repair via GP

Quite counter-intuitive (so we assert)

* robustness != good (tradeoff with evolvability)

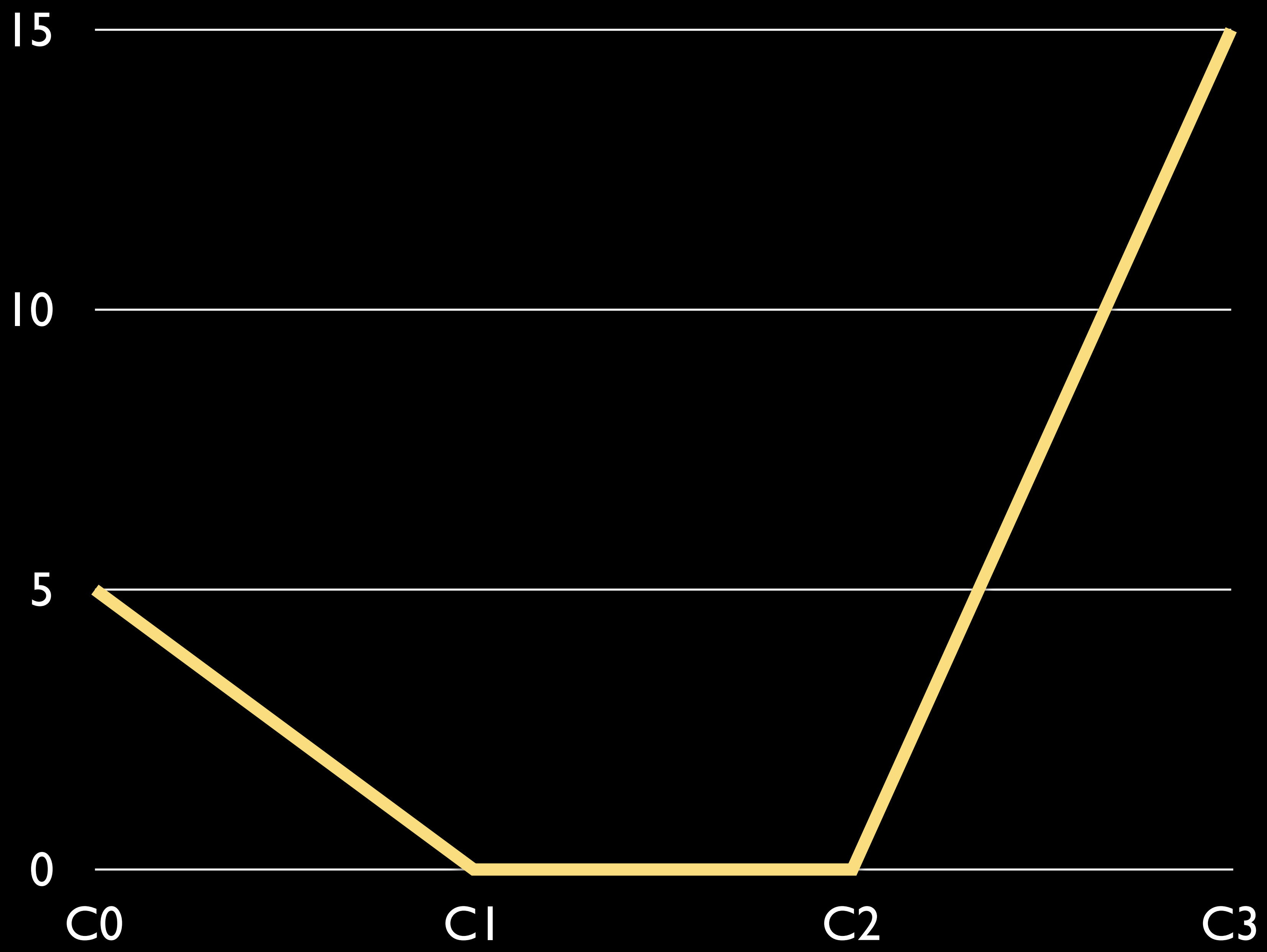
Relating Robustness to Repair Difficulty

A Problem?



