Taking Search-based Software Testing to the Real World

Robert Feldt
Professor of Software Engineering
Chalmers University and Blekinge Inst of Tech
robert.feldt@gmail.com

1. Search / Optimization / Machine Learning

- useful tools for improving testing!

2. No guarantees - but useful in practice

- less formal than many alternatives
- no guarantees in testing anyway

3. Good when “exact” alg missing

- more problems than specialized solutions
- search/optimization can adapt

What is SBSE?

What is SBST?

Why is it not real-world (enough)?

Examples of real-world applications:

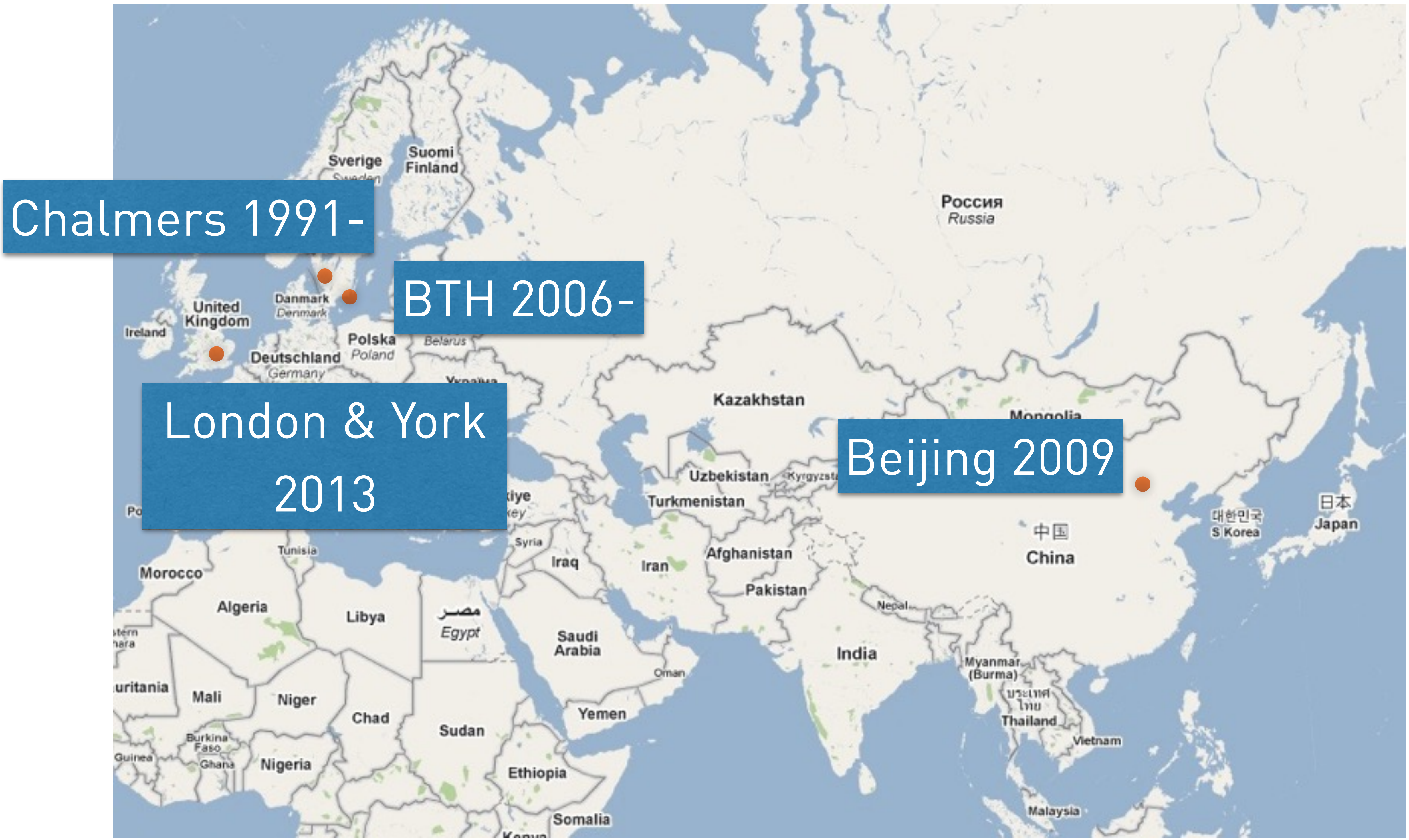
- Optimizing test case selection
- Generating complex test data
- Searching for diverse test suites

Who am I?

Tech Competence/Innovation, Broad knowledge SE

Consulting in telecom, aerospace industry, AI/Machine learning

Early pioneer of SBSE, Dynamic programming languages



What is SBSE?

What is SBST?

Why is it not real-world (enough)?

Examples of real-world applications:

- Optimizing test case selection
- Generating complex test data
- Searching for diverse test suites

What is search-based software engineering?

Search-based software engineering

From Wikipedia, the free encyclopedia

Search-based software engineering (**SBSE**) applies metaheuristic search techniques such as genetic algorithms, simulated annealing and tabu search to software engineering problems. Many activities in software engineering can be stated as optimization problems. Optimization techniques of operations research such as linear programming or dynamic programming are mostly impractical for large scale software engineering problems because of their computational complexity. Researchers and practitioners use metaheuristic search techniques to find near-optimal or "good-enough" solutions.

Search-Based Software Engineering (SBSE)

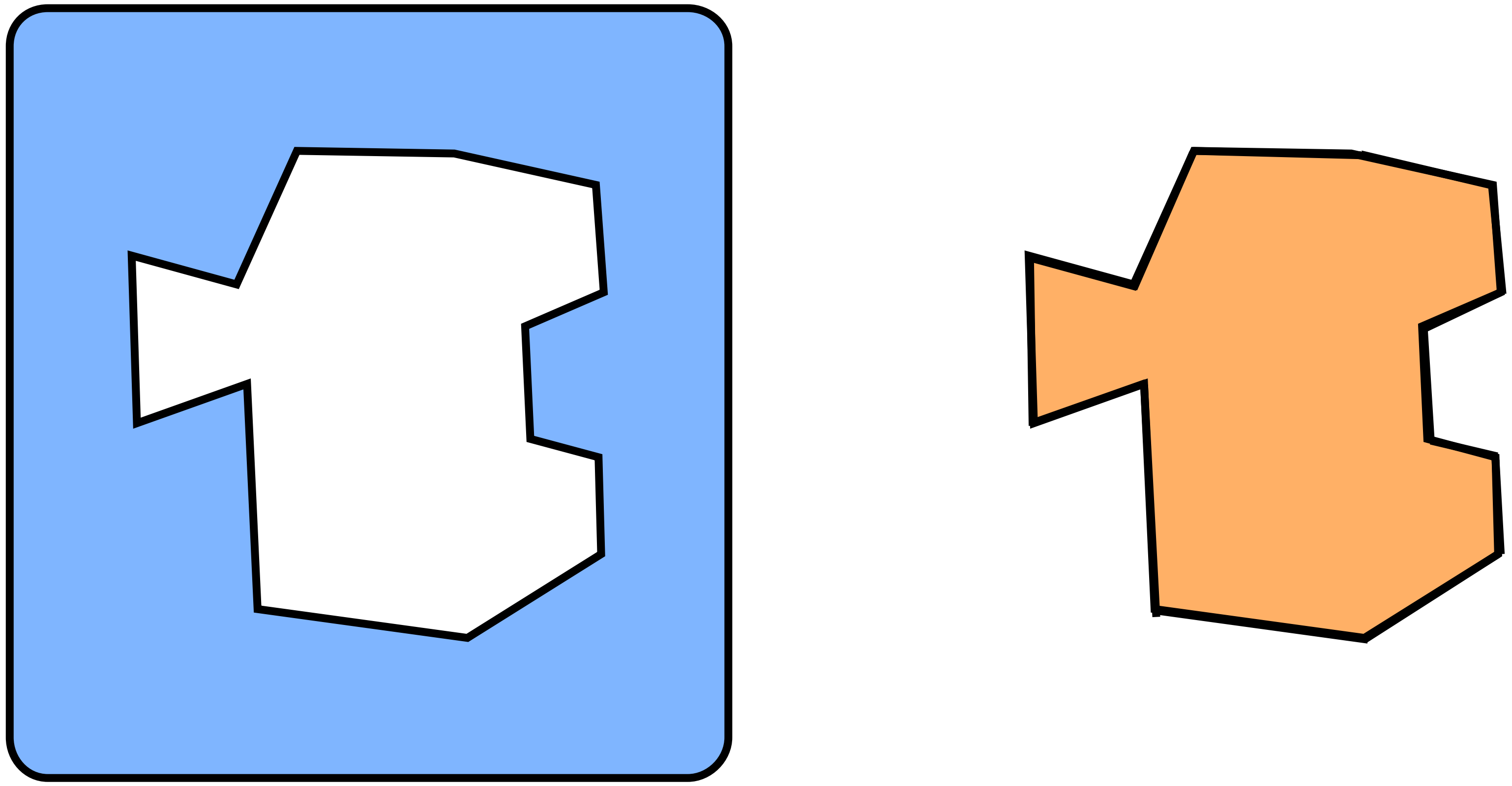
Many software engineering problems have the property that:

constructing a solution: difficult

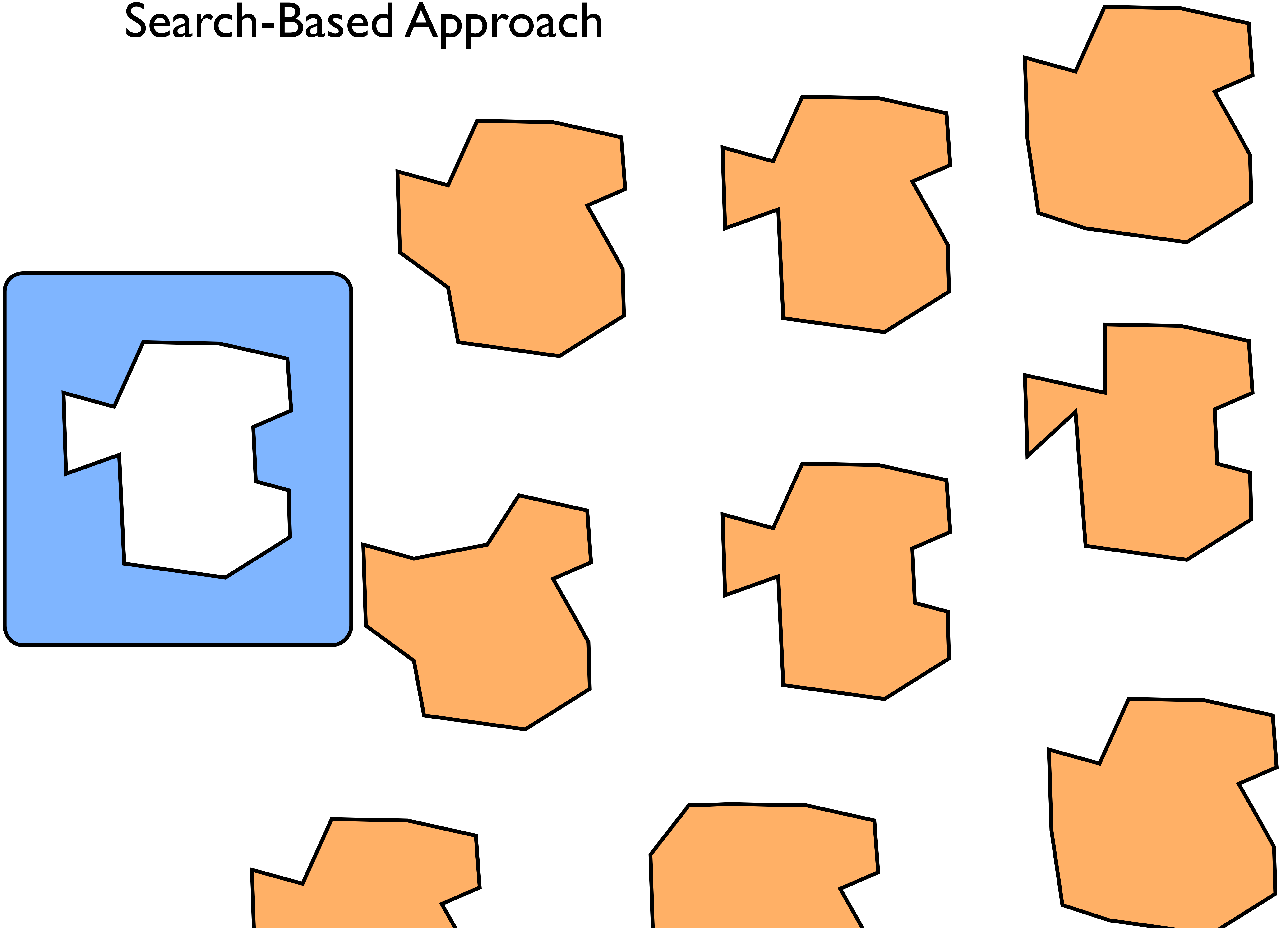
checking a potential solution: easy

Such tasks are amenable to solution using modern optimisation ("search") algorithms such as evolutionary computation

Traditional Approach



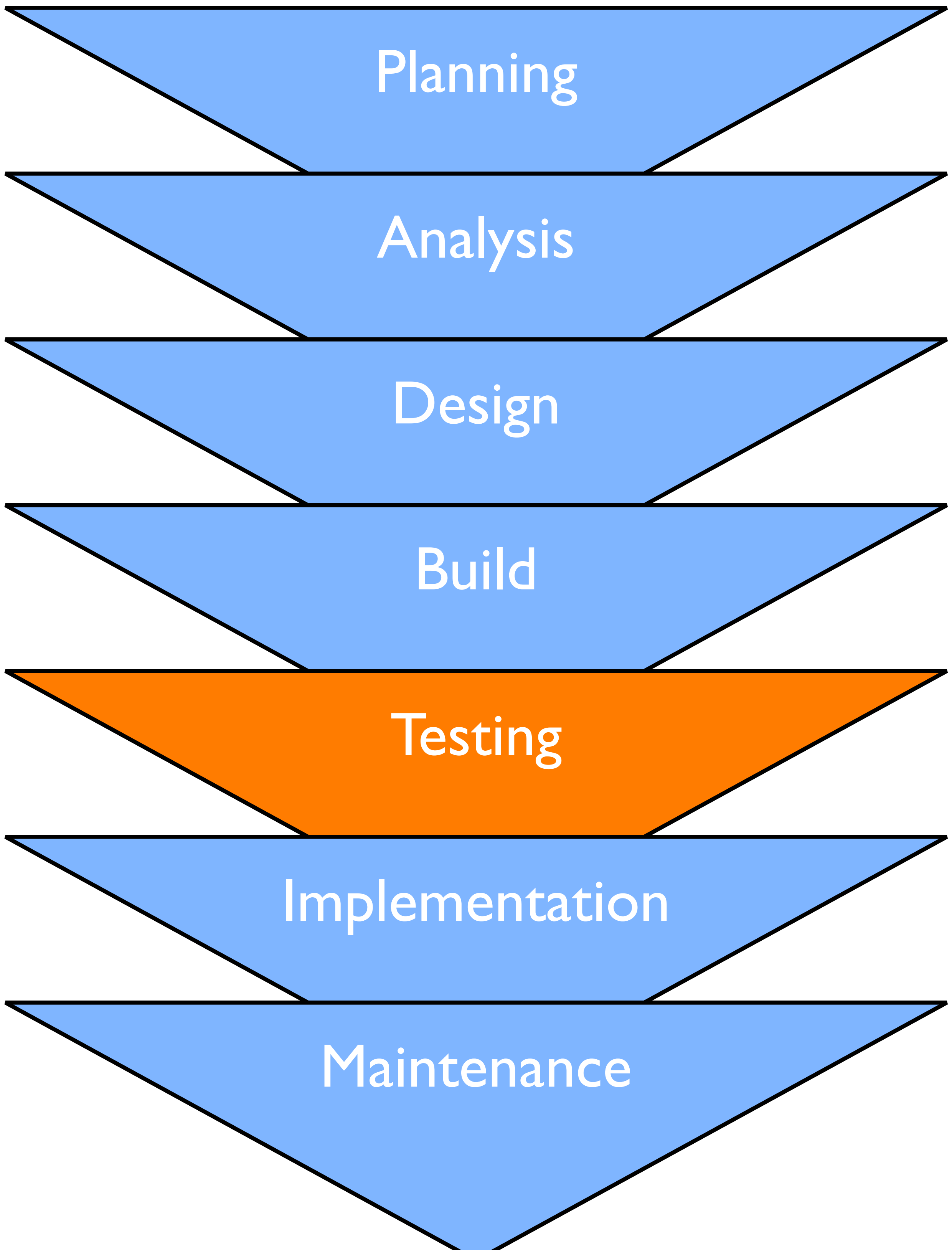
Search-Based Approach



SBSE - Advantages

- automation
- scalability
    - complex problems become tractable
    - leverages high-performance computing power
- lack of bias
    - innovation
    - more diversity
- multi-objective problems

SBSE - Applications



project planning

feature selection

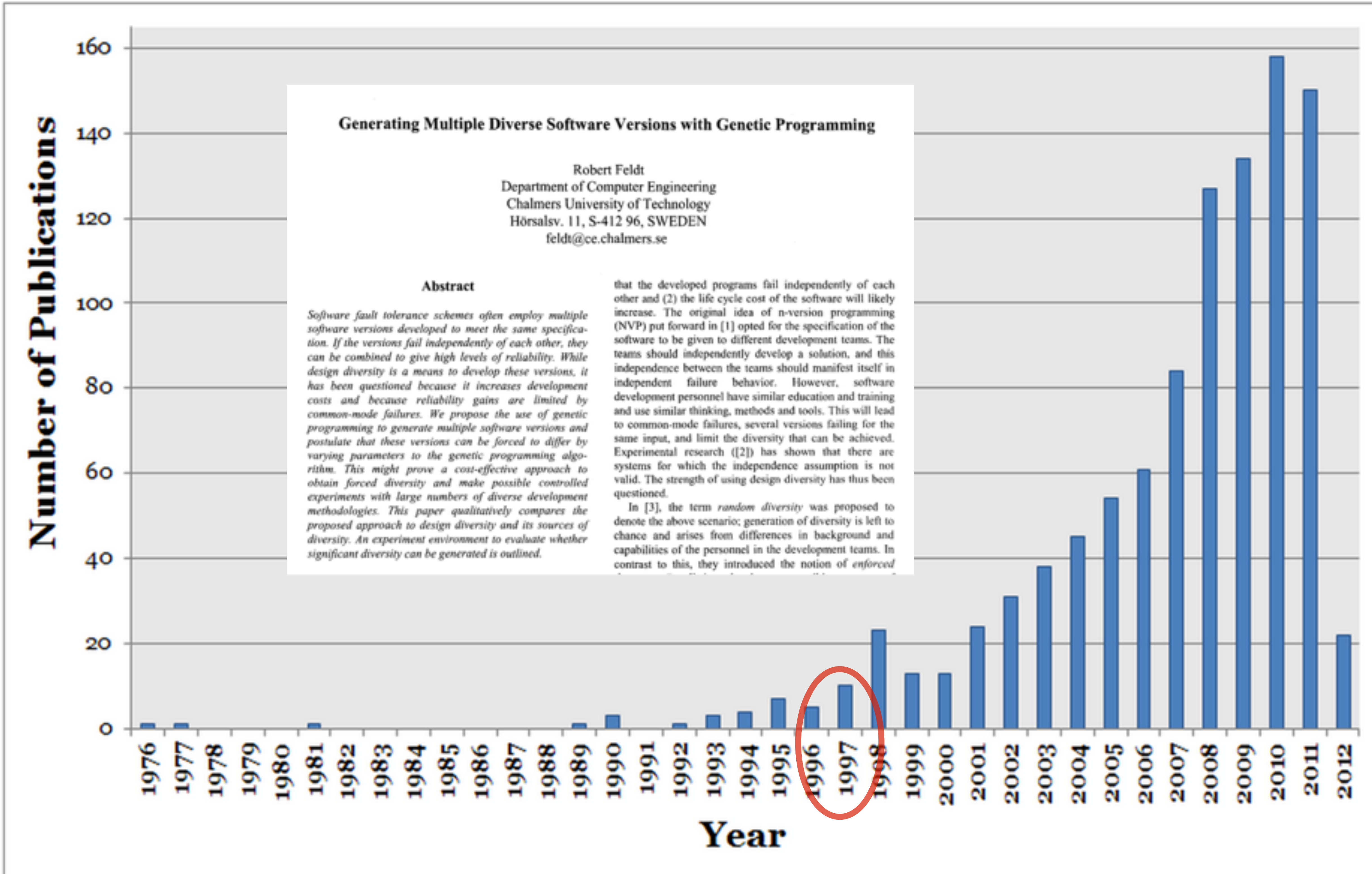
real-time systems design
module clustering
protocol synthesis
concurrent software verification
algorithm construction
code refactoring

test data generation
test case selection
test case prioritisation

system tuning

automated bug fixing

Why is interest in SB/ML-SE growing?



