CCF-1629431
CCF-1703637

Statistical Algorithmic Profiling for Randomized Approximate Programs

Keyur Joshi, Vimuth Fernando, Sasa Misailovic
University of Illinois at Urbana-Champaign

ICSE 2019

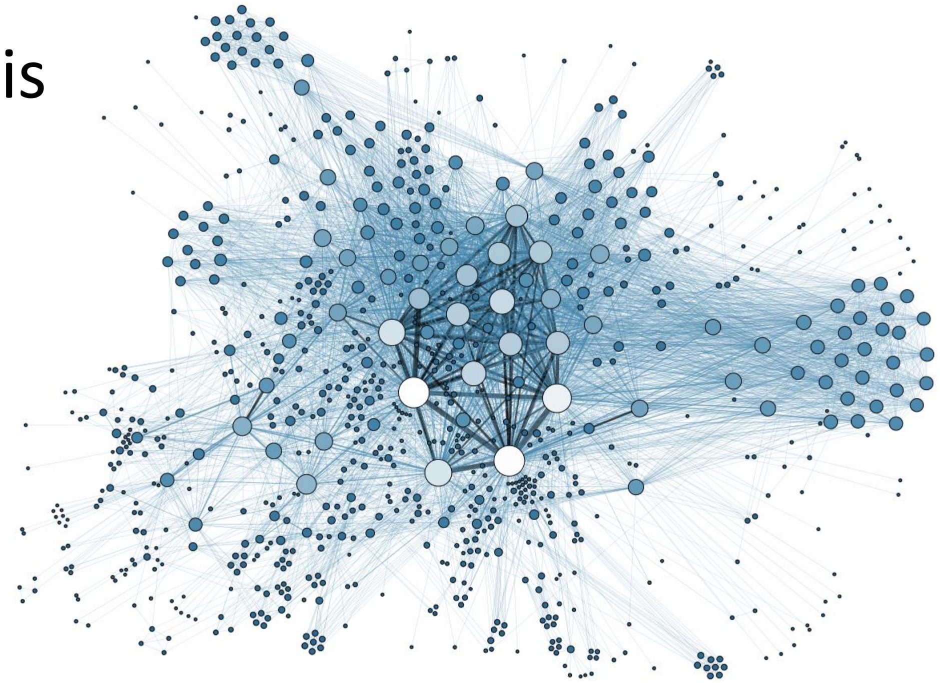
ARC UIUC

Randomized Approximate Algorithms

Modern applications deal with large amounts of data

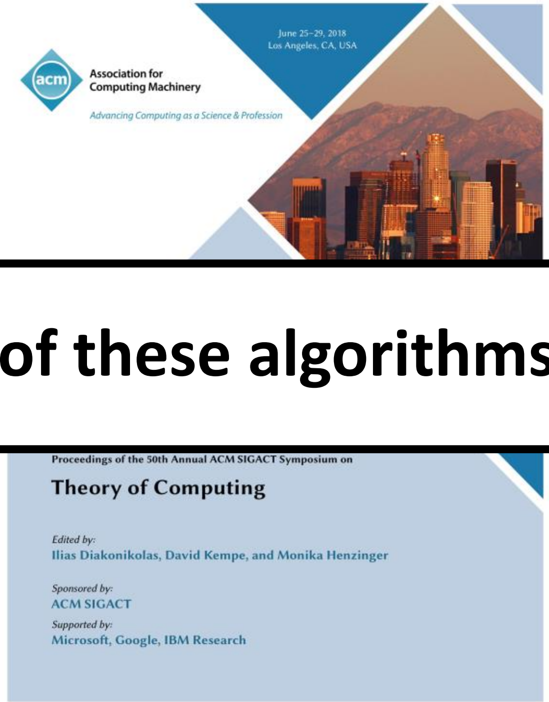
Obtaining exact answers for such applications is resource intensive

Approximate algorithms give a “good enough” answer in a much more efficient manner