Three-Step Repair

A pathological case study



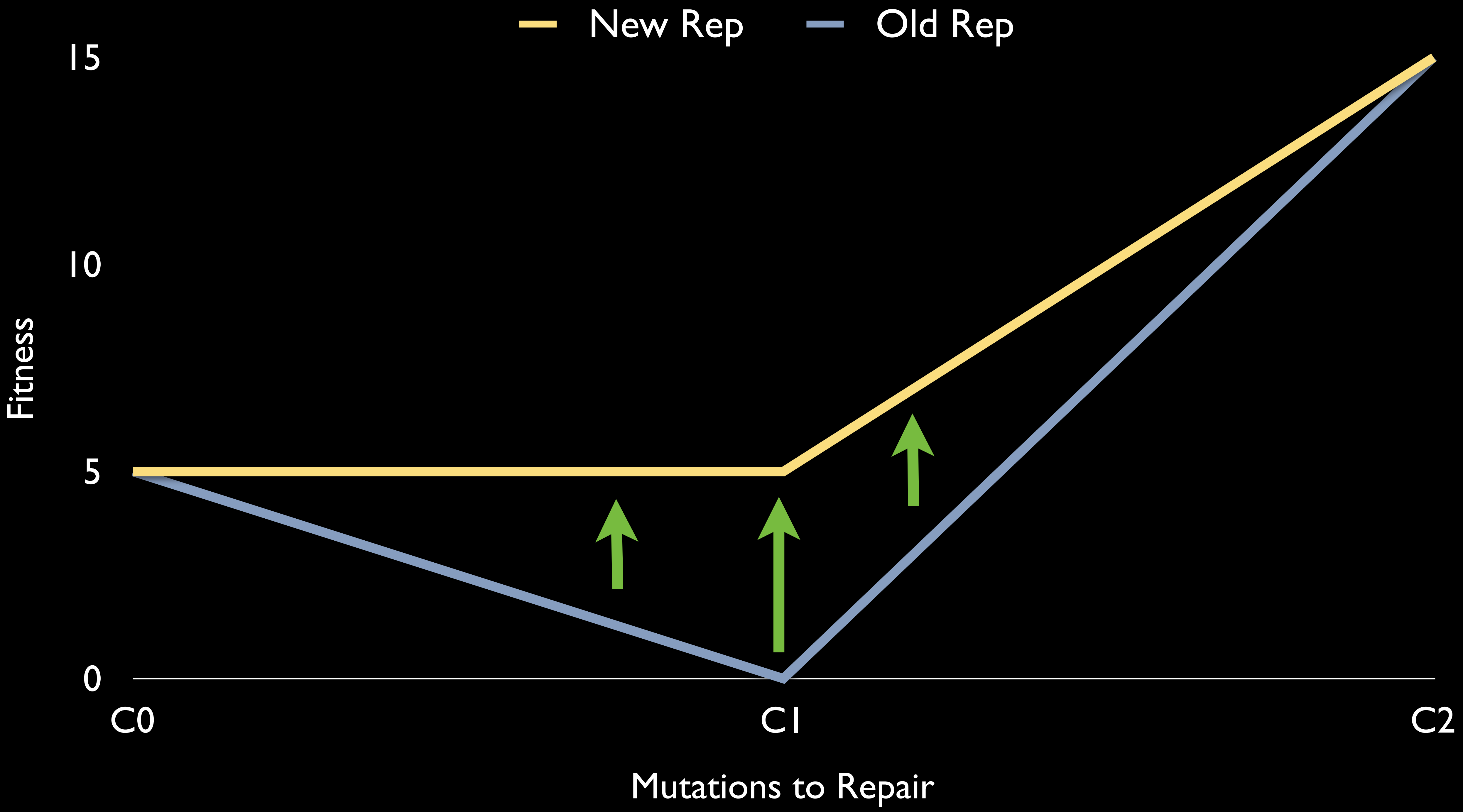
How might we solve this?