Generating Multiple Diverse Software Versions with Genetic Programming

Robert Feldt
Department of Computer Engineering
Chalmers University of Technology
Hörsalsv. 11, S-412 96, SWEDEN
feldt@ce.chalmers.se

Abstract

Software fault tolerance schemes often employ multiple software versions developed to meet the same specification. If the versions fail independently of each other, they can be combined to give high levels of reliability. While design diversity is a means to develop these versions, it has been questioned because it increases development costs and because reliability gains are limited by common-mode failures. We propose the use of genetic programming to generate multiple software versions and postulate that these versions can be forced to differ by varying parameters to the genetic programming algorithm. This might prove a cost-effective approach to obtain forced diversity and make possible controlled experiments with large numbers of diverse development methodologies. This paper qualitatively compares the proposed approach to design diversity and its sources of diversity. An experiment environment to evaluate whether significant diversity can be generated is outlined.

that the developed programs fail independently of each other and (2) the life cycle cost of the software will likely increase. The original idea of n-version programming (NVP) put forward in [1] opted for the specification of the software to be given to different development teams. The teams should independently develop a solution, and this independence between the teams should manifest itself in independent failure behavior. However, software development personnel have similar education and training and use similar thinking, methods and tools. This will lead to common-mode failures, several versions failing for the same input, and limit the diversity that can be achieved. Experimental research ([2]) has shown that there are systems for which the independence assumption is not valid. The strength of using design diversity has thus been questioned.

In [3], the term *random diversity* was proposed to denote the above scenario; generation of diversity is left to chance and arises from differences in background and capabilities of the personnel in the development teams. In contrast to this, they introduced the notion of *enforced*

Why is interest in SB/ML-SE growing?

The world's top supercomputer: the Tihane-2. Jack Dongarra

Exchange human time with CPU power

Good enough solutions without specifying details

More problems than time to find specific algorithm

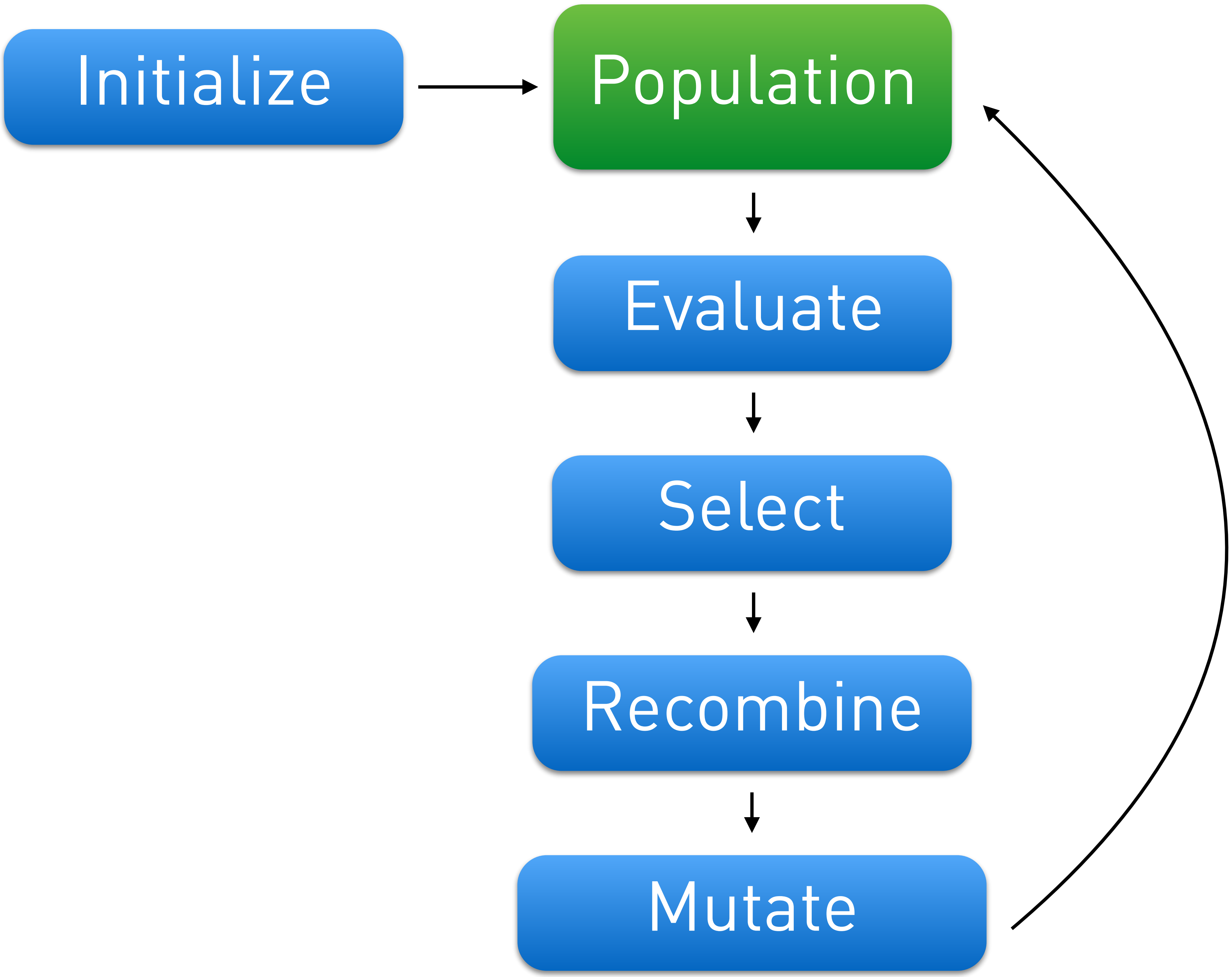
Subfield of Stochastic optimization:

"algorithms that apply some degree of randomness to find (more) optimal solutions to hard problems"

Apply for "I-Know-it-when-I-see-it" problems:

- You lack alg to find optimal solution
- You can't brute-force search
- You CAN score how good a candidate is or
- You CAN score which candidate is better

Example algorithm: Genetic Algorithm



But there are so many other search/opt algorithms!

Simulated Annealing

Hill-climbing

CMA-ES

Differential evolution

Gradient descent

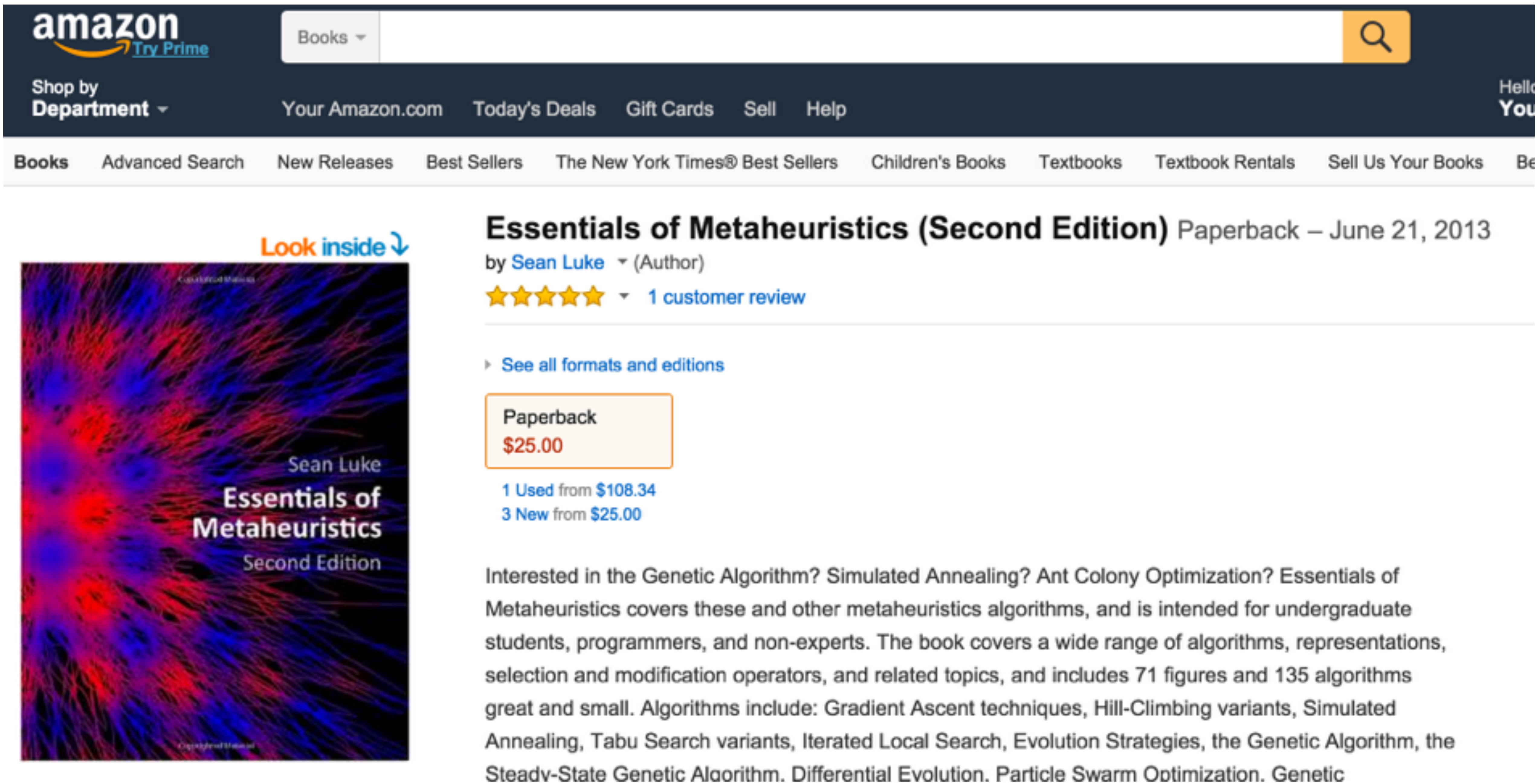
Newton's method

Nesterov's method

Illustration

Learning more about "meta-heuristics"

https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf



- What is SBSE?
- What is SBST?
- Why is it not real-world (enough)?
- Examples of real-world applications:
    - Optimizing test case selection
    - Generating complex test data
    - Searching for diverse test suites

What is Search-Based Software Testing (SBST)?

8th International Workshop on Search-Based Software Testing (SBST) 2015

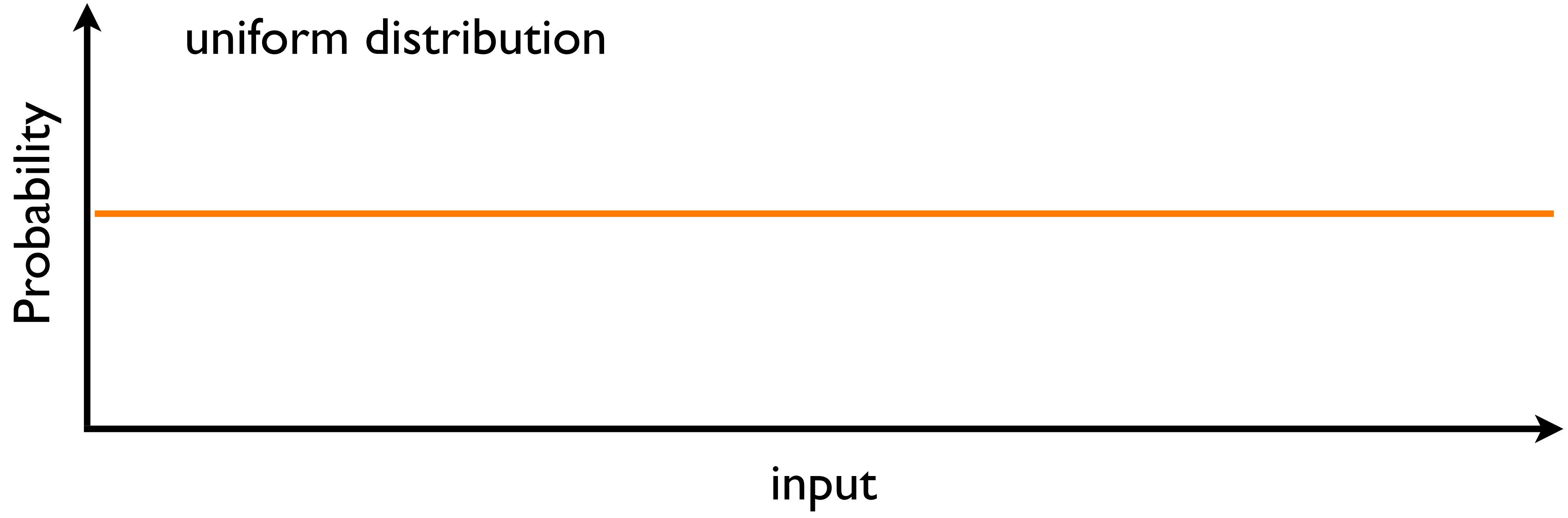
HOME CALL COMMITTEE CONTEST DINNER IMPORTANT DATES KEYNOTES PROGRAM SPONSORS

Search-Based Software Testing (SBST) is the application of optimizing search techniques (for example, Genetic Algorithms) to solve problems in software testing. SBST is used to generate test data, prioritize test cases, minimize test suites, optimize software test oracles, reduce human oracle cost, verify software models, test service-orientated architectures, construct test suites for interaction testing, and validate real-time properties.

Search

NEWS

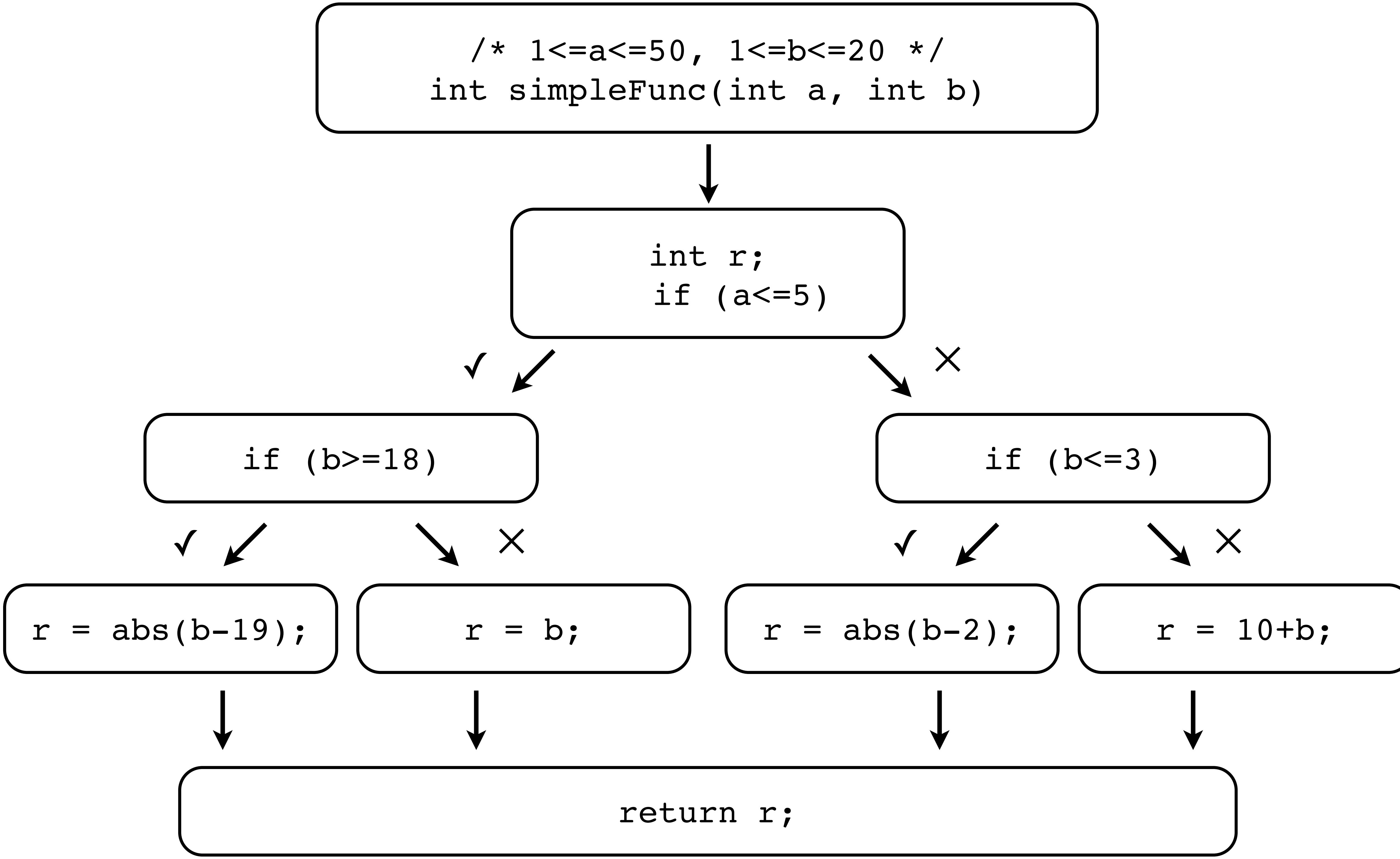
Random Testing



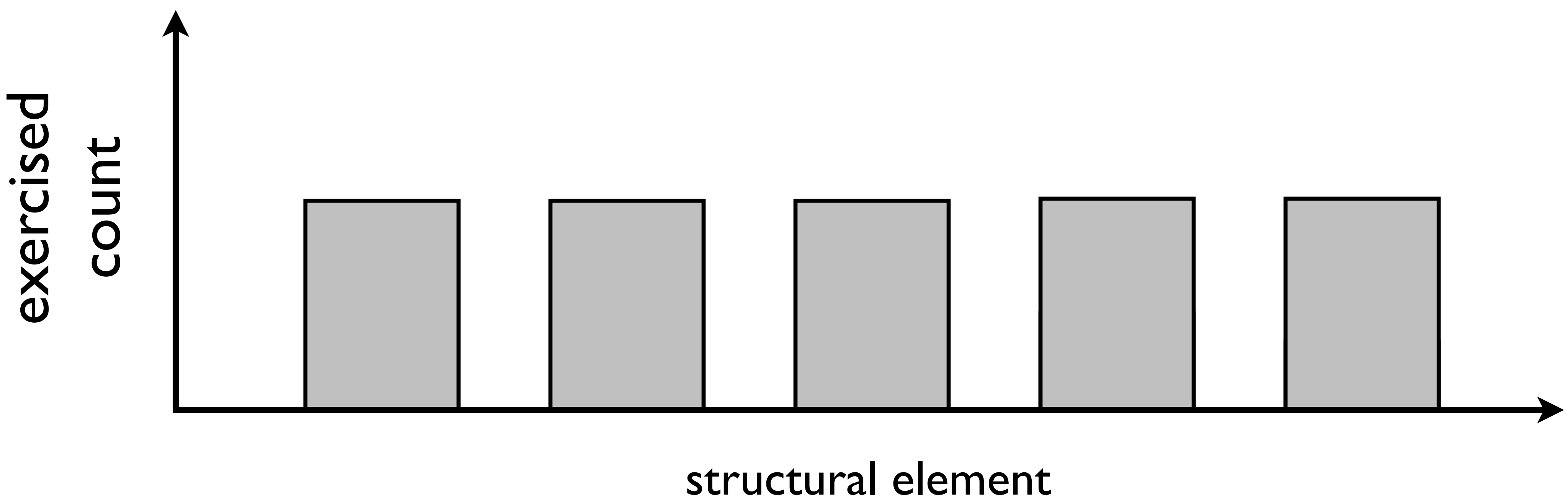
Random Testing

```
/* 1<=a<=50, 1<=b<=20 */
int simpleFunc(int a, int b)
{
 ...
}
```

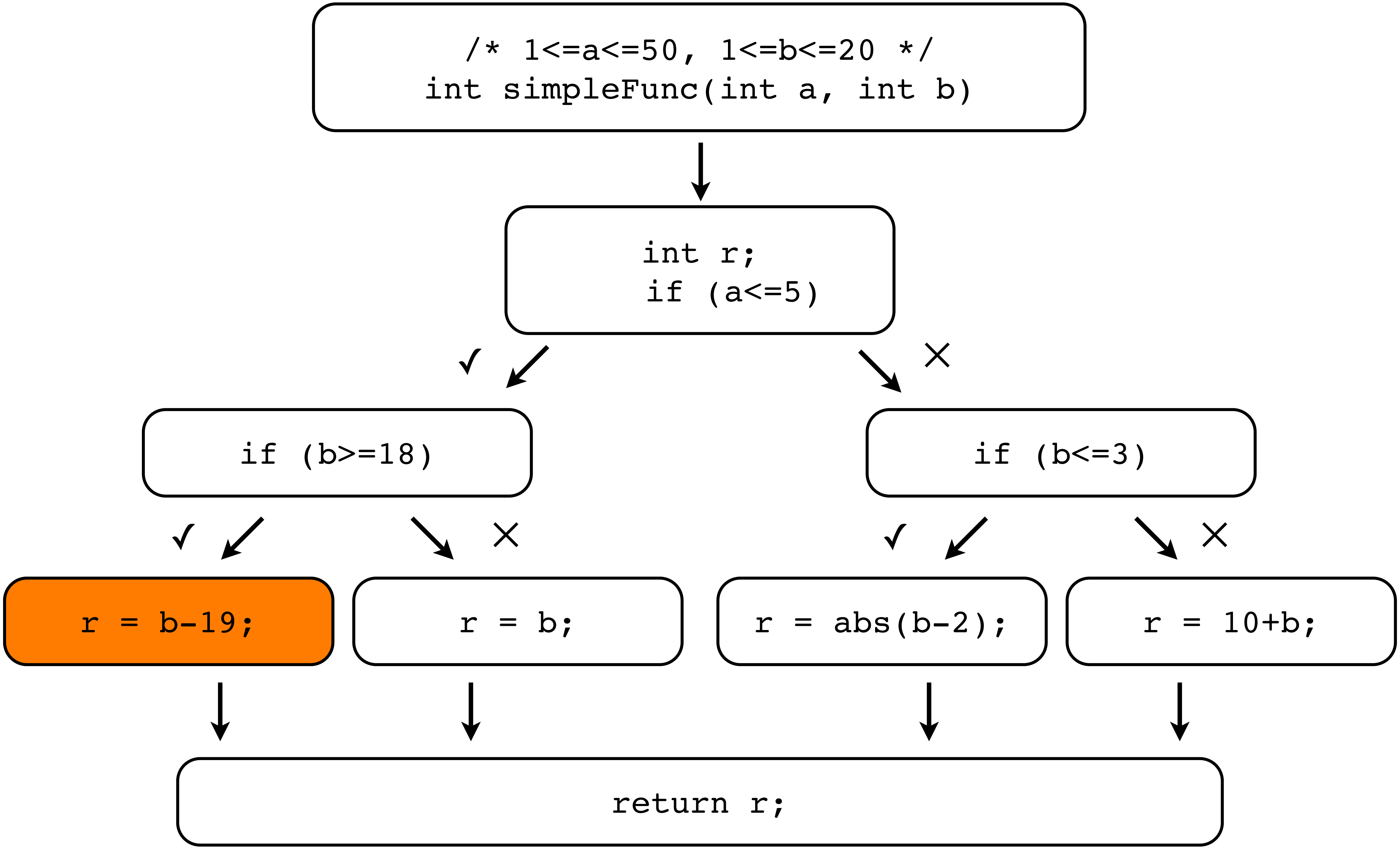
Structural Coverage Testing



Structural Coverage Testing



Structural Coverage Testing



Structural Coverage Testing

- Difficult to derive test sets within specific coverage criteria
- ... but easy to check whether a test set satisfies a coverage criteria