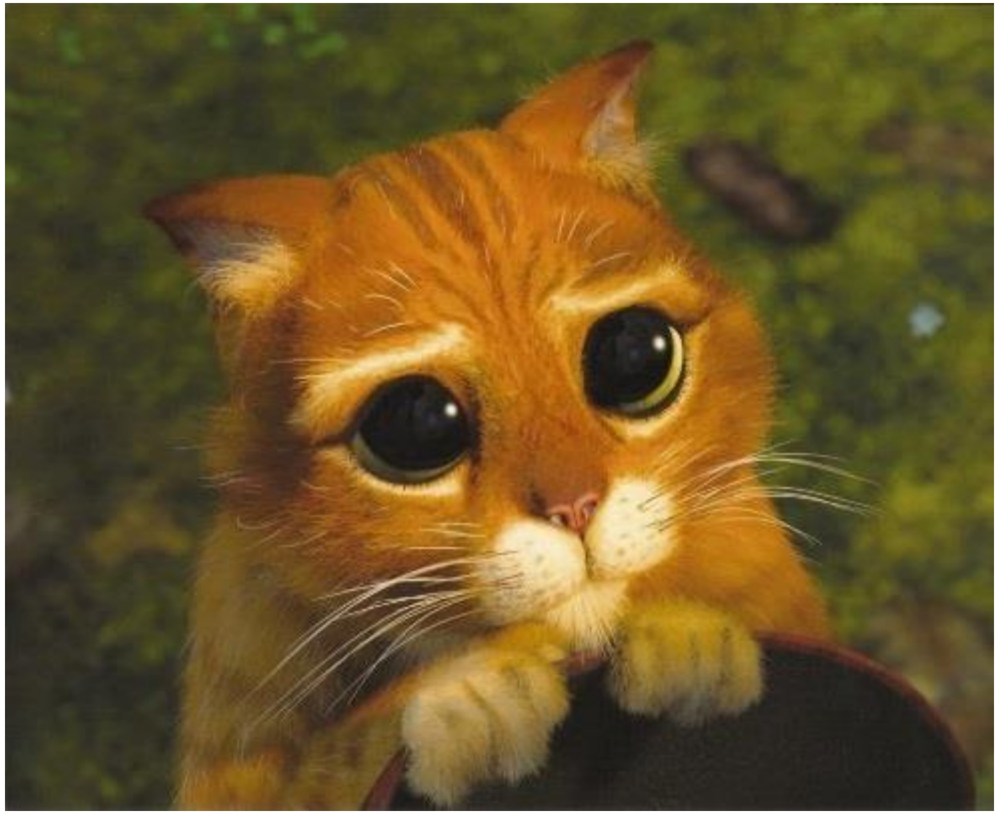
Randomized Approximate Algorithms

Randomized approximate algorithms have attracted the attention of many authors and researchers



Developers still struggle to properly test implementations of these algorithms

Example Application: Finding Near-Duplicate Images



Locality Sensitive Hashing (LSH)