Use a new representation, with a higher degree of mutational robustness

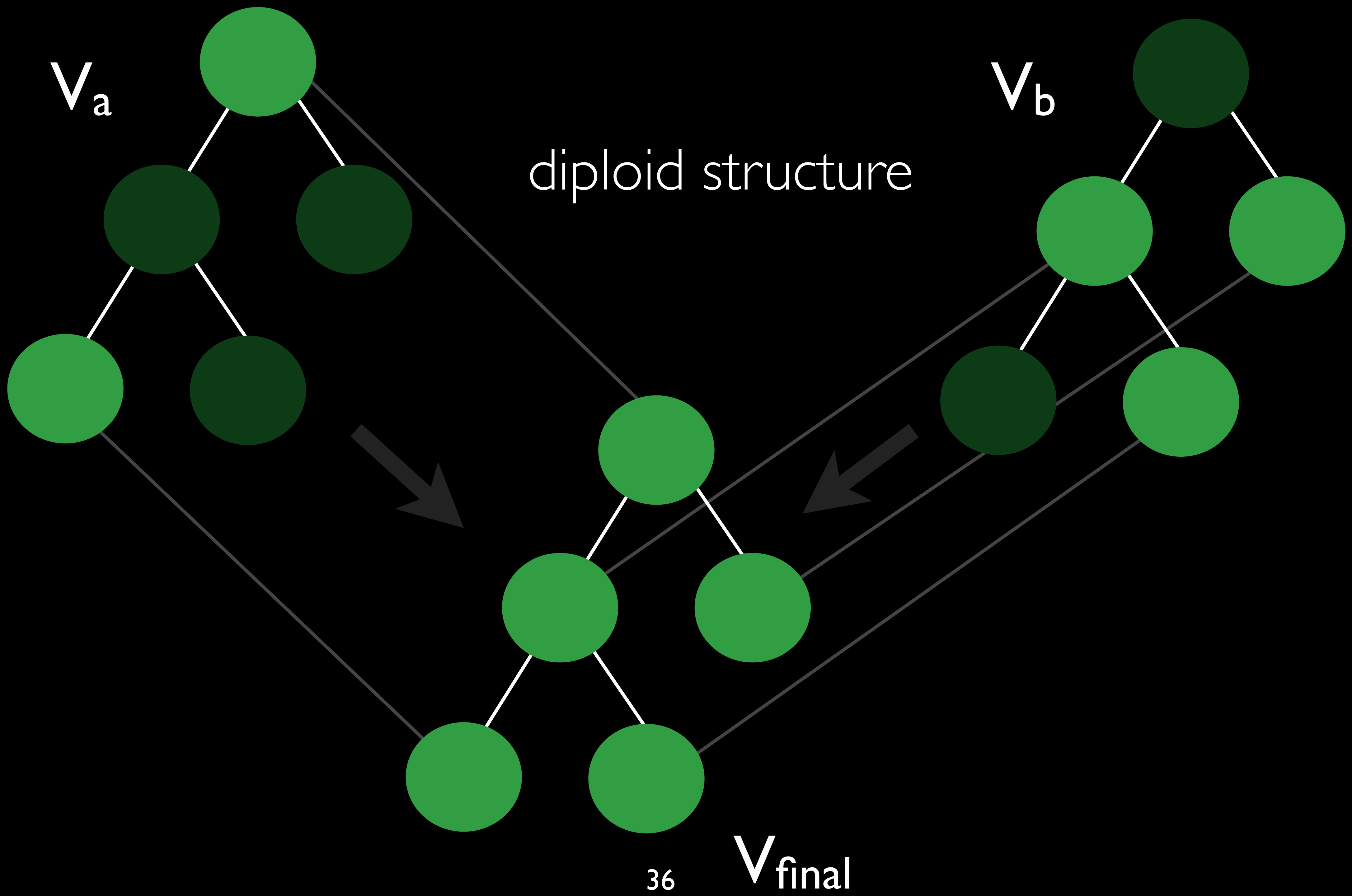
Inspiration: a diploid chromosomal structure