Enter Search-Based Software Testing (SBST)

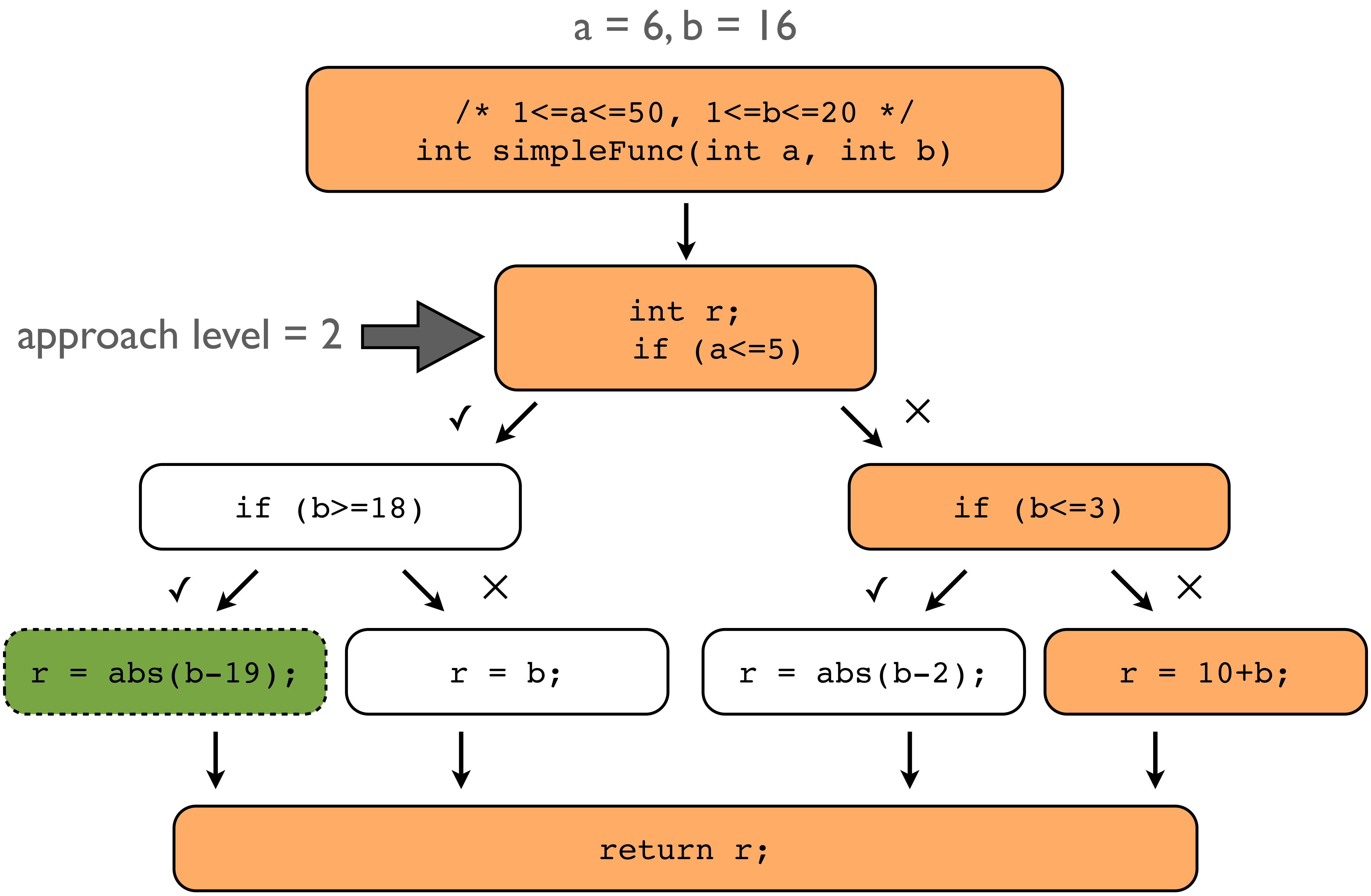
- traditional approach is to considers each coverage element (e.g. each branch) in turn
- find inputs that exercised the element using search methods such as
  - genetic algorithms
  - simulated annealing

Search-Based Software Testing (SBST)

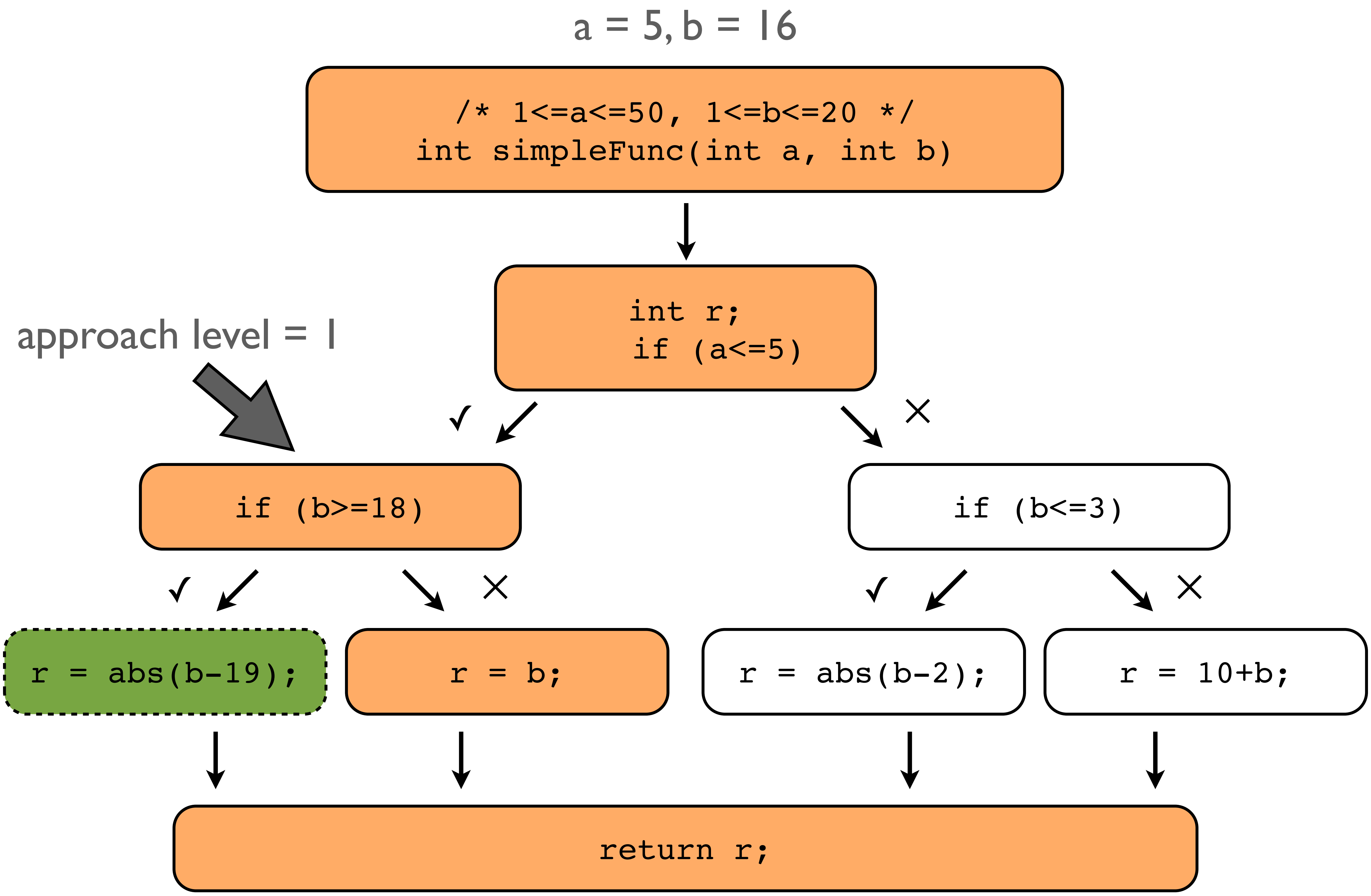
Fitness function based on:

- approach level - how close to executing desired branch condition
- branch distance - how close returning the correct value for the branch condition

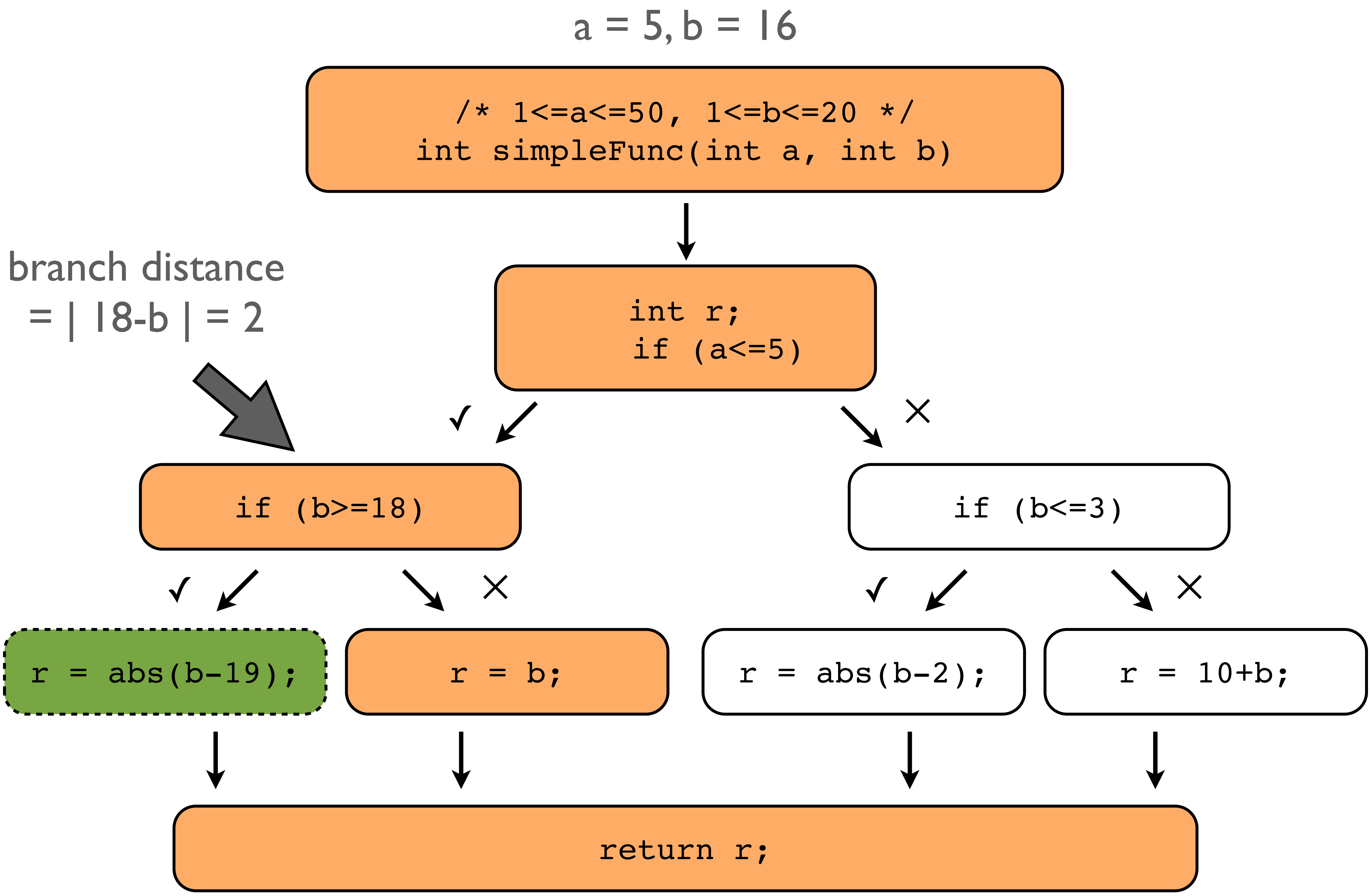
SBST - Approach Level



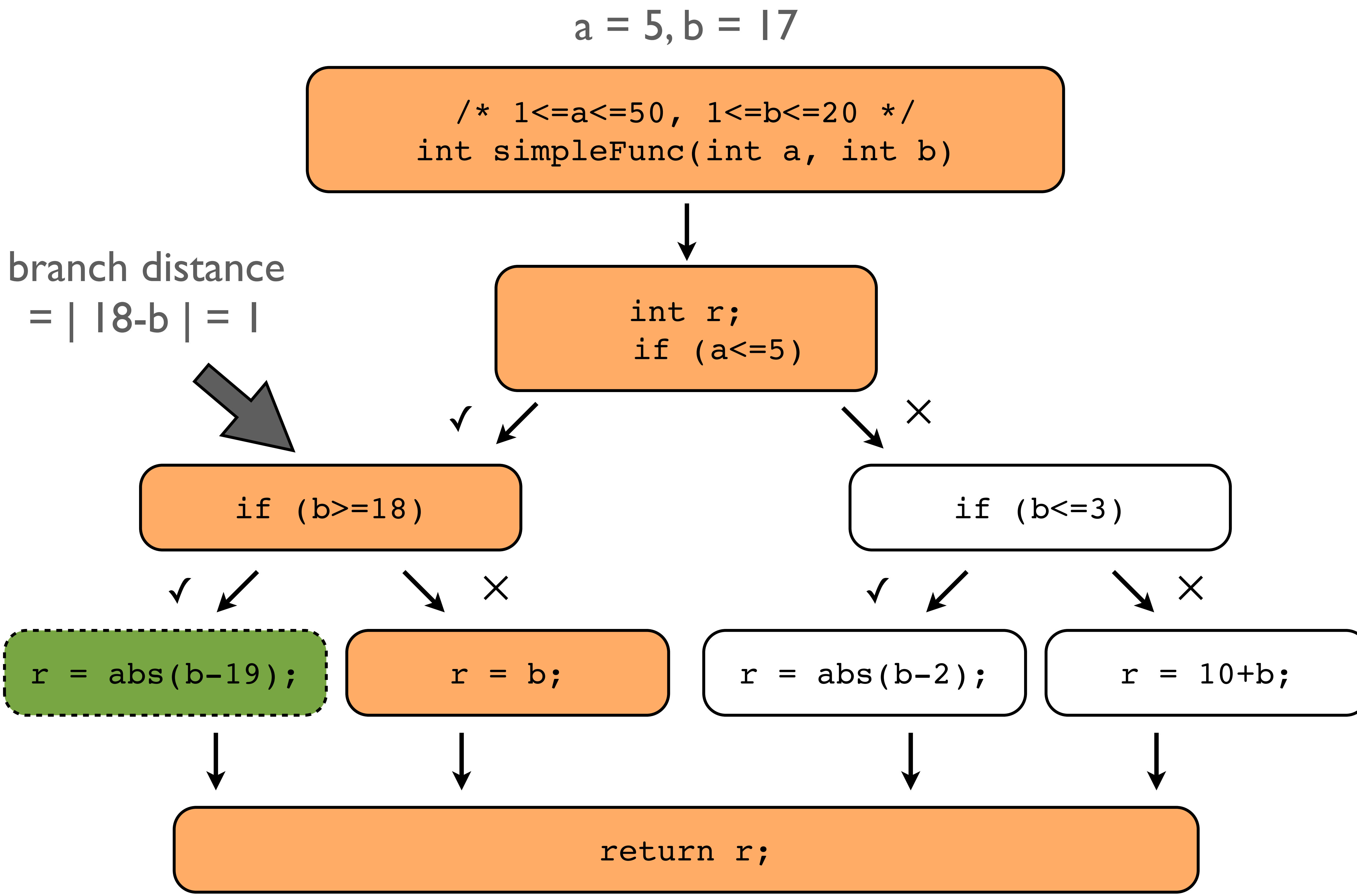
SBST - Approach Level



SBST - Branch Distance



SBST - Branch Distance



Broadening the Search in Search-Based Software Testing: It Need Not Be Evolutionary

Robert Feldt and Simon Poulding
Dept. of Software Engineering
Belkinge Institute of Technology, Karlskrona, Sweden
Email: robert.feldt@bth.se and simon.poulding@bth.se

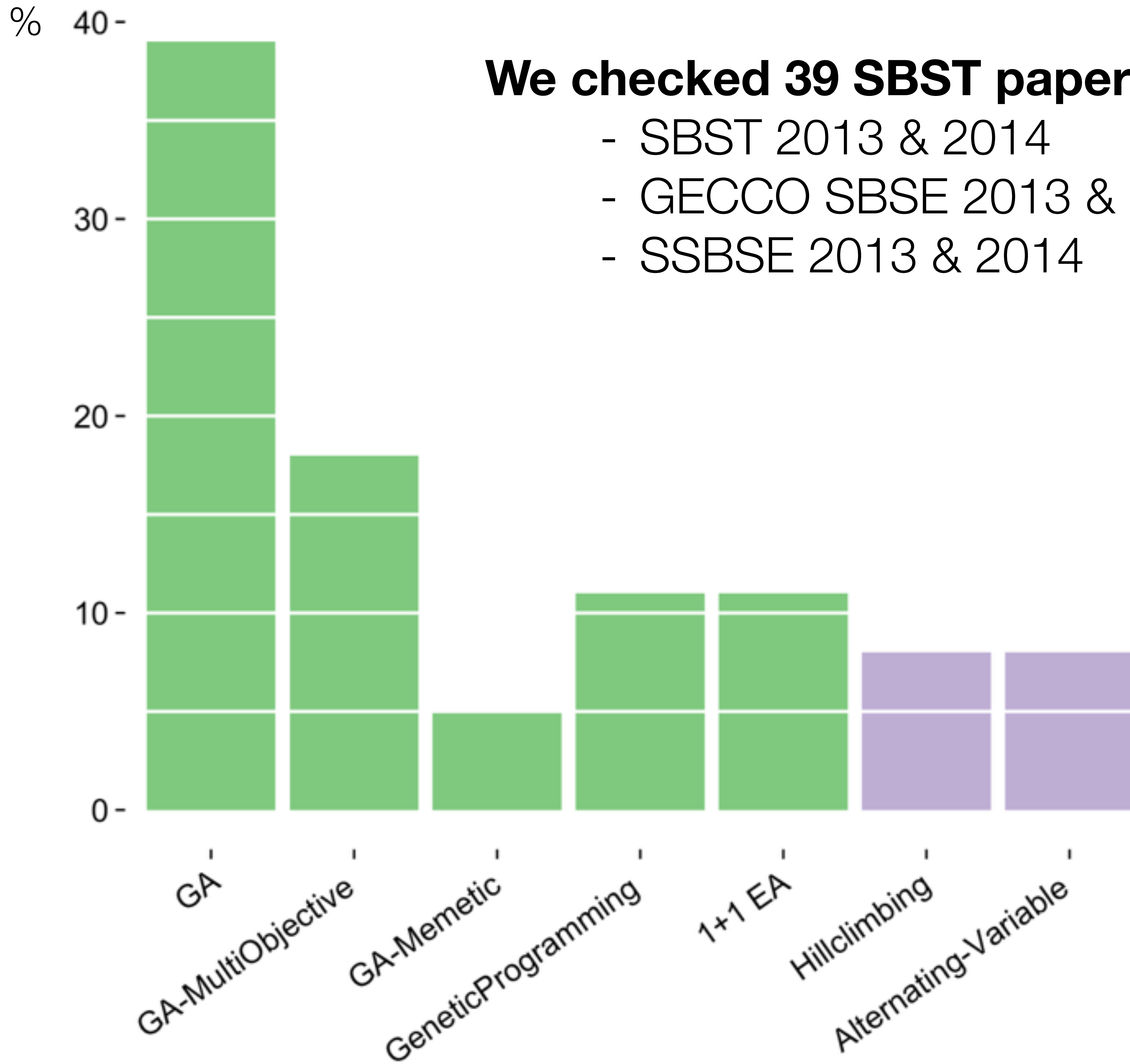
***Abstract*—Search-based software testing (SBST) can potentially help software practitioners create better test suites using less time and resources by employing powerful methods for search and optimization. However, research on SBST has typically focused on only a few search approaches and basic techniques. A majority of publications in recent years use some form of evolutionary search, typically a genetic algorithm, or, alternatively, some other optimization algorithm inspired from nature. This paper argues that SBST researchers and practitioners should not restrict themselves to a limited choice of search algorithms or approaches to optimization. To support our argument we empirically investigate three alternatives and compare them to the de facto SBST standards in regards to performance, resource efficiency and robustness on different test data generation problems: classic algorithms from the optimization literature, bayesian optimization with gaussian processes from machine learning, and nested monte carlo search from game playing / reinforcement learning. In all cases we show comparable and sometimes better performance than the current state-of-the-SBST-art. We conclude that SBST researchers should consider a more general set of solution approaches, more consider combinations and hybrid solutions and look to other areas for how to develop the field.**

I. Introduction

The term Search-Based Software Testing (SBST) describes a number of powerful methods that permit practitioners to published at these venues in 2013 and 2014. Two-thirds of the papers—26 out of 39—applied an evolutionary algorithm, of which 23 applied a Genetic Algorithm (GA): 15 as a standard GA, 2 as a GA-based memetic algorithm, and 7 as a multi-objective GA (the majority using NSGA-II)[1]. The next most-frequently applied algorithms were Genetic Programming (4 papers), (1+1) EA (4 papers), hill-climbing (3 papers), and alternating variable methods (3 papers). Our analysis suggests that evolutionary search, and GAs in particular, are the algorithms of choice for both single- and multi-objective problems in SBST.