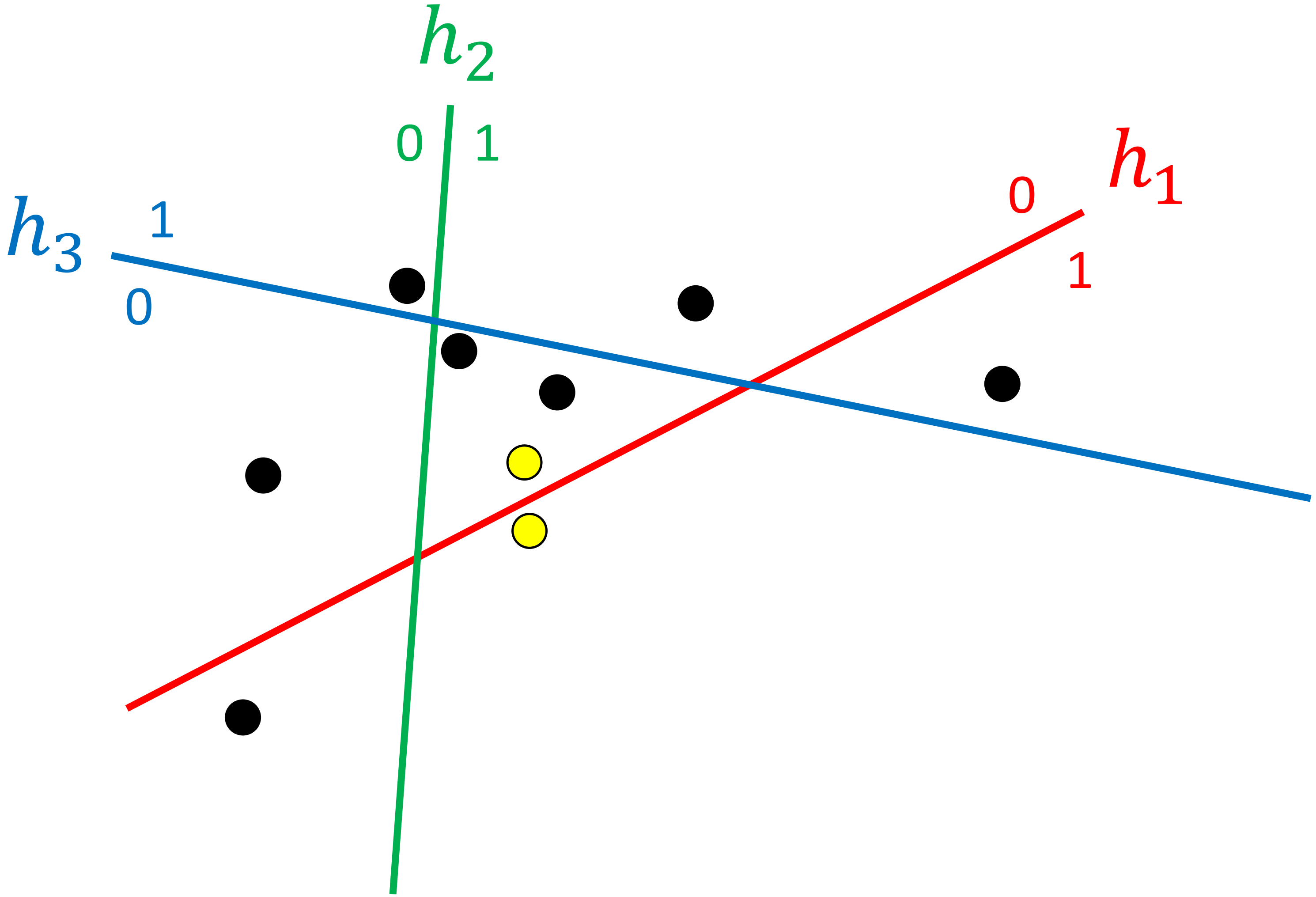
Finds vectors near a given vector in high dimensional space

LSH randomly chooses some *locality sensitive* hash functions in every run

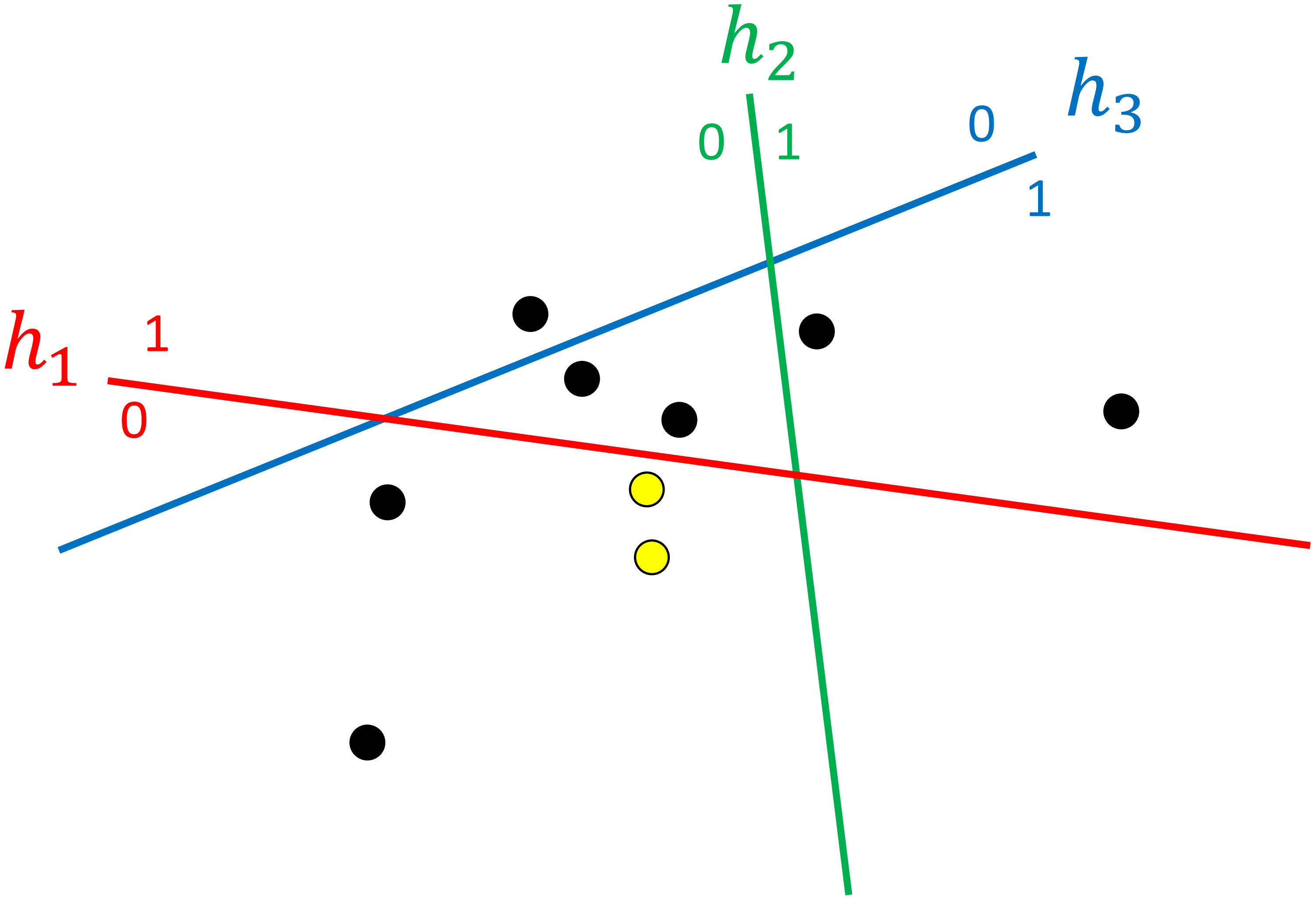
Locality sensitive – <u>nearby</u> vectors are <u>more likely</u> to have the <u>same hash</u>