Change the gradient of the fitness landscape leading to repair

The Basic Idea



New Representation



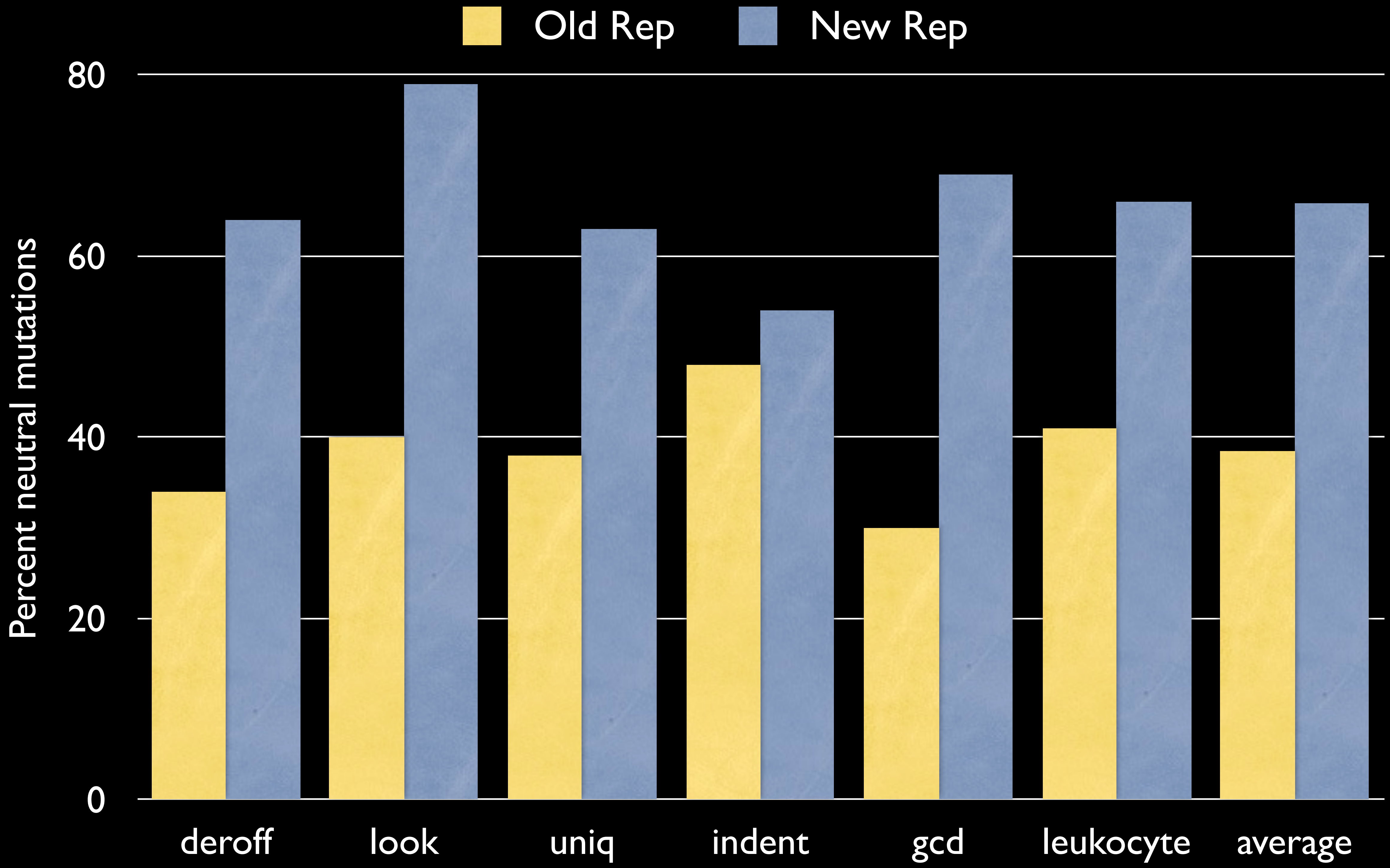
Upshot

Mutations can be made to program segments that are not applied functionally

A smoother fitness gradient to repair

Innovation: Occasionally these non-functional mutations will be transformed into functional mutations

New Rep More Robust?