We offer a number of explanations for this prevalence of GAs as the search technique. GAs can be applied to a wide range of problem classes and typically find solutions with acceptably good quality. This wide applicability permits us, as researchers, to re-use the knowledge gained in applying GAs to one testing problem when solving subsequent problems. In addition, there is a great deal of active research in GAs that can guide their application to testing problems, and this research is typically disseminated in a form that is readily-accessible to us. In contrast, the research on classic optimization algorithms is often described for fellow mathematicians and may be less accessible.

But EA's and GA's are used in 60-80% of papers



We checked 39 SBST papers from:

- SBST 2013 & 2014
- GECCO SBSE 2013 & 2014
- SSBSE 2013 & 2014

What is SBSE?

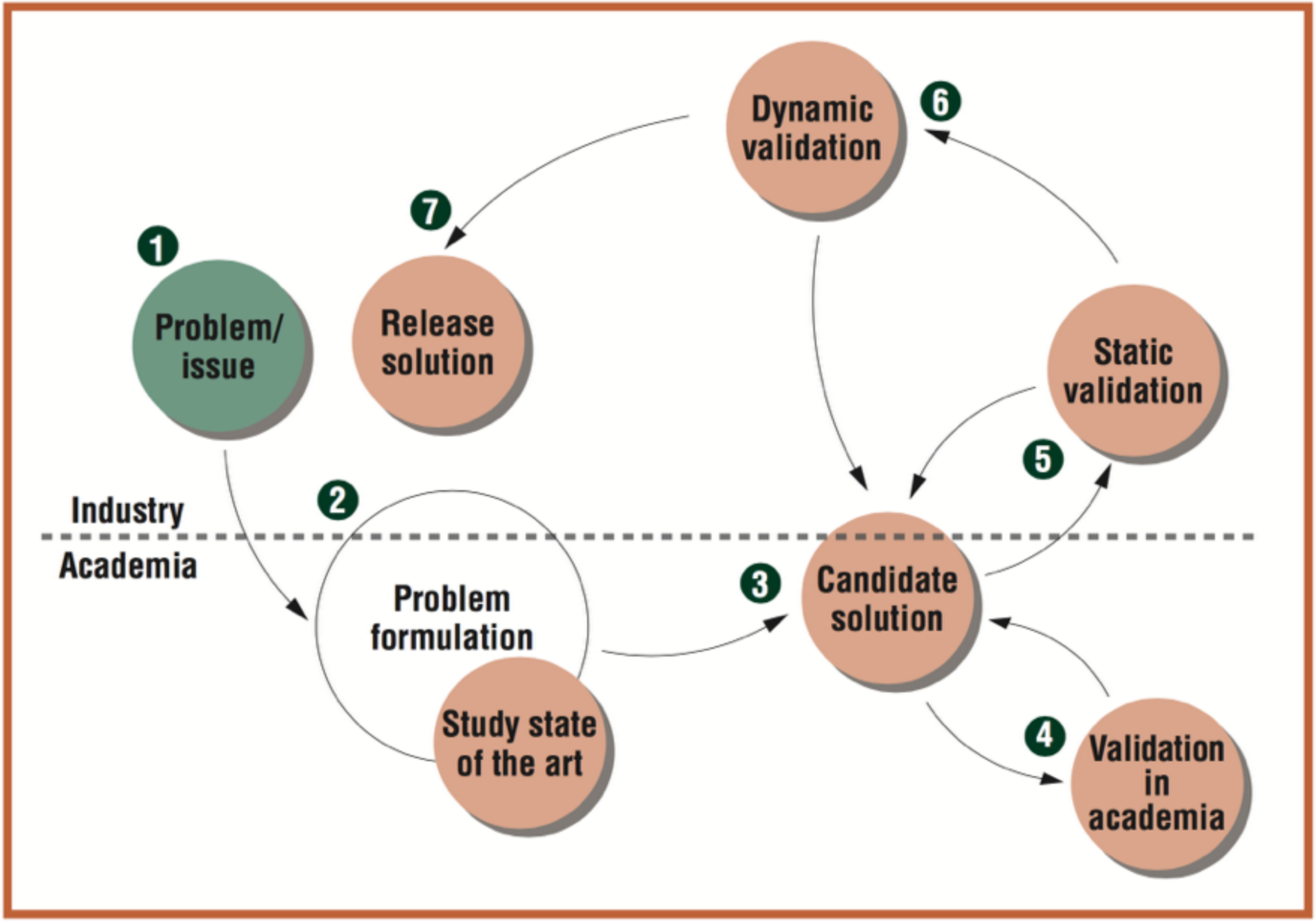
What is SBST?

Why is it not real-world (enough)?

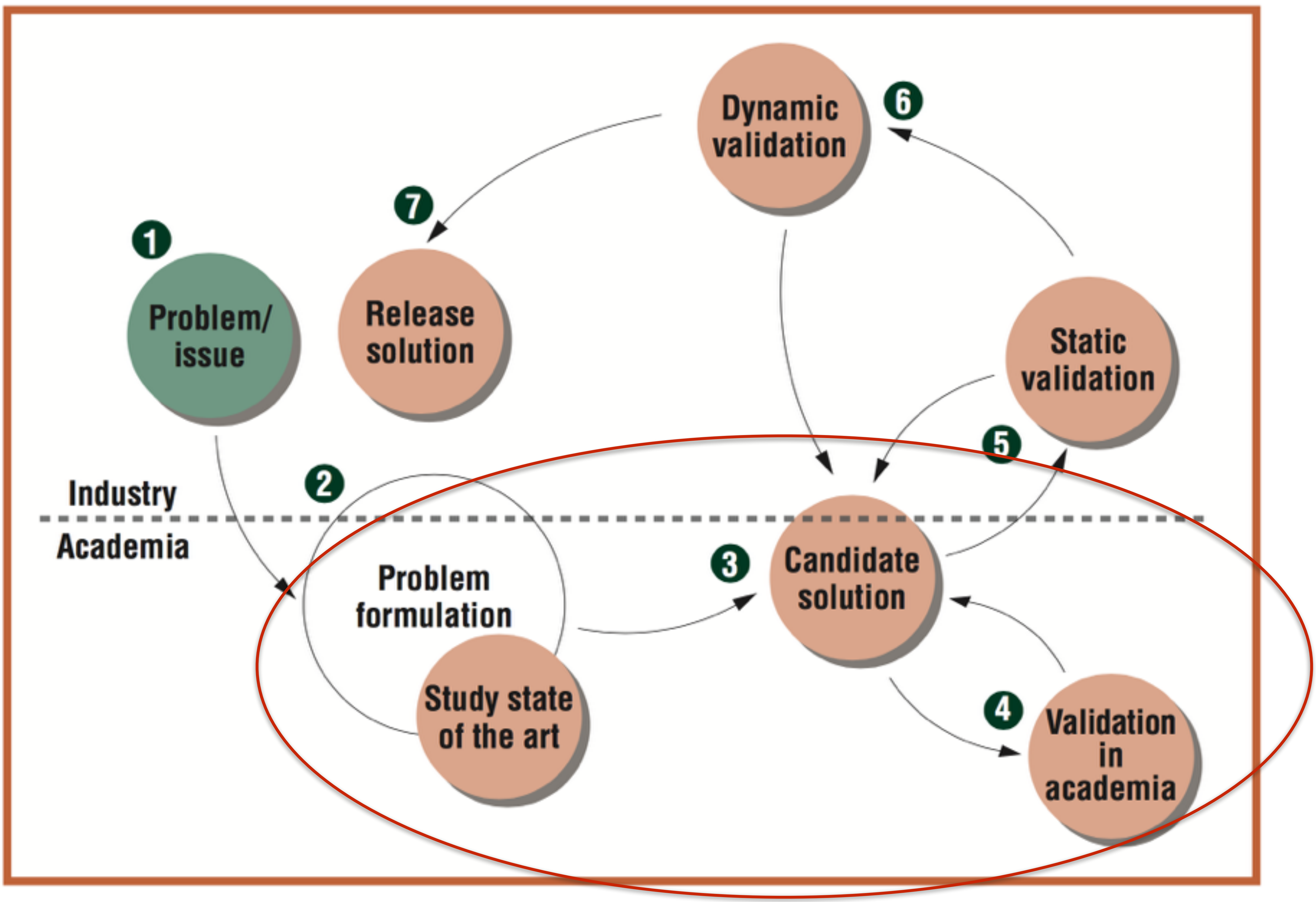
Examples of real-world applications:

- Optimizing test case selection
- Generating complex test data
- Searching for diverse test suites

Technology Transfer Model



SBST has stayed mostly within academia!



So what is SBST missing?

Application to industrial-scale systems

Not generating only numbers

Interfaces: Interaction, Visualisation

What is SBSE?

What is SBST?

Why is it not real-world (enough)?

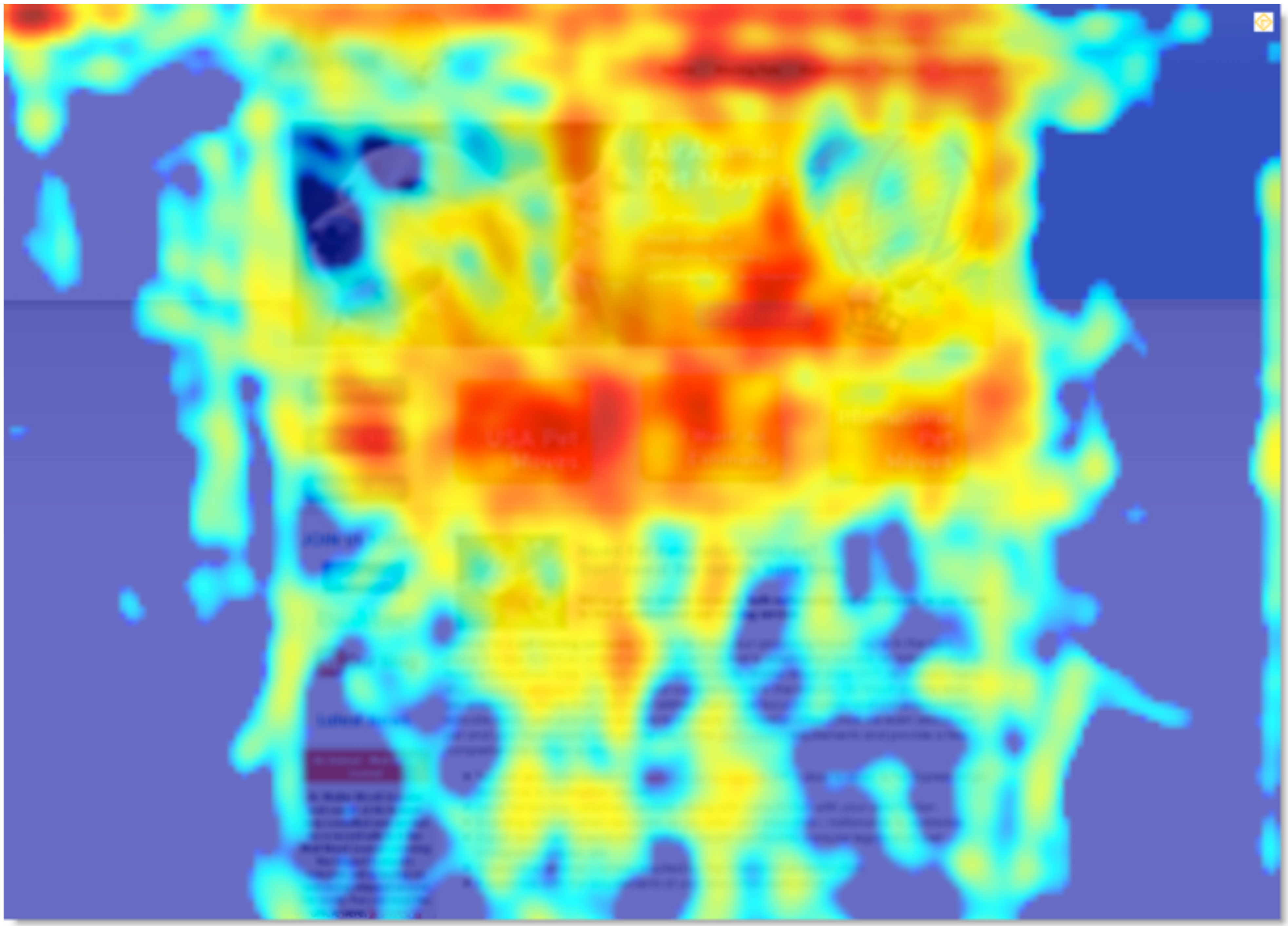
Examples of real-world applications:

- Optimizing test case selection
- Generating complex test data
- Searching for diverse test suites

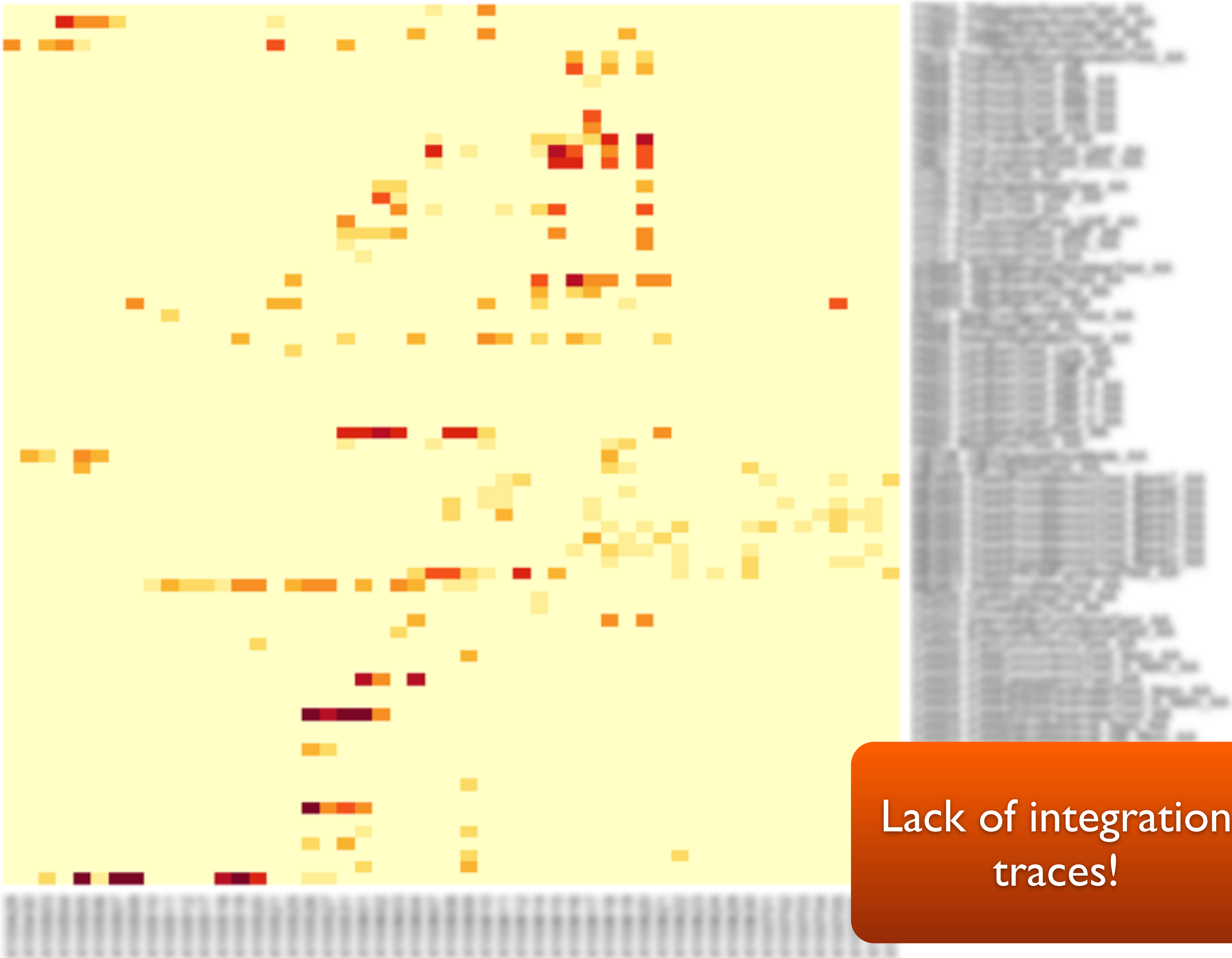
Data + Visualisation + Discussions = Insights

TEST START TIME	TEST CASE	SYSTEM VERSION	OUTCOME
2013-09-04 04:17:12	Login non existing user	1.32 - Build 3476	PASS
2013-09-04 04:17:12	Login existing user	1.32 - Build 3476	FAIL

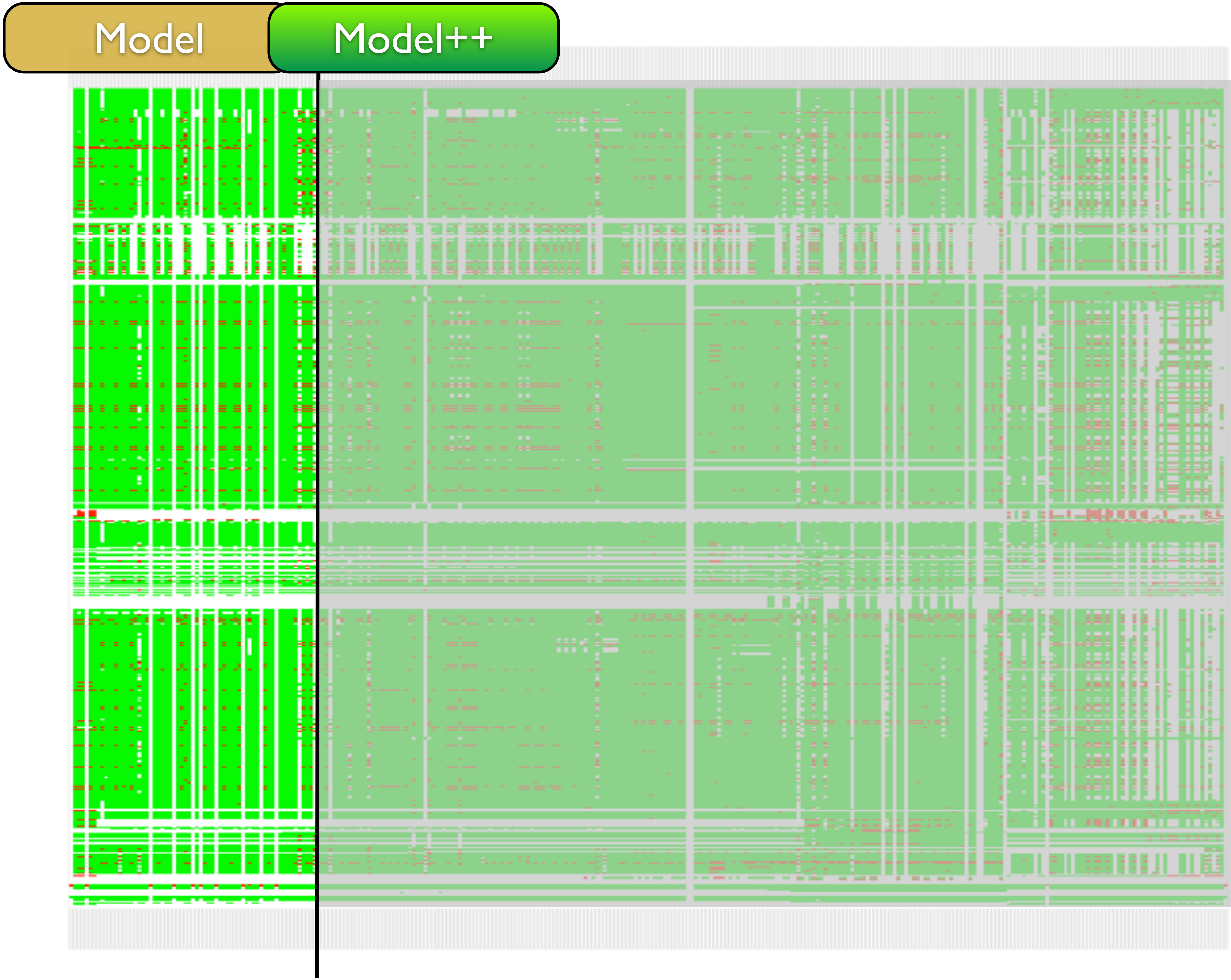
Heatmaps shows “raw” data and reveals patterns



Test failures



Lack of integration traces!