Every run uses different hash functions – output can vary

Locality Sensitive Hashing (LSH) Visualization

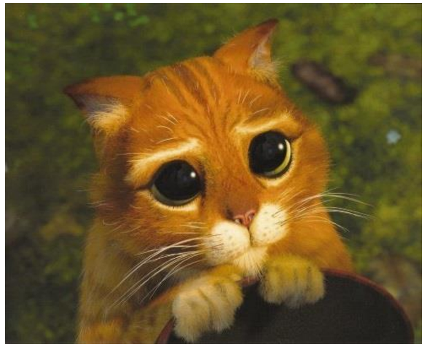


Locality Sensitive Hashing (LSH) Visualization



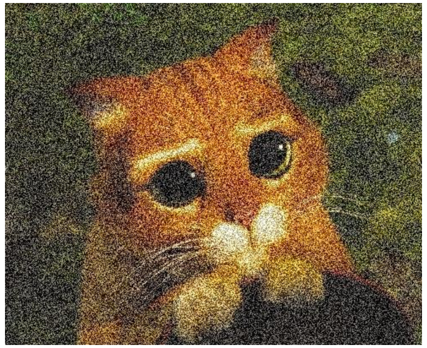
Comparing Images with LSH

Suppose, over 100 runs, an LSH implementation considered the images similar 90 times



Is this the expected behavior?

Usually, algorithm designers state the expected behavior by providing an accuracy specification



We wish to ensure that the implementation satisfies the accuracy specification

LSH Accuracy Specification*

Correct LSH implementations consider two vectors a and b to be neighbors with probability $p_{sim} = 1 - \left(1 - p_{a,b}^{k}\right)^{l}$ over runs

p_{sim} depends on:

- k, l: algorithm parameters (number of hash functions)
- $p_{a,b}$: dependent on the hash function and the distance between a and b (part of the specification)

*P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in STOC 1998

Challenges in Testing an LSH Implementation

Output can vary in every run due to different hash functions

Need to run LSH multiple times to observe value of p_{sim}

Need to compare expected and observed values of p_{sim}

Values may not be exactly the same – how close must they be?