Preliminary Results

Of a Mixed Nature

- Two-step repair found 3x as often
- Three-step repair never found

Working on Additional Strategies

- Different representations
- Fitness function

Conclusions

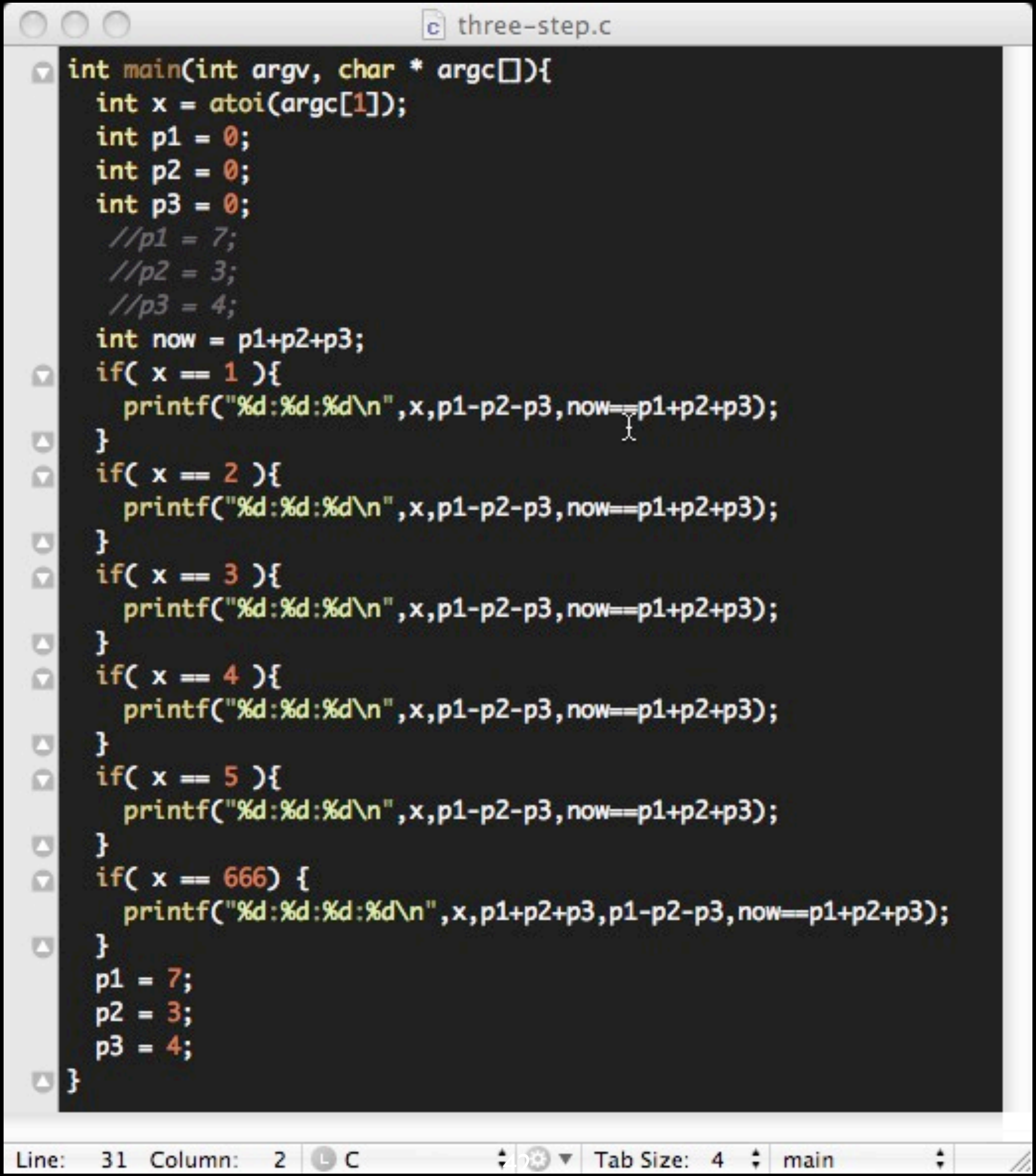
Programs are surprisingly robust

Result holds for large and complicated programs and test suites

But more robust representations may help in repairing certain kinds of bugs

Questions?

Suggestions are also welcome



three-step.c

```c
int main(int argv, char * argc[]){
  int x = atoi(argc[1]);
  int p1 = 0;
  int p2 = 0;
  int p3 = 0;
   //p1 = 7;
   //p2 = 3;
   //p3 = 4;
  int now = p1+p2+p3;
  if( x == 1 ){
    printf("%d:%d:%d\n",x,p1-p2-p3,now==p1+p2+p3);
  }
  if( x == 2 ){
    printf("%d:%d:%d\n",x,p1-p2-p3,now==p1+p2+p3);
  }
  if( x == 3 ){
    printf("%d:%d:%d\n",x,p1-p2-p3,now==p1+p2+p3);
  }
  if( x == 4 ){
    printf("%d:%d:%d\n",x,p1-p2-p3,now==p1+p2+p3);
  }
  if( x == 5 ){
    printf("%d:%d:%d\n",x,p1-p2-p3,now==p1+p2+p3);
  }
  if( x == 666) {
    printf("%d:%d:%d:%d\n",x,p1+p2+p3,p1-p2-p3,now==p1+p2+p3);
  }
  p1 = 7;
  p2 = 3;
  p3 = 4;
}
```

Line: 31 Column: 2 C Tab Size: 4 main

Robustness Benchmark

Program	MR *	Neutral **
deroff	20%	34%
look	20%	40%
uniq	24%	38%
indent	16%	48%
gcd	23%	30%
leukocyte	19%	41%

* measured average change in test case fitness

** percent of mutations that do not affect fitness

With Mutation Operators

Program	MR *	Neutral **	Append	Swap	Delete
deroff	20%	31%	30%	11%	59%
look	20%	43%	40%	14%	46%
uniq	24%	34%	43%	14%	43%
gcd	23%	34%	55%	19%	25%
leukocyte	19%	39%	12%	23%	64%

* measured average change in test case fitness

** percent of mutations that do not affect fitness

With No Path Weights

Program	Neutral	Append	Swap	Delete
deroff	60%	28%	20%	52%
look	53%	34%	15%	51%
uniq	55%	27%	17%	56%
gcd	37%	61%	11%	28%
leukocyte	39%	32%	13%	56%

Even more robust to random mutations

For Larger Test Suites?

Program	Neutral	Append	Swap	Delete
leukocyte	35%	26%	14%	60%
potion	39%	18%	6%	76%
vyquon	32%	22%	4%	74%
redis	31%	26%	10%	64%