Model
Model++

Most RegTest **research not** realistic

Requires too much (or complete) information

- Source & Test code, Changes, Outcome,
  Bug reports, Priorities, Severities, ...

Focus on only one criteria (Mono-objective)

- In companies, things change & situations differ

Evaluated on very small examples

- Scalability problems don't surface
- Unclear if relevant for you

But simple modeling often give BIG benefits!

3 Types of Regression Test Selection

1. TCS = “**Which n** test cases should I run given **P2!=P1**?”
2. TSM = “**Minimal** test suite that gives **adequate testing**?”
3. TCP = “**Priority order** of test cases to find faults early?”

robert.feldt@gmail.com

History-based Prioritization

"Use history of test outcomes to focus testing where most needed"

Example, Fazlilazadeh (2009):

Priority

Fails

Prev. Priority

Sessions since last run

$$PR_k = \alpha * \frac{f_{ck}}{e_{ck}} + \beta * PR_{k-1} + \gamma * h_k$$

This session

$$0 \leq \alpha, \beta, \gamma \leq 1, k \geq 1$$

Runs

robert.feldt@gmail.com

How to optimize history-based reg testing?

robert.feldt@gmail.com

Optimizing Fazlilazadeh

Priorit

Fails

Prev. Priority

Sessions since last run

$$PR_k = \alpha * \frac{f_{ck}}{e_{ck}} + \beta * PR_{k-1} + \gamma * h_k$$

Next session

$$0 \leq \boxed{\alpha, \beta, \gamma} \leq 1, k \geq 1$$

Runs

Now: Find optimal constants

In our previous work:

	TCSR	FDR
0.3, 0.3, 0.3	60 %	86 %
0.9, 0.05, 0.1	60 %	94 %
Random sel.	60 %	59 %

Which constants give best FDR?

	Optimization	α, β, γ	TCSR	FDR
old	None, standard	Faz(0.3, 0.3, 0.3)	60 %	86 %
old	Manual testing	Faz(0.9, 0.05, 0.1)	60 %	94 %
baseline	N/A	Random sel.	60 %	59 %
new1	Simulated Annealing	None better!	60 %	94 %

Faz is robust when many test cases are selected!
(Less space for improvement)

robert.feldt@gmail.com

Is Faz robust for different TCSR's?

	Optimization	α, β, γ	**TCSR**	**FDR**
old	None, standard	Faz(0.3, 0.3, 0.3)	20 %	10 %
old	Manual testing	Faz(0.9, 0.05, 0.1)	20 %	57 %
baseline	NAP	Random sel.	20 %	20 %
new1	Simulated annealing	Faz(0.18, 0.47, -0.08)	20 %	89 %

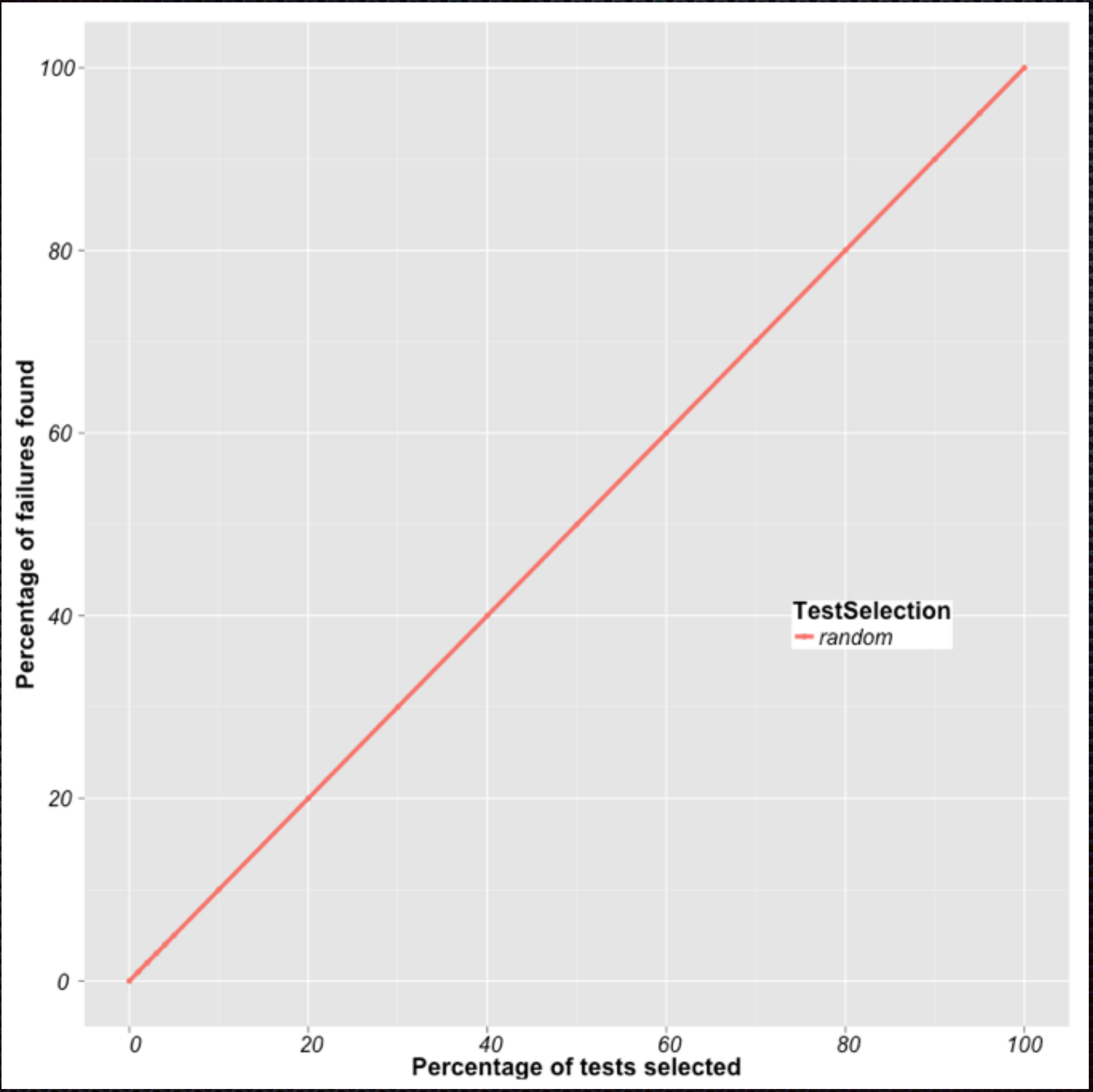
Faz not robust when selecting few test cases!
Large variation between param settings

robert.feldt@gmail.com

How much can we gain?

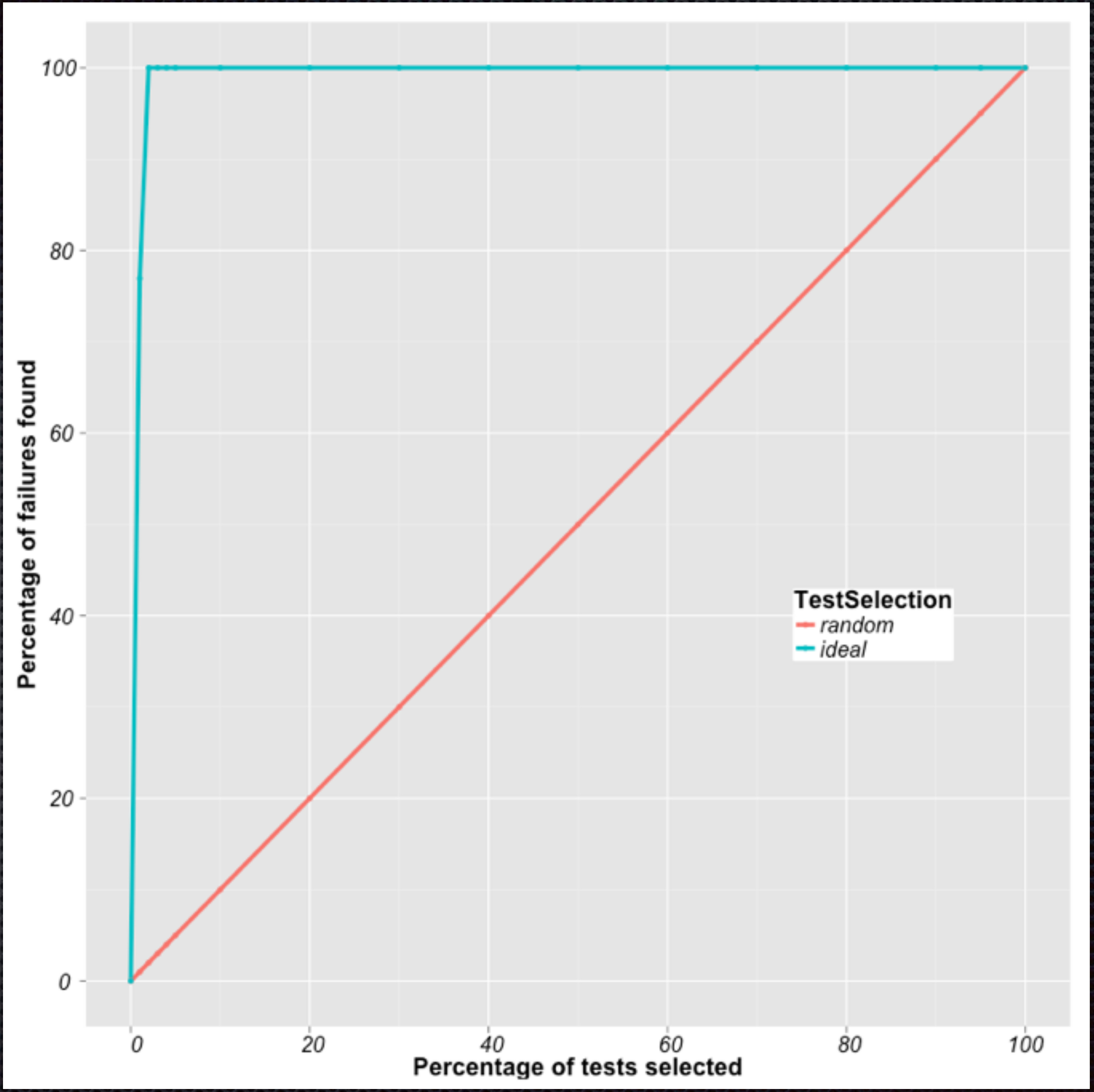
robert.feldt@gmail.com

How much can we gain? (FDE-S curves)



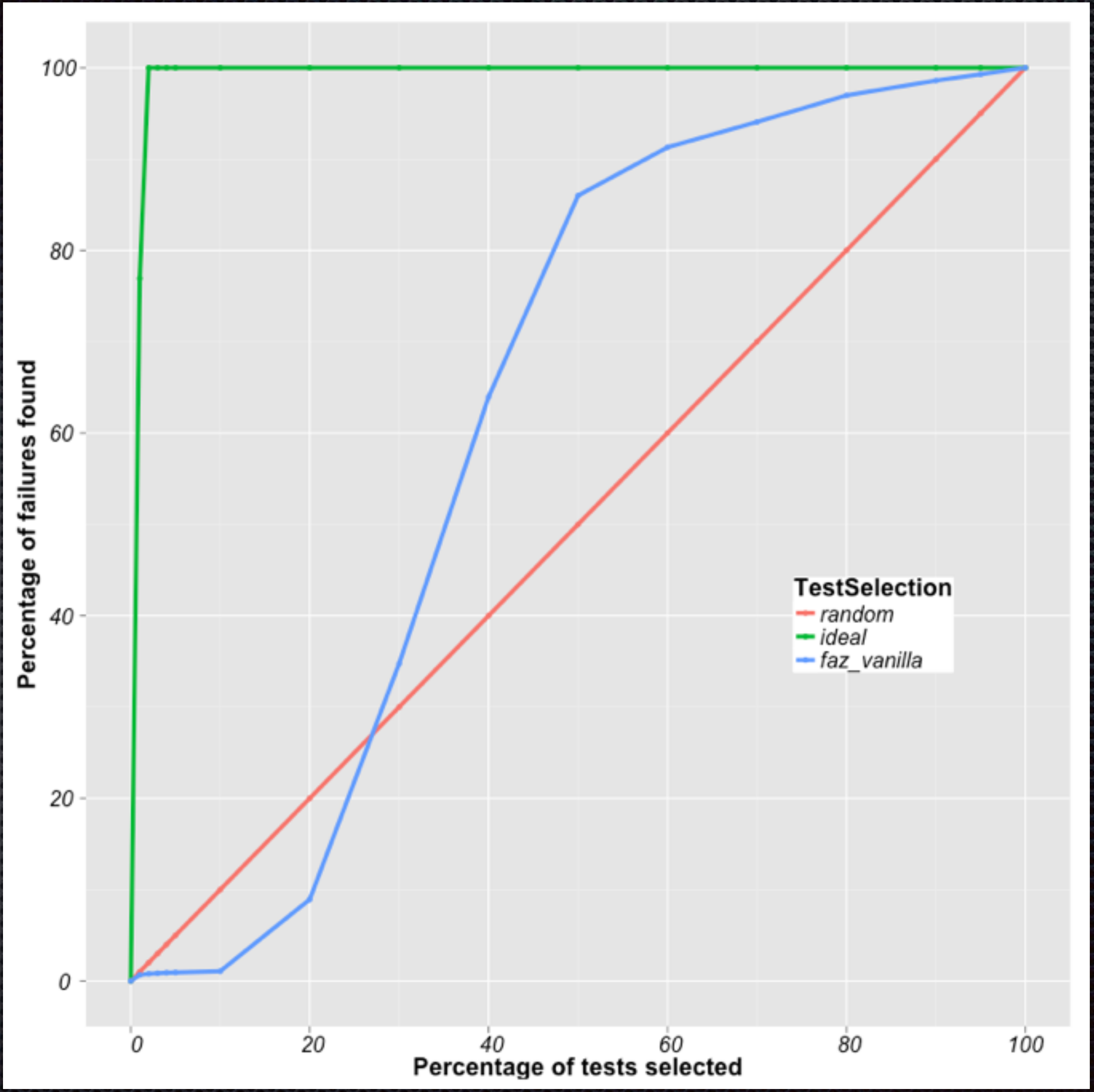
robert.feldt@gmail.com

How much can we gain?



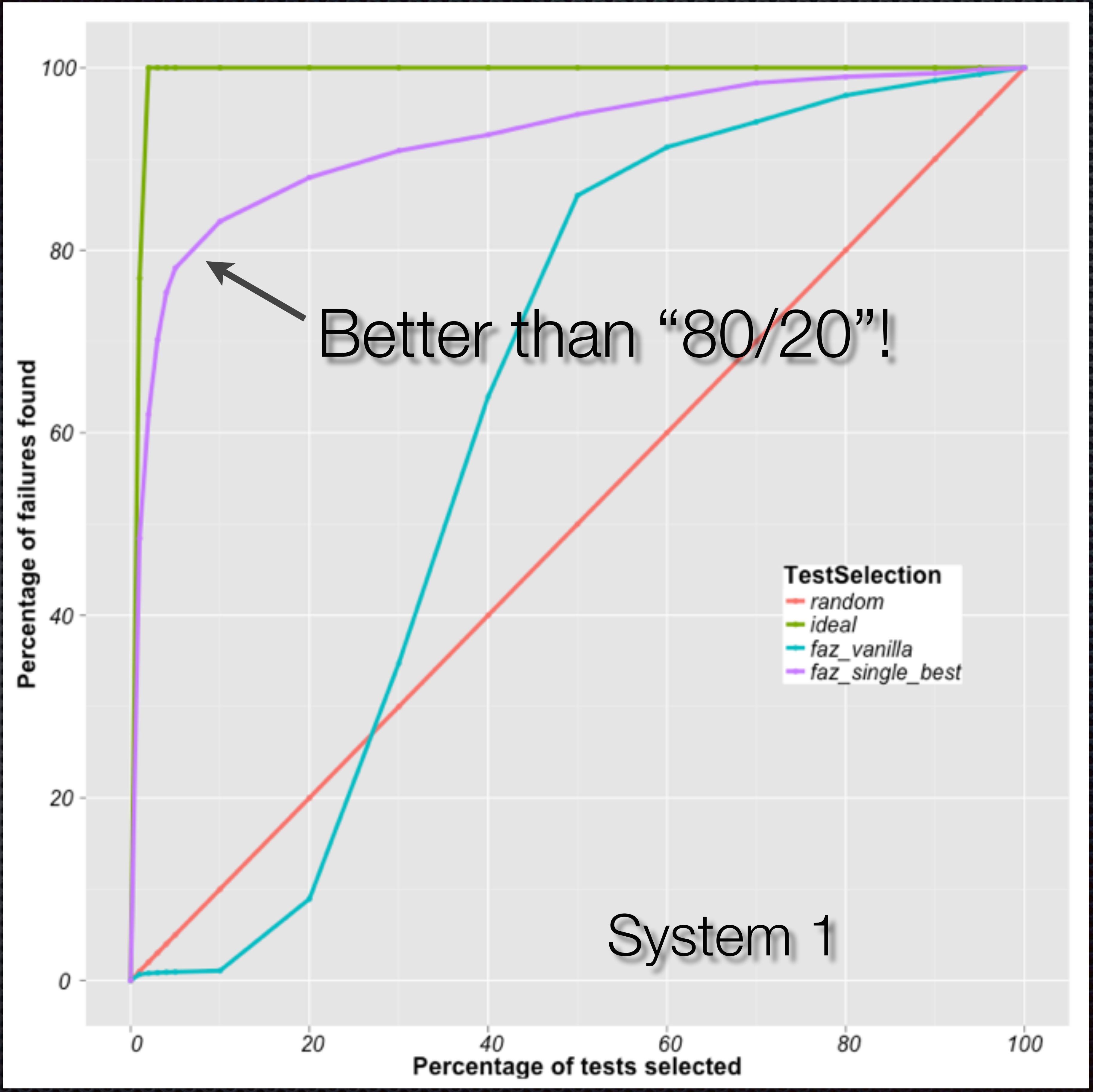
robert.feldt@gmail.com

How much can we gain?

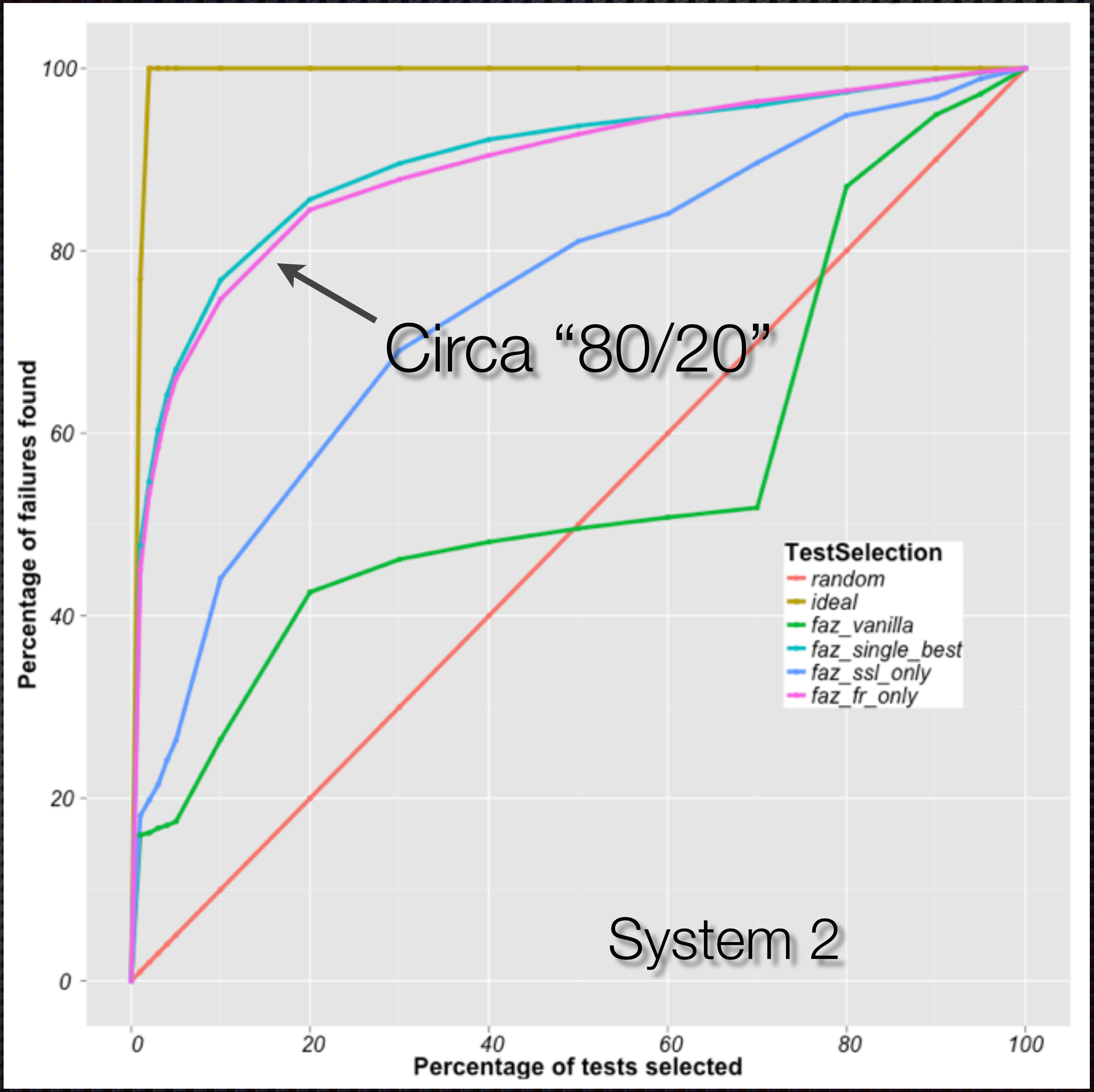


robert.feldt@gmail.com

How much can we gain?



Are the result robust (for other system)?



robert.feldt@gmail.com

Can we be sure this will work?

No, it will work if there are clear patterns in testing

- For less regular systems we need more data.
  The next natural step is to add info about source code changes

In [Wikstrand2009] we showed a simple file-based technique

- Keeps a cache of when source code files were changed

This is one way we can extend the model if not good enough.

Combined method for test prioritization

If Top-10/20% selected, ~80-90% failures caught

“But want to be sure every test run every 4th test”

Add 20% more from prio list that not executed in last 4 tests

Ensures all tests executed at least every 4th full test but reduces total number of test runs <u>-60%</u>

Many different trade-offs/solutions possible once optimization framework in place!

robert.feldt@gmail.com

So what is the best way forward?

If someone forces me to make a recommendation today I would suggest:

1. Prioritize test cases on their historical failure rate (number of failures / number of executions)

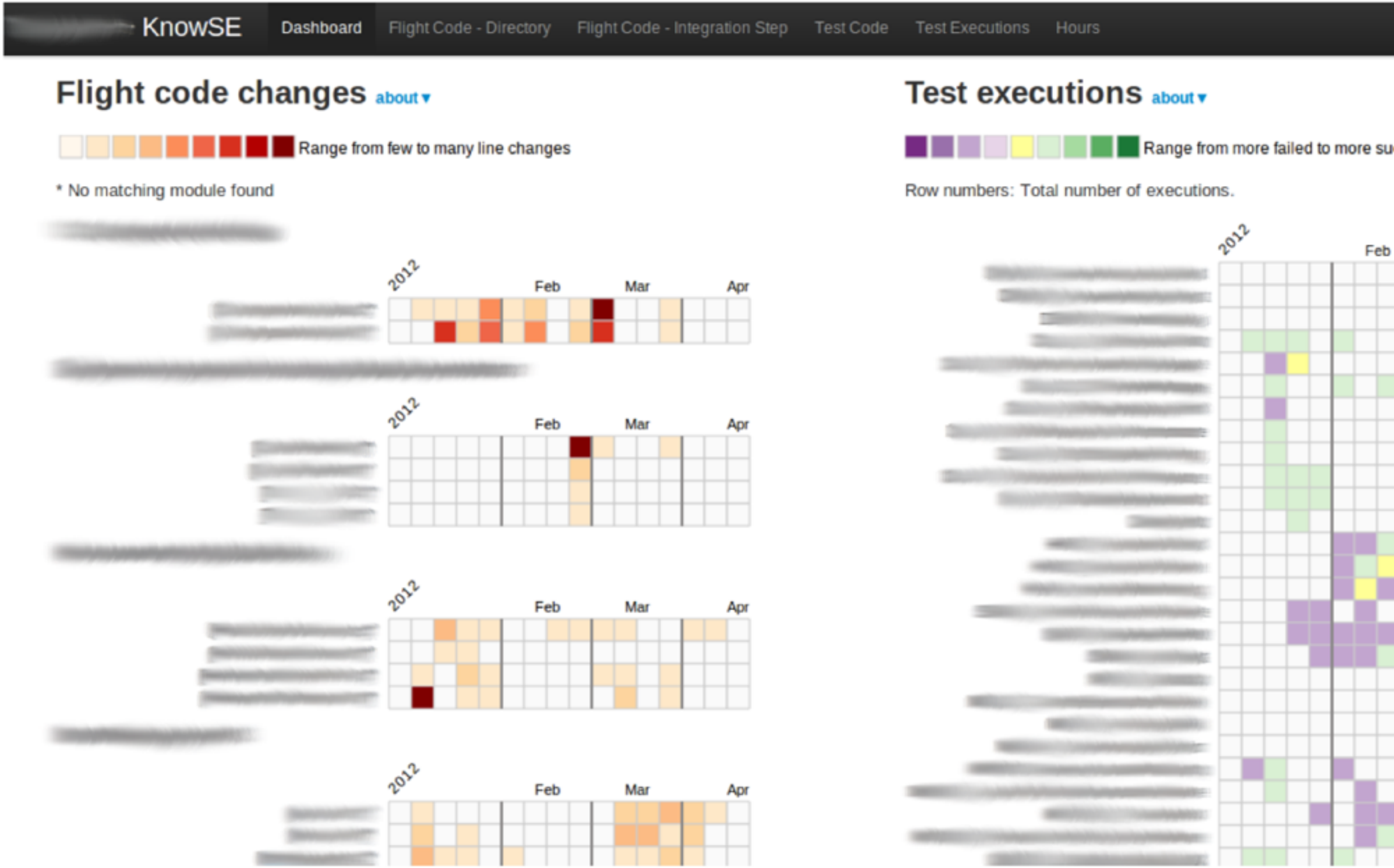
2. Refine the priorities based on source code file changes since last test run

3. Add the time since last execution to prioritization

For all three "levels" use simple optimization to adapt to the project

robert.feldt@gmail.com

Taking it **Online**



What is SBSE?

What is SBST?

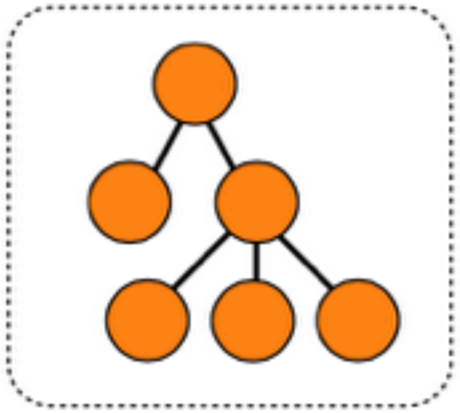
Why is it not real-world (enough)?

Examples of real-world applications:

- Optimizing test case selection
- Generating complex test data
- Searching for diverse test suites

Highly-Structured Test Data

Some types of software take highly-structured inputs that must satisfy often complex constraints



data indexing systems

```
int main( ) {
  printf(“Hello”);
}
```

gcc

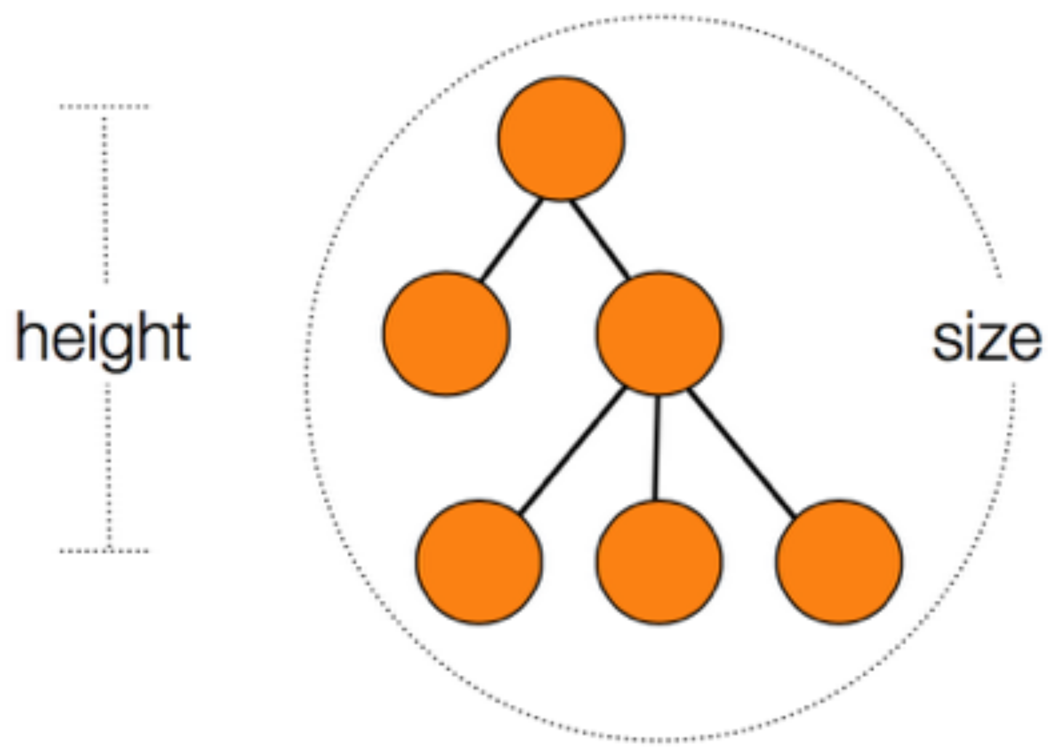
compilers

```
<html>
  <body>
    <h1>A</h1>
  </body>
</html>
```

HTML rendering engines

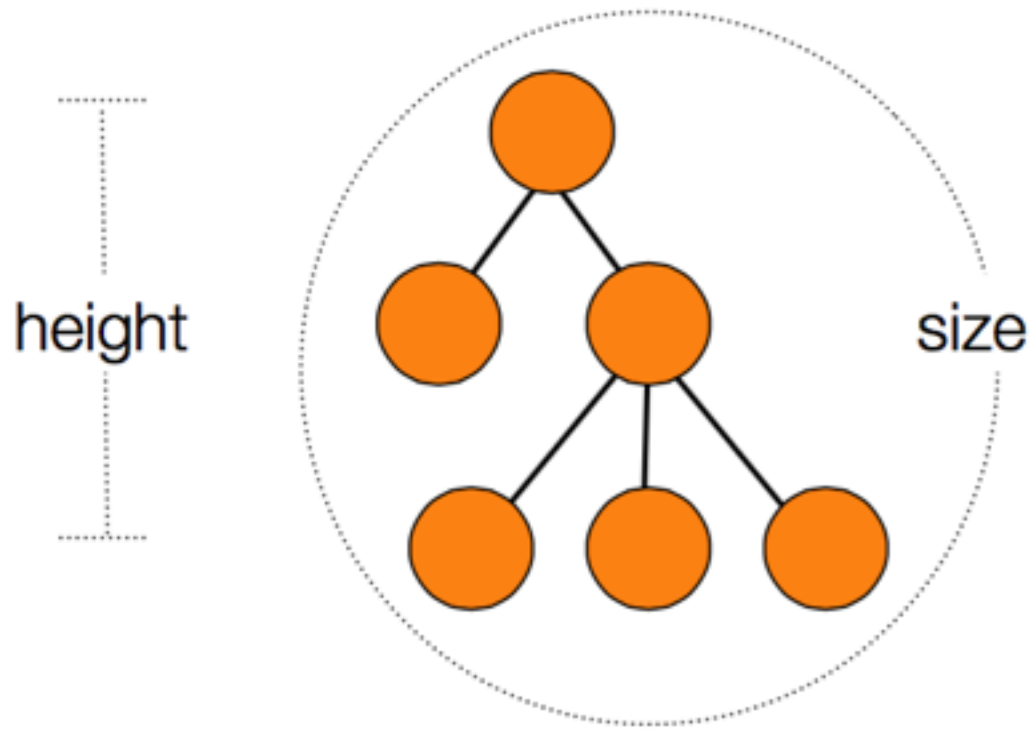
Bias Objectives

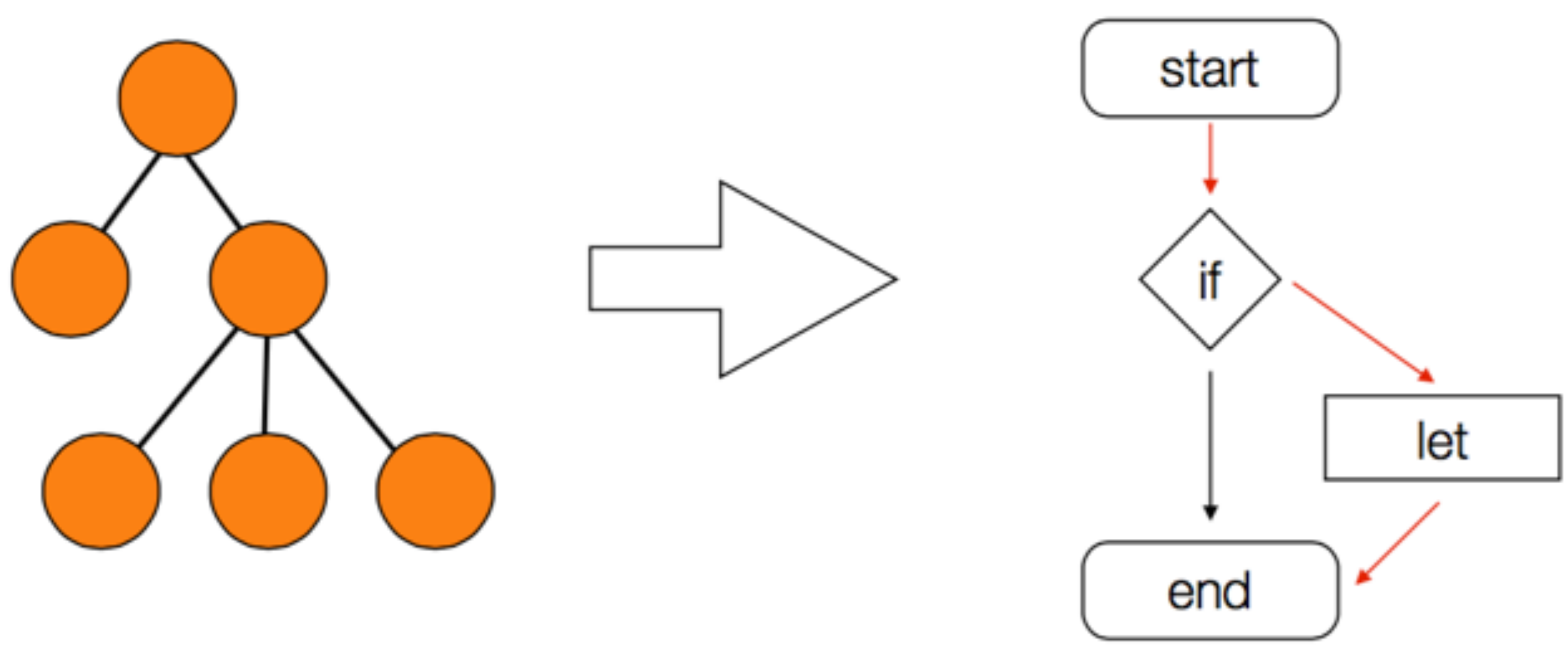
Effective testing may require the structured data to have specific intrinsic or extrinsic properties



Bias Objectives

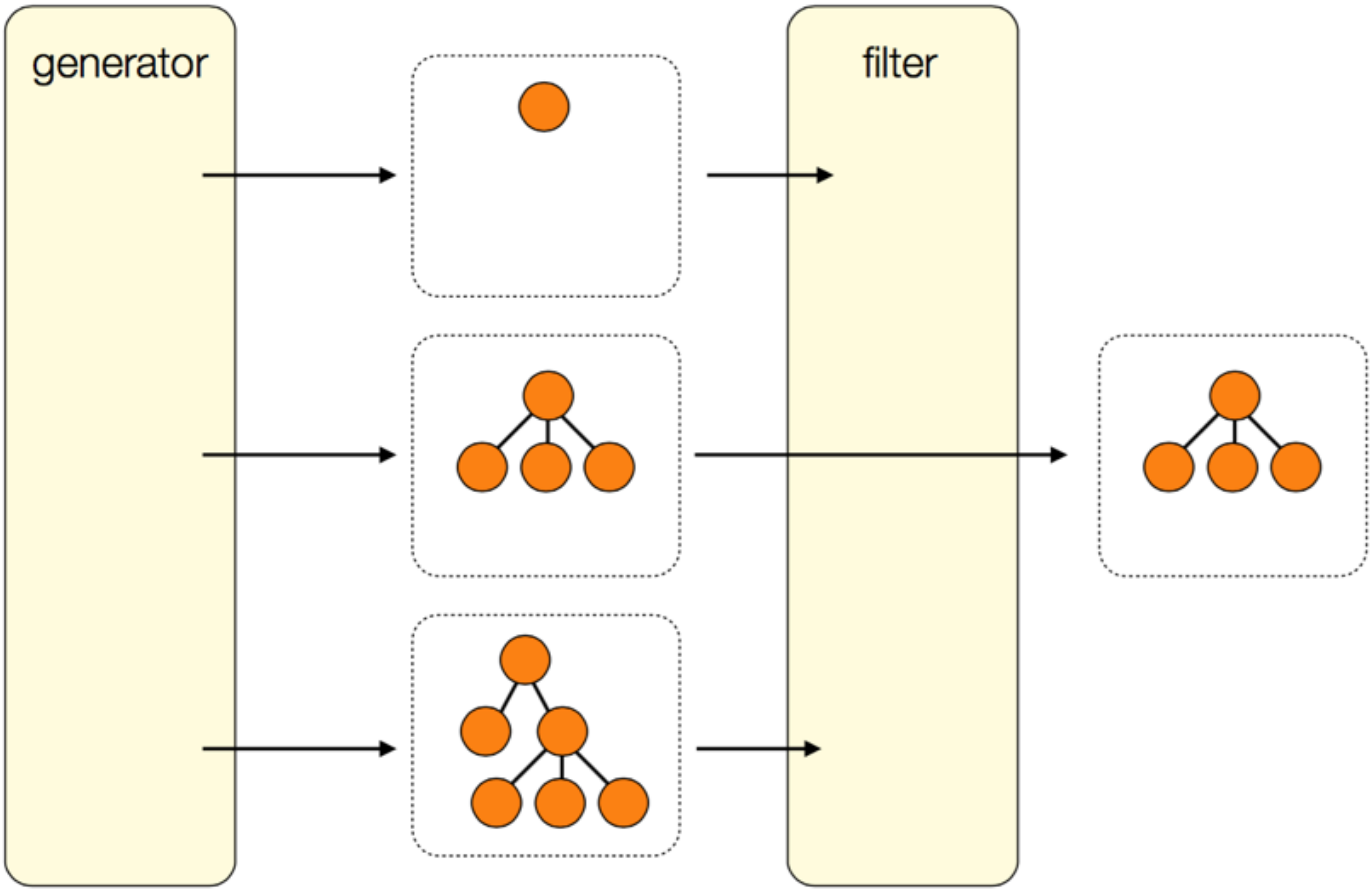
Effective testing may require the structured data to have specific intrinsic or extrinsic properties





Generation and Filtering

Bias objectives are typically met using a generator that builds test data with properties close to the desired values, often supplemented by an exact filter



Boltzmann Samplers

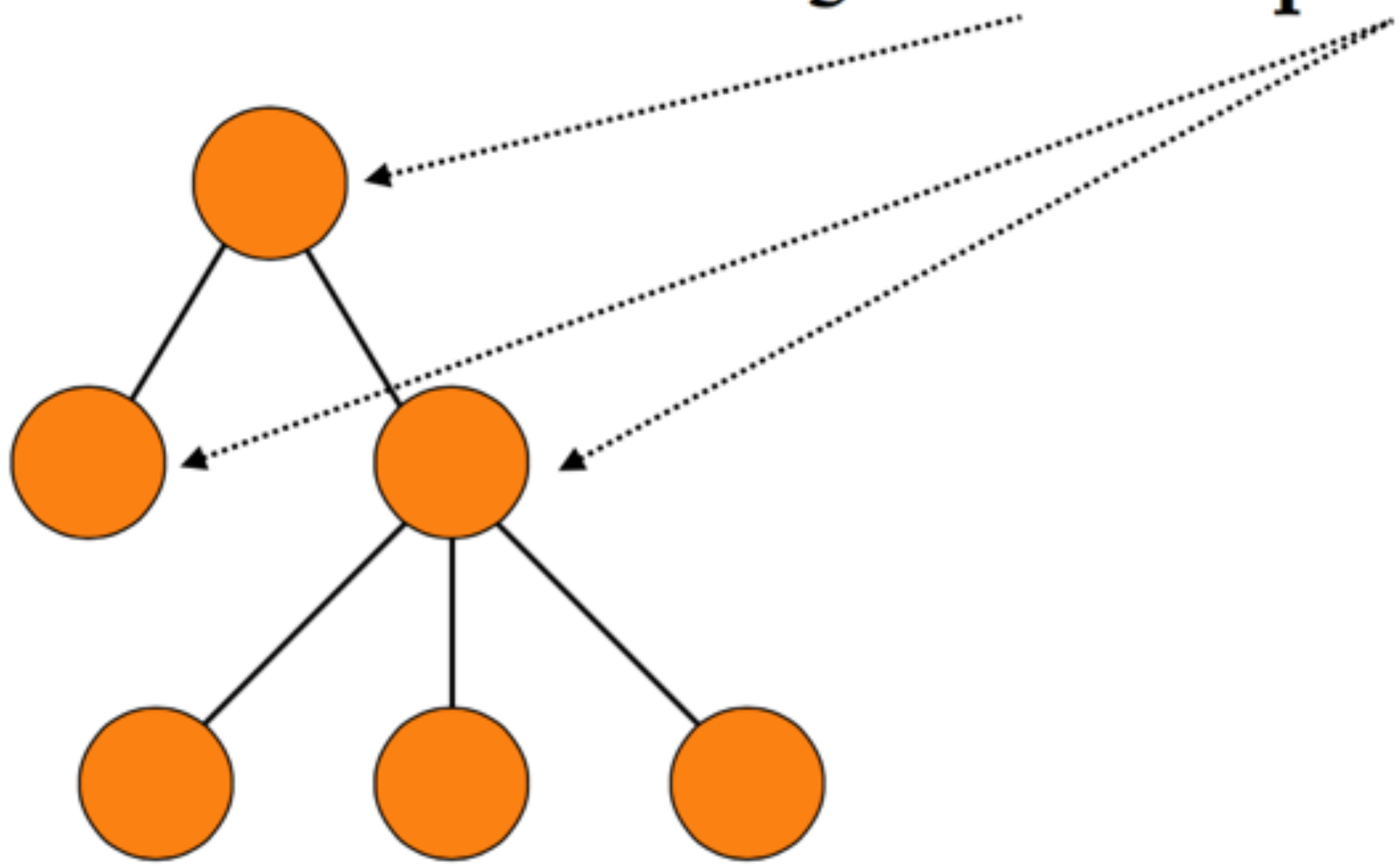
A simple grammar describes how to generate test data using elementary operations that combine simpler objects

$$\mathcal{G} = \mathcal{Z} \cdot \mathbf{sequence}(\mathcal{G})$$

Boltzmann Samplers

A simple grammar describes how to generate test data using elementary operations that combine simpler objects