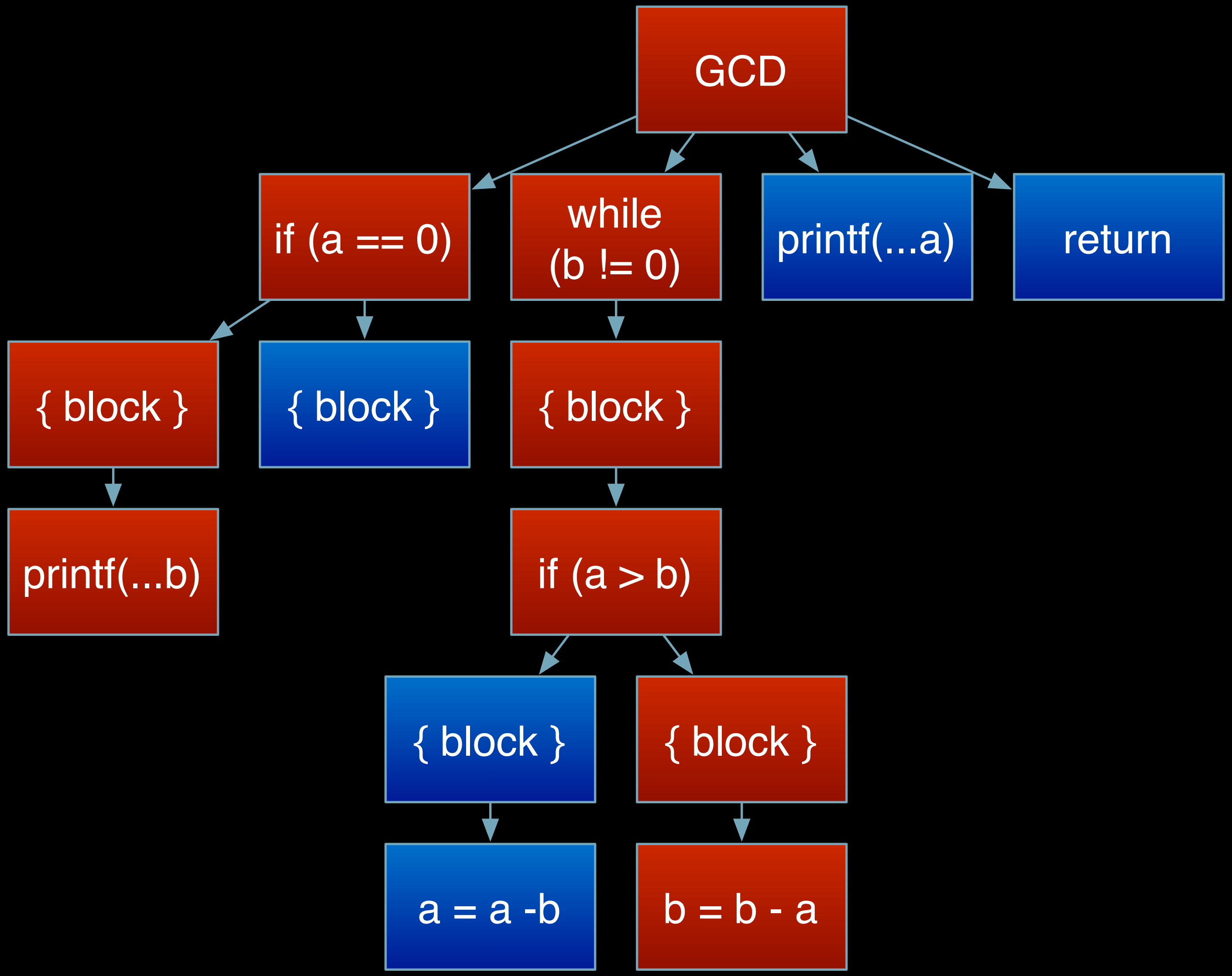
Seems not to be artifact of small test suites

New Rep More Robust?

Program	Old Rep *	New Rep *
deroff	34%	64%
look	40%	79%