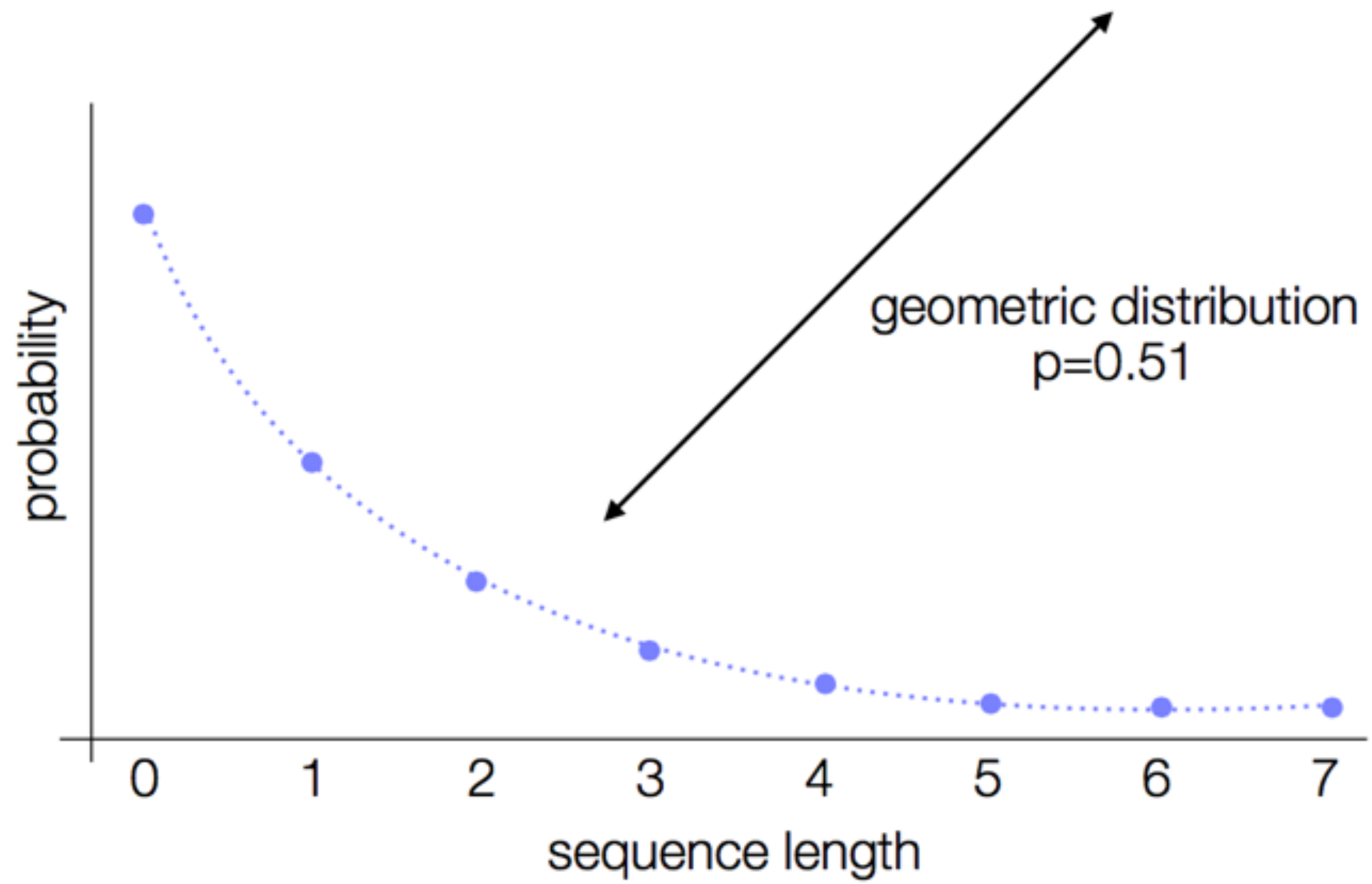
$$\mathcal{G} = \mathcal{Z} \cdot \mathbf{sequence}(\mathcal{G})$$



Boltzmann Samplers

Mathematically tractable: it is relatively easy to calculate the local distributions at operators that produce a given mean tree size. But are limited to specific structures and only for the property of size.

$$\mathcal{G} = \mathcal{Z} \cdot \mathbf{sequence}(\mathcal{G})$$



Stochastic Grammars

More flexible than Boltzmann samplers, but in general there is no analytical method for setting production weights to achieve bias objectives

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{GeneralTree} \\
\mathrm{GeneralTree} &\rightarrow \mathrm{nodeLabel\ SubTrees} \\
\mathrm{SubTrees} &\rightarrow \epsilon \mid \mathrm{GeneralTree\ SubTrees}
\end{aligned}
$$

Non-Deterministic Programs

More flexible than grammars, but again there is generally no analytical methods of tuning non-determinism to achieve bias objectives

```
data Tree = Node Int [Tree] deriving (Eq, Show, Ord)

instance Arbitrary Tree where
  arbitrary = sized tree'
  where tree' n = liftM2 Node arbitrary (
    resize (n-1) (listOf' arbitrary))

listOf' gen = sized $ \n ->
  do k <- choose (0,n)
    if k == 0
      then vectorOf 0 gen
      else vectorOf k (resize ((n+k-1) `div` k) gen)
```

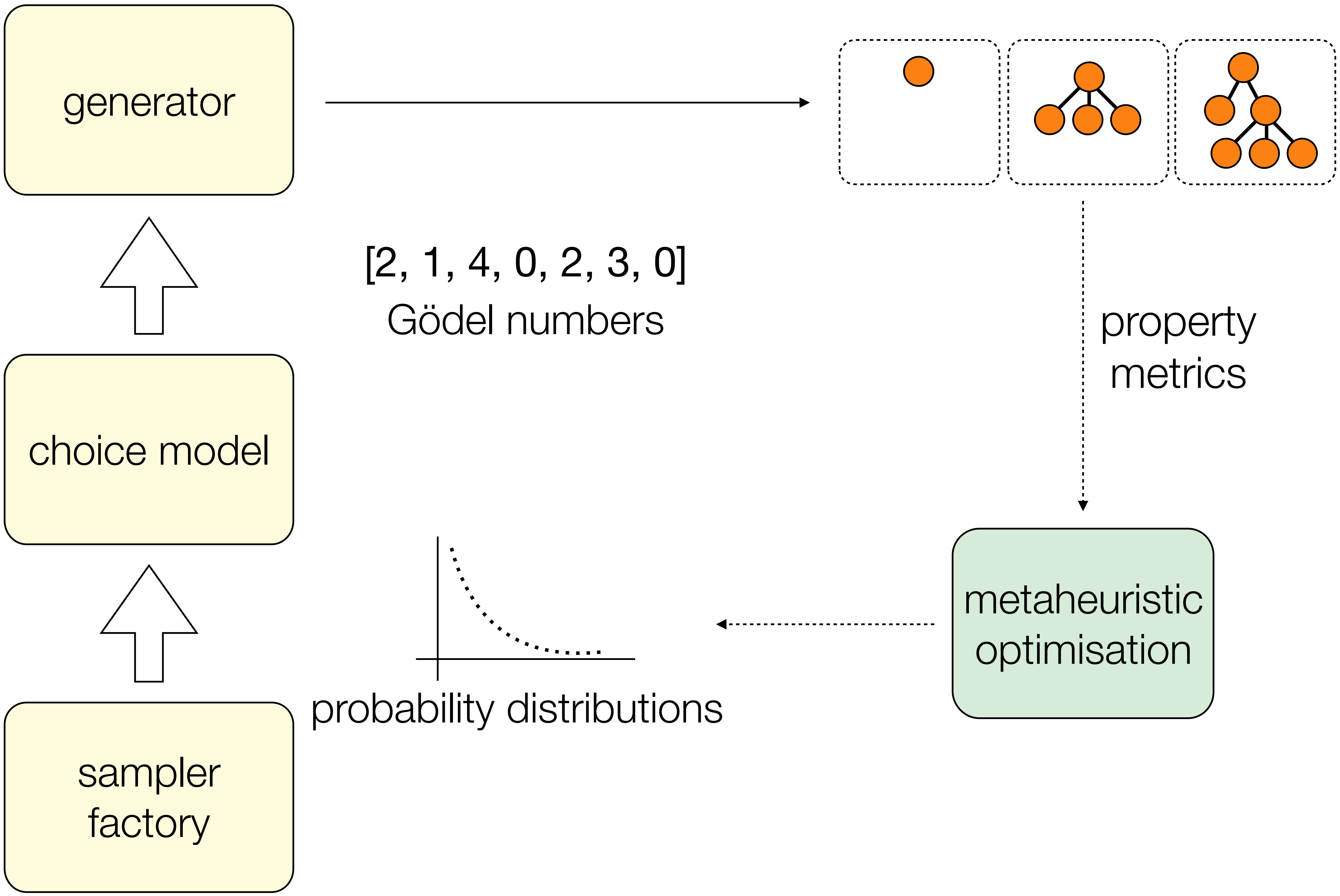
Objective

Require a technique:

- has the flexibility of non-deterministic programs
- can bias the generation to any property (not just size)
- can automatically tune the generator to achieve these biases

GödelTest Framework

Extracts a model of choice points from a non-deterministic generator; optimises the choice model using metaheuristic optimisation to met bias objectives



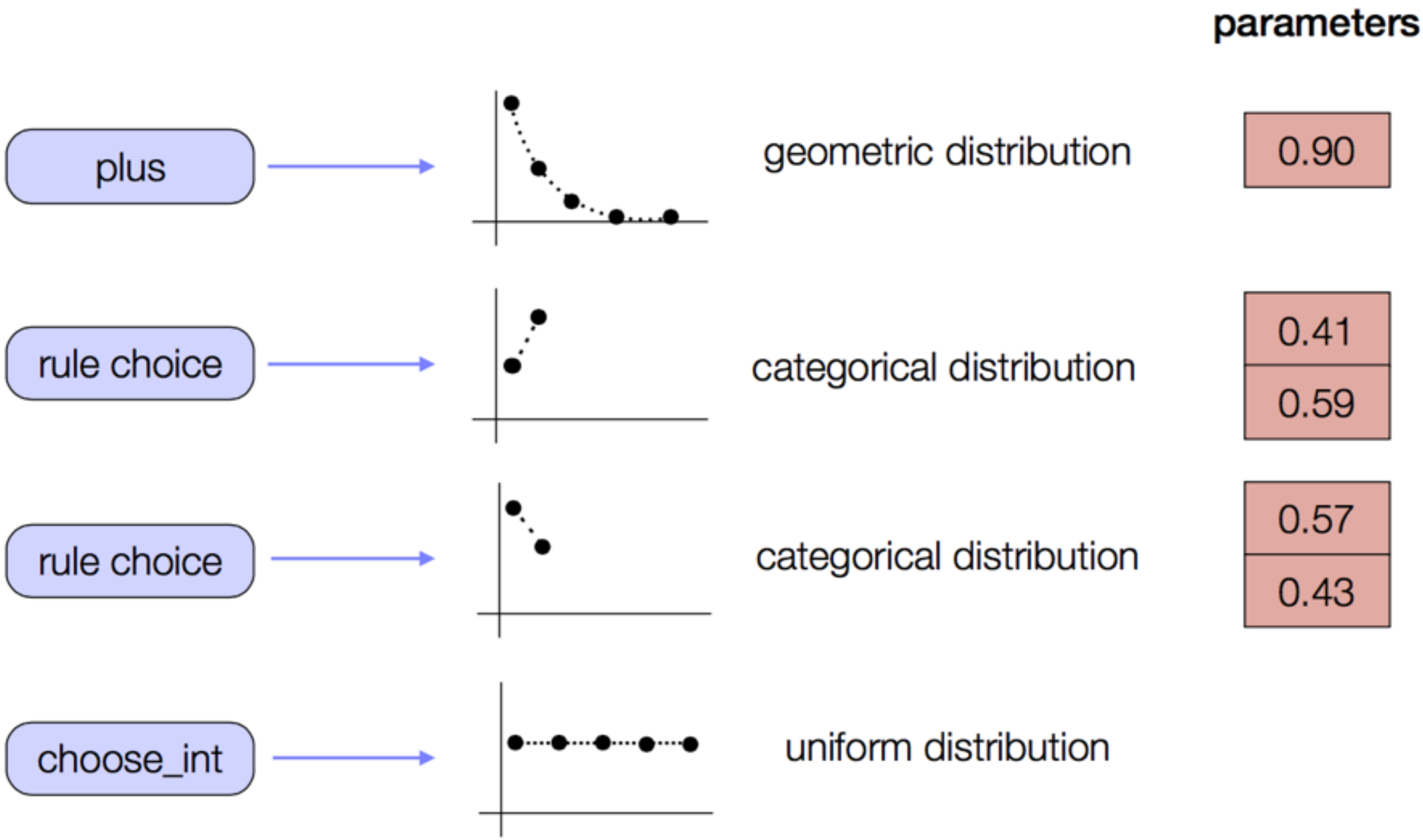
Generators

A DSL in Julia for constructing well-formed data;
non-determinism arises from a small set of implicit
and explicit choice points

```
# recursive generator for arithmetic expressions
@generator RecursiveExprGen begin
  start = expression
  expression = operand * " " * operator * " " * operand
  operand = number
  operand = "(" * expression * ")"
  number = (choose(Bool) ? "-" : "") * join(plus(digit))
  digit = string(choose(Int,0,9))
  operator = "+"
  operator = "-"
  operator = "/"
  operator = "*"
end
```

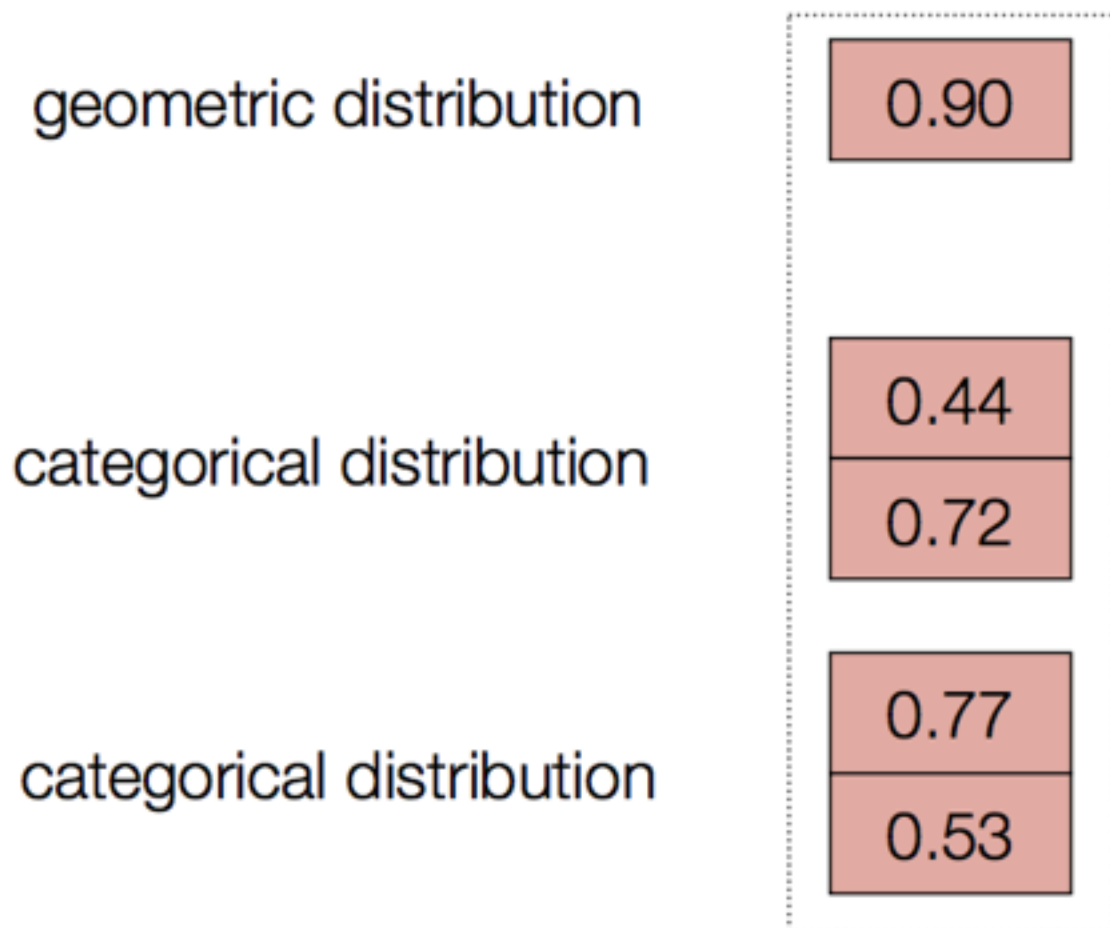
Sampler Factory

A sampler factory associates appropriate local distribution (samplers) to each choice point in the model



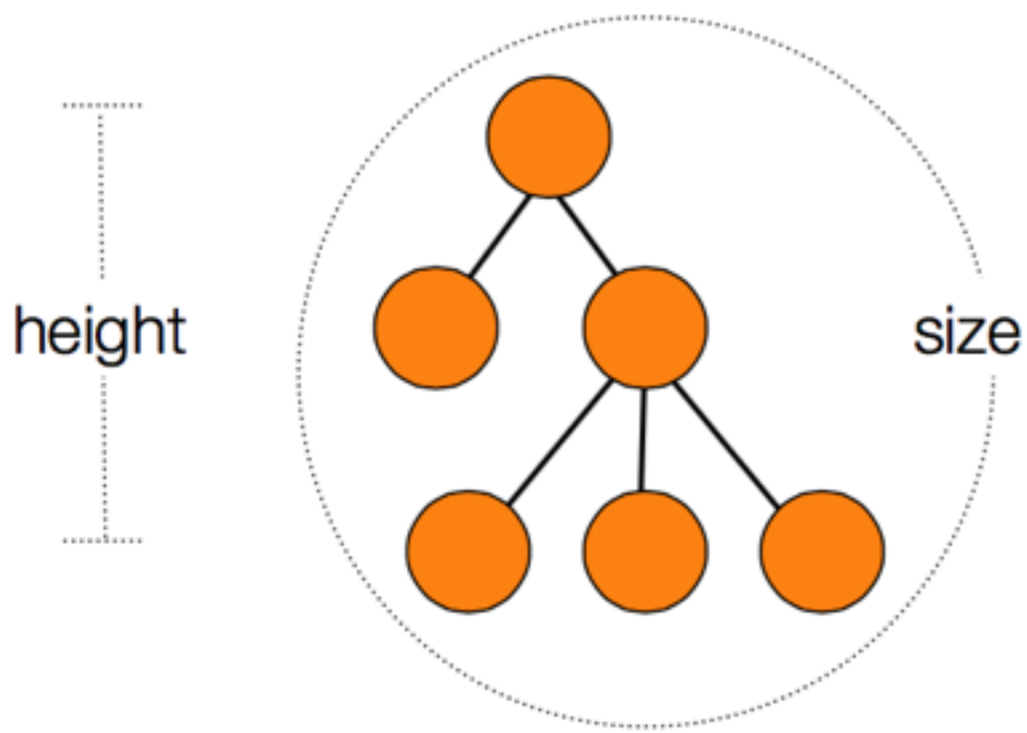
Metaheuristic Search

Metaheuristic search acts on the set of sampler parameters in order to modify the local distribution of Gödel numbers associated with each choice points



Problem

Two target bias objectives specified in terms of tree size and height

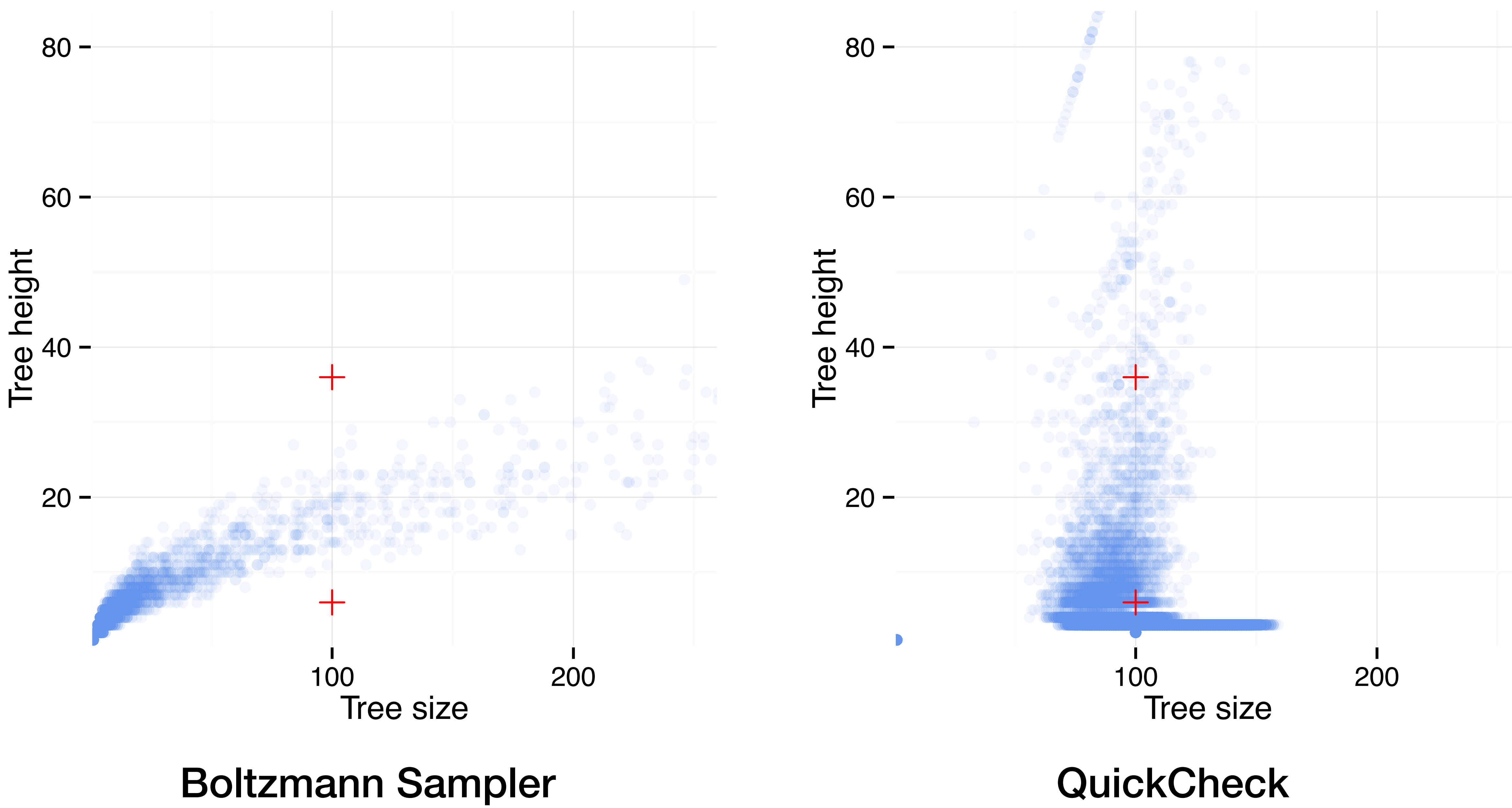


target 1: size = 100 AND height = 36

target 2: size = 100 AND height = 6

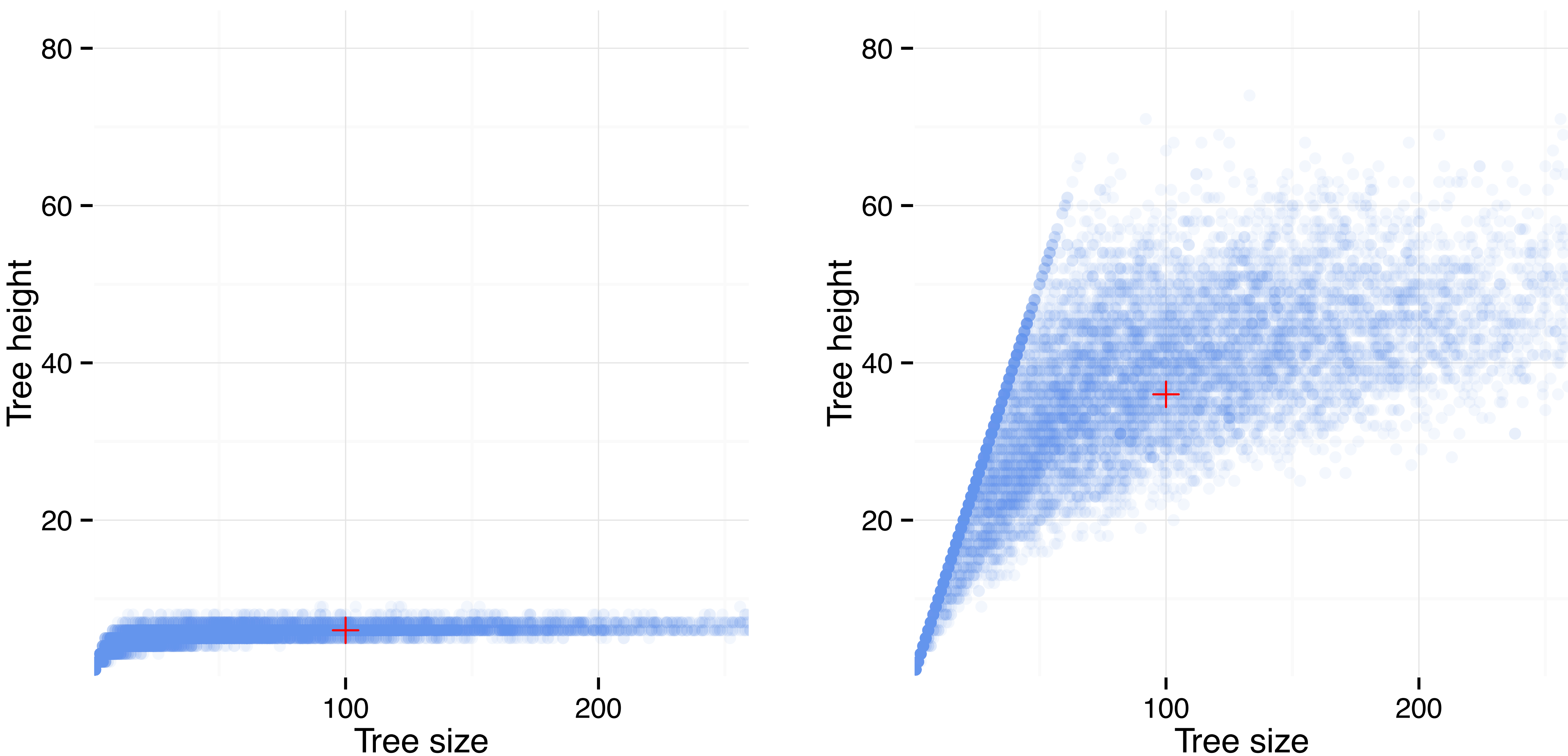
Results

Scatter plots show the distribution of tree sizes and heights; target bias objectives are indicated by crosses



Results

Scatter plots show the distribution of tree sizes and heights; target bias objectives are indicated by crosses

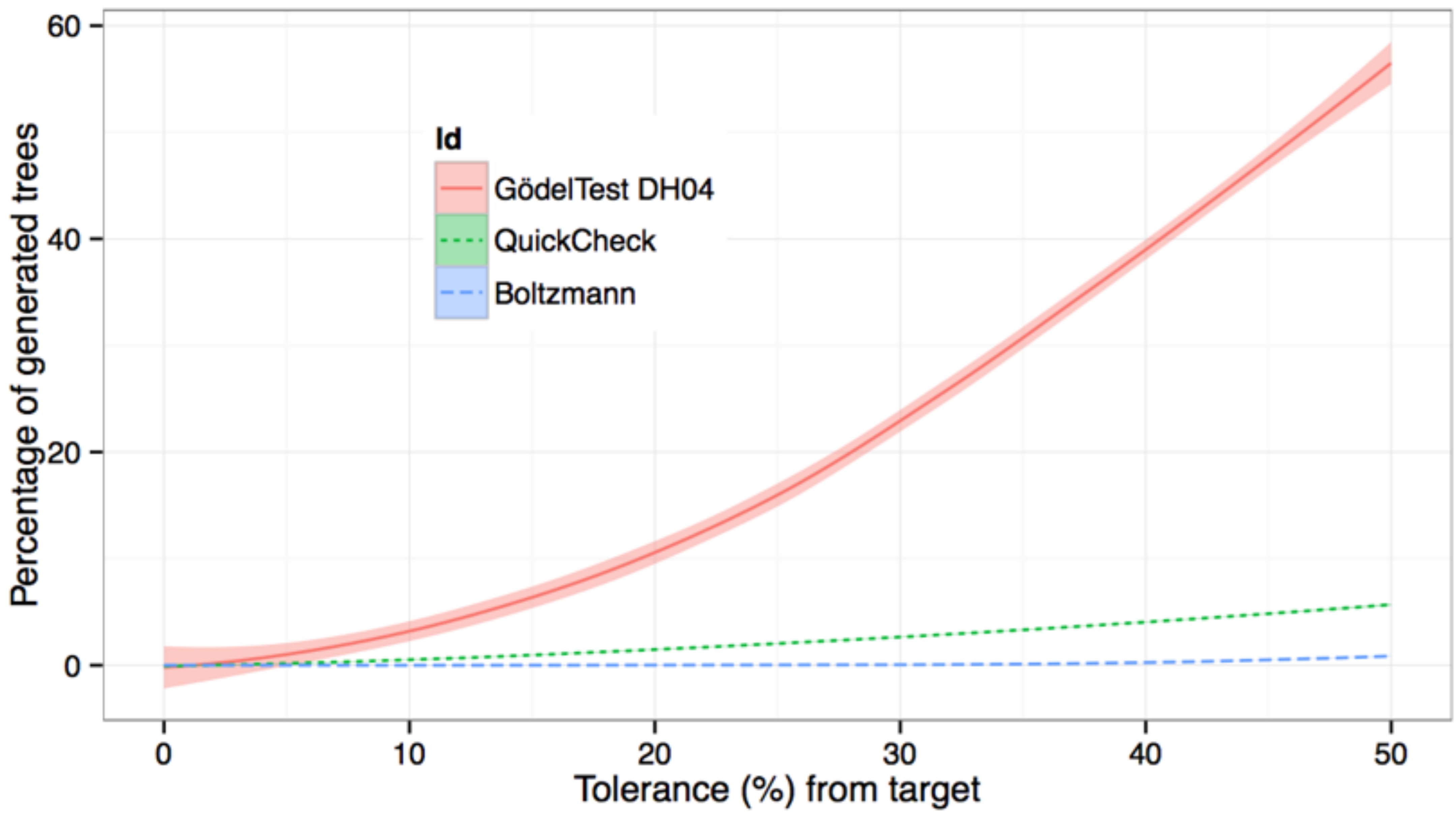


GödelTest (Decay Distribution)

GödelTest (Decay Histogram)

Results

The percentage of generated trees within a given tolerance of the bias objective target of size = 100, height = 36



Summary

GödelTest can efficiently generate highly-structured test data with specific desirable properties

The use of a non-deterministic program as a generator enables GödelTest to generate a wider range of data structures than Boltzmann samplers

The choice model is abstracted from the generator, and the local probability distributions associated with the model are optimised using metaheuristic search

Any property that is quantifiable can be used to specify a bias objective, and GödelTest is able to optimise for multiple bias objectives simultaneously

What is SBSE?

What is SBST?

Why is it not real-world (enough)?

Examples of real-world applications:

- Optimizing test case selection
- Generating complex test data
- Searching for diverse test suites

Test Set Diameter: Quantifying the Diversity of Sets of Test Cases

Robert Feldt and Simon Poulding
Software Engineering Research Lab
Blekinge Institute of Technology
Karlskrona, Sweden
Email: robert.feldt@bth.se, simon.poulding@bth.se

David Clark and Shin Yoo
Department of Computer Science
University College London
London, UK
Email: david.clark@ucl.ac.uk, shin.yoo@ucl.ac.uk

***Abstract*—A common and natural intuition among software testers is that test cases need to differ if a software system is to be tested properly and its quality ensured. Consequently, much research has gone into formulating distance measures for how test cases, their inputs and/or their outputs differ. However, common to these proposals is that they are data type specific and/or calculate the diversity only between pairs of test inputs, traces or outputs.**

We propose a new metric to measure the diversity of sets of tests: the test set diameter (TSDm). It extends our earlier, pairwise test diversity metrics based on recent advances in information theory regarding the calculation of the normalized compression distance (NCD) for multisets. An advantage is that TSDm can be applied regardless of data type and on any test-related information, not only the test inputs. A downside is the increased computational time compared to competing approaches.

Our experiments on four different systems show that the test set diameter can help select test sets with higher structural and fault coverage than random selection even when only applied to test inputs. This can enable early test design and selection, prior to even having a software system to test, and complement other types of test automation and analysis. We argue that this quantification of test set diversity creates a number of opportunities to better understand software quality and provides practical ways to increase it.

has support in the research literature. For example, adaptive random testing [1] only adds a new, randomly-generated test case if it has large distance to existing test cases. But Chen et al. [1] also note that a number of testing methods such as Restricted Random Testing [2], Antirandom testing [3], and Quasi-Random Testing [4] are all based on the same idea: 'evenly spreading' test cases over the input domain. Critical to the success of these techniques is a generically applicable diversity measure and Chen et al. go as far as saying that 'We have come to realise that "even spreading" can be better described as a form of diversity' [1]. They describe a distance calculation scheme based on the category-partition (partition testing) method, but it requires that the tester manually identifies categories and levels which can be varied.

Most approaches to quantifying diversity among test cases are specific to a certain type of data. It is common to assume that the data is numeric since there are a multitude of existing distance functions that can then be applied. One example is the approach of Bueno et al. [5] which selects test sets that maximize the sum of the distances from each test input to its nearest neighbor. In their empirical work they use the Euclidean distance which requires the inputs to be numerical vectors. More recently, Alshawan et al. [6] proposed to select

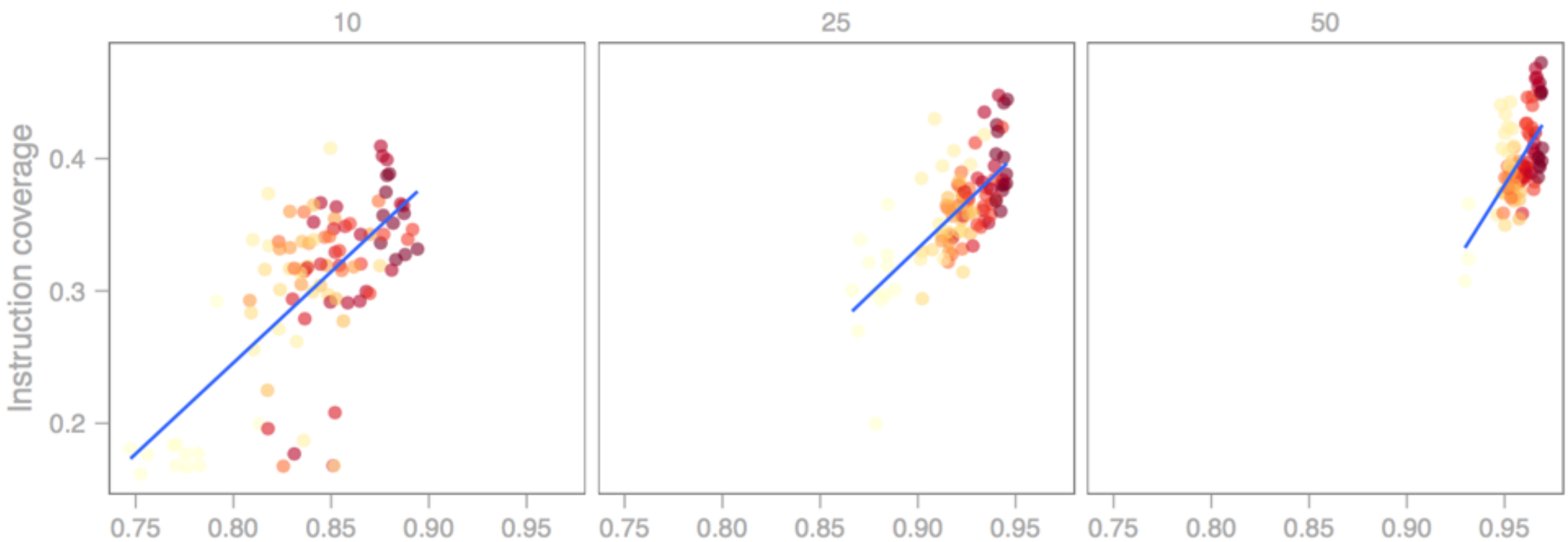
The Kolmogorov complexity of a string of symbols, x, is the length of the shortest program that outputs x [10]. It is a measure of the information contained in x, and we denote it here as $K(x)$. The conditional Kolmogorov complexity of x given y, denoted $K(x|y)$ is the length of the shortest program that outputs x given the input y.

Normalized Info Distance

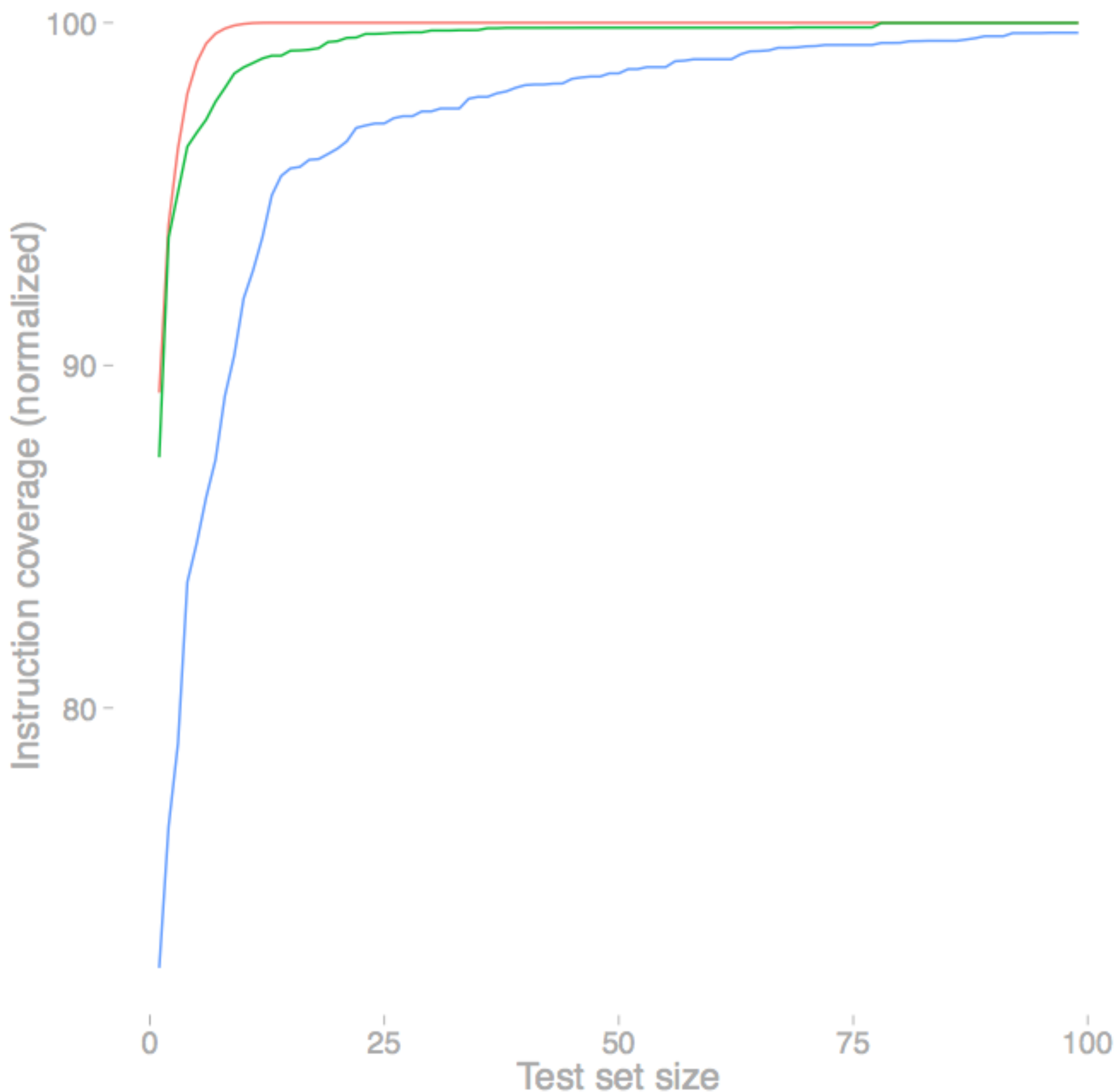
$$\mathbf{NID}(x,y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}$$

Normalized Compression Distance

$$\mathbf{NCD}(x,y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$



(a) JEuclid



100
90
80
Instruction coverage (normalized)
0
25
50
75
100
Test set size

Thanks to

My colleague Dr. Simon Poulding for many of the slides and for our collaboration on GödelTest

My other colleagues, students and co-authors for the research presented here...

Thank you! Questions?

robert.feldt@gmail.com

http://www.robertfeldt.net

@drfeldt