uniq	38%	63%
indent	48%	54%
gcd	30%	69%
leukocyte	41%	66%
Average	38.5%	65.8%

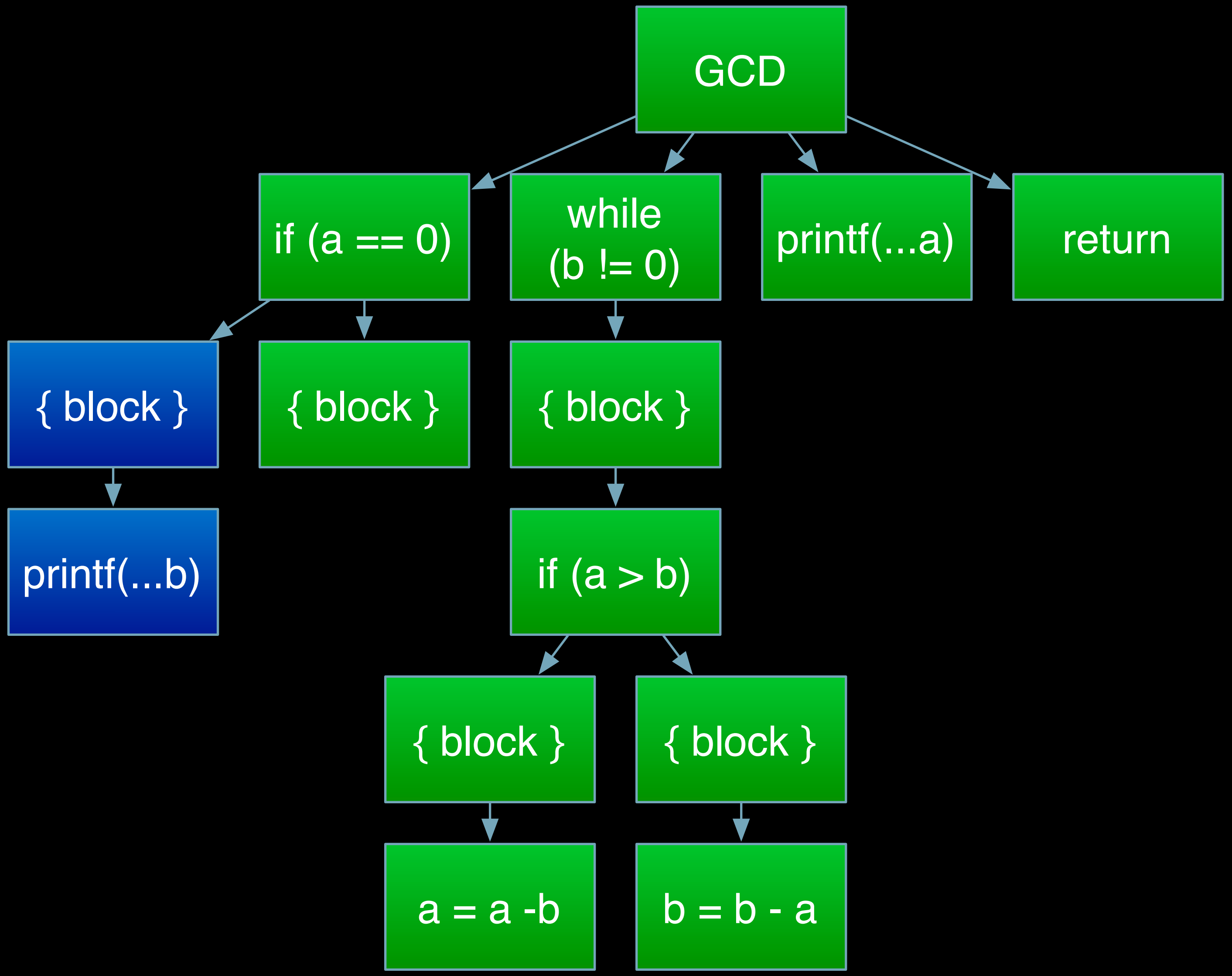
* percent of mutations that do not affect fitness

Weighted Path



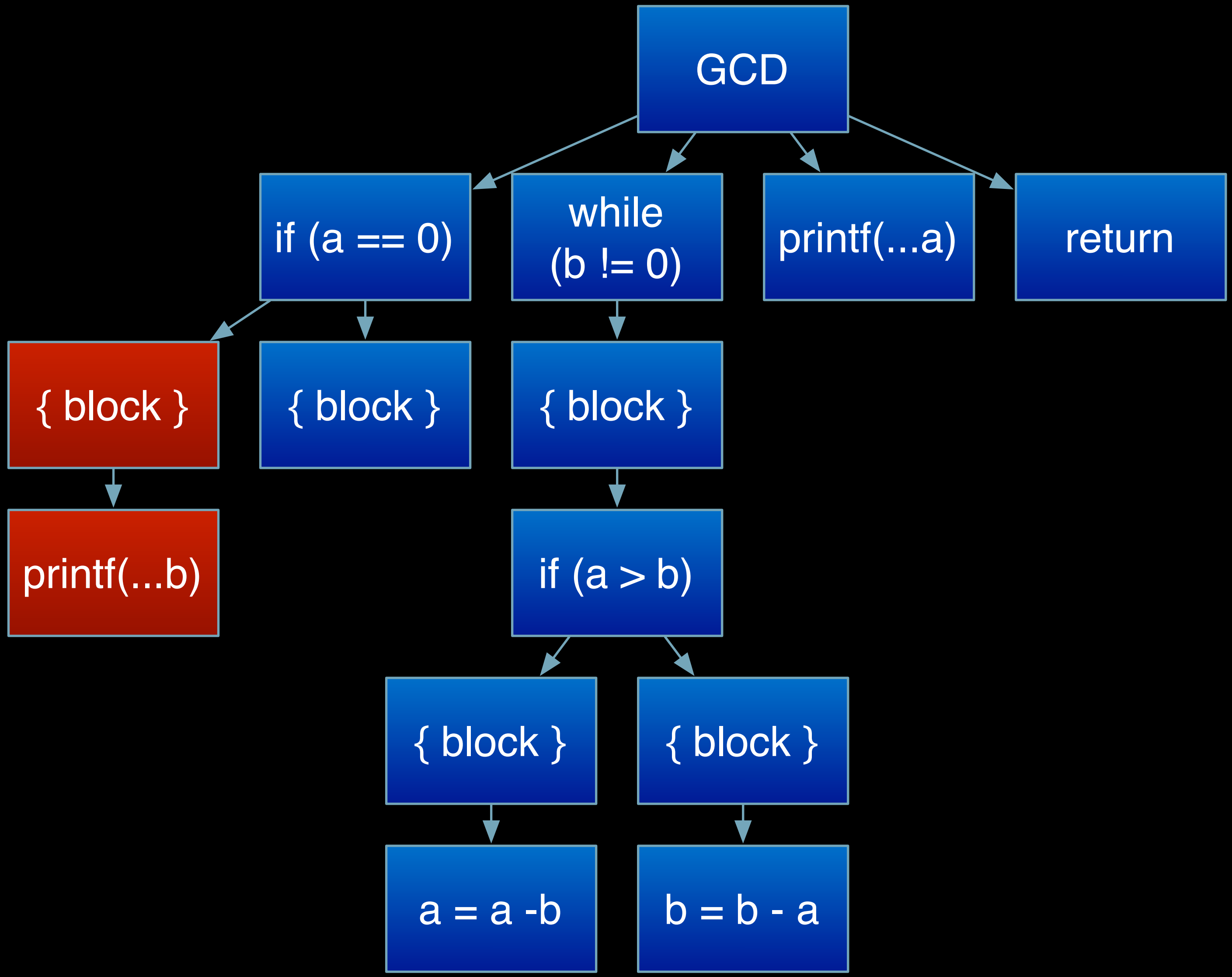
Negative Test Case

Weighted Path



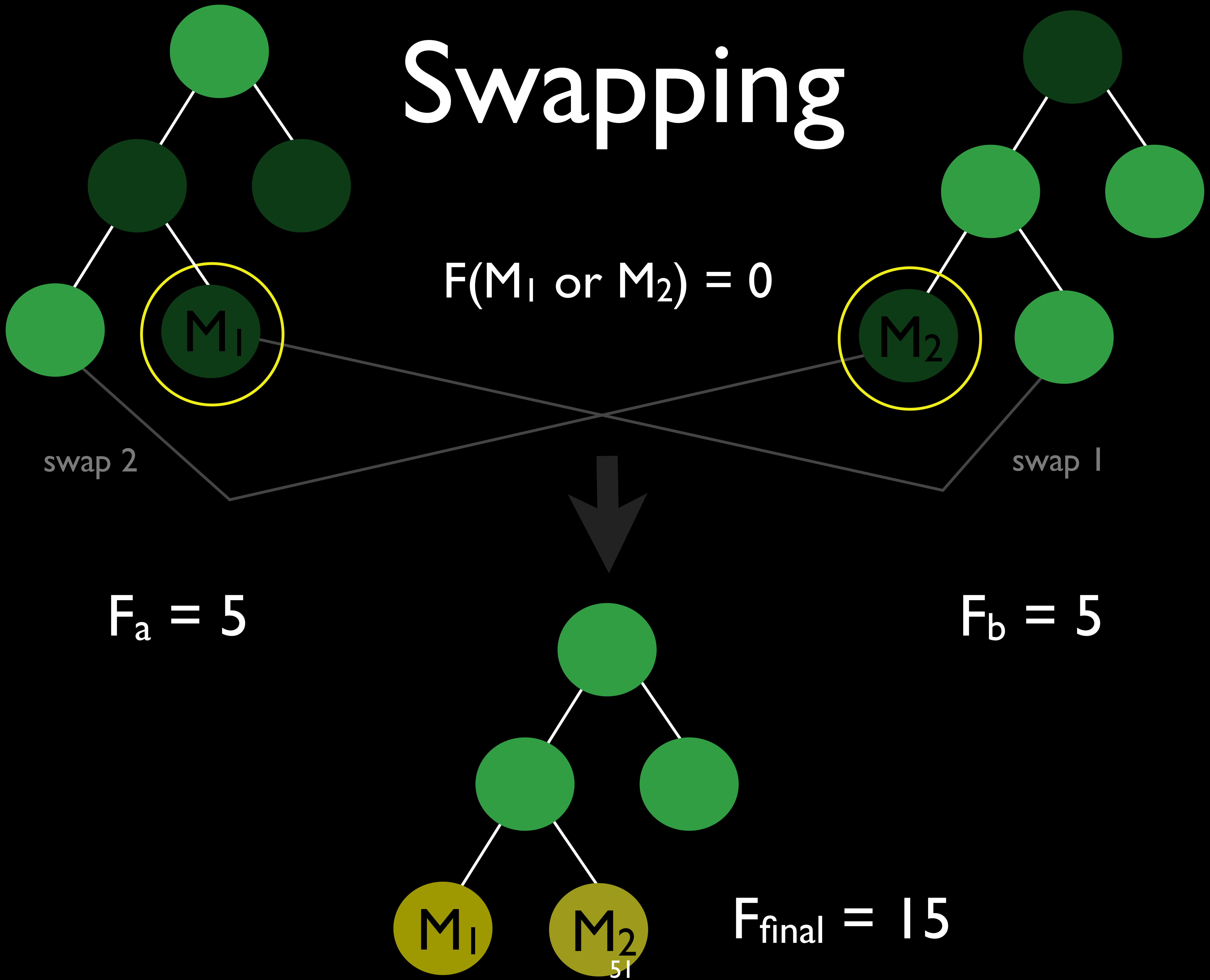
Positive Test Case

Weighted Path



Final Path

Swapping



$F(M_1 \text{ or } M_2) = 0$

$F_a = 5$

$F_b = 5$

$F_{final} = 15$