Need to use an appropriate statistical test for such a comparison

Testing an LSH Implementation Manually

To test manually, the developer must provide:

Algorithm Parameters (for LSH: range of k, l values)	Implementation Runner
Appropriate Statistical Test	Number of Times to Run LSH
Multiple Test Inputs	Visualization Script

Testing an LSH Implementation With AxProf

To test with AxProf, the developer must provide:

- Accuracy / Performance Specification (math notation)
- Input and Output Types (for LSH: list of vectors)
- Algorithm Parameters (for LSH: range of k, l values)
- Implementation Runner

AxProf

- Appropriate Statistical Test
- Number of Times to Run LSH
- Multiple Test Inputs
- Visualization Script

Approximate Algorithm
Testing an ~~LSH~~ Implementation With AxProf

To test with AxProf, the developer must provide:

- Accuracy / Performance Specification (math notation)
- Input and Output Types (vectors / matrices / maps)
- Algorithm Parameters
- Implementation Runner
- Appropriate Statistical Test
- Number of Samples (runs / inputs)
- Multiple Test Inputs
- Visualization Script

AxProf

LSH Accuracy Specification Given to AxProf

Math Specification: A vector pair (a, b) appears in the output if LSH considers them neighbors. This should occur with probability $p_{sim} = 1 - \left(1 - p_{a,b}^{k}\right)^{l}$

AxProf specification:

```
Input list of (vector of real);
Output list of (pair of (vector of real));
forall a in Input, b in Input :
    Probability over runs [ [a, b] in Output ] ==
        1 - (1 – (p_ab(a, b)) ^ k) ^ l
```

p_ab is a helper function that calculates $p_{a,b}$

Example LSH Implementation: TarsosLSH

Popular (150 stars) LSH implementation in Java available on GitHub*

Includes a (faulty) benchmark which runs LSH once and reports accuracy

AxProf found a fault not detected by the benchmark

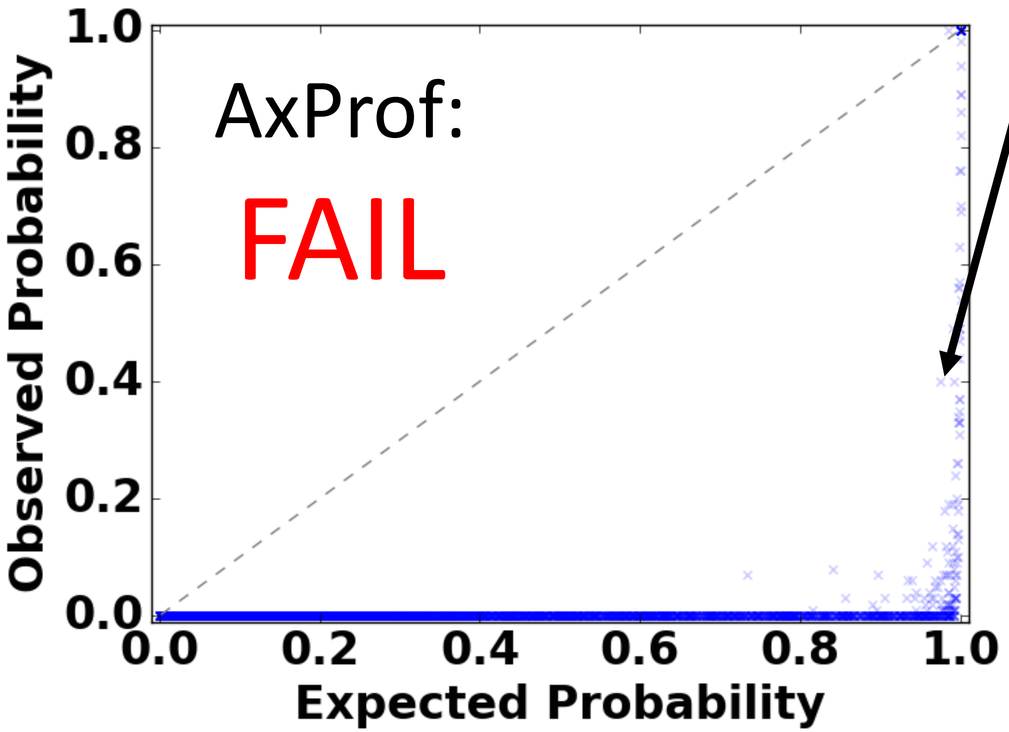
Fault is present for one hash function for the ℓ_1 distance metric

*https://github.com/JorenSix/TarsosLSH

TarsosLSH Failure Visualization 1

Represents a pair of neighboring vectors **Should ideally lie along the diagonal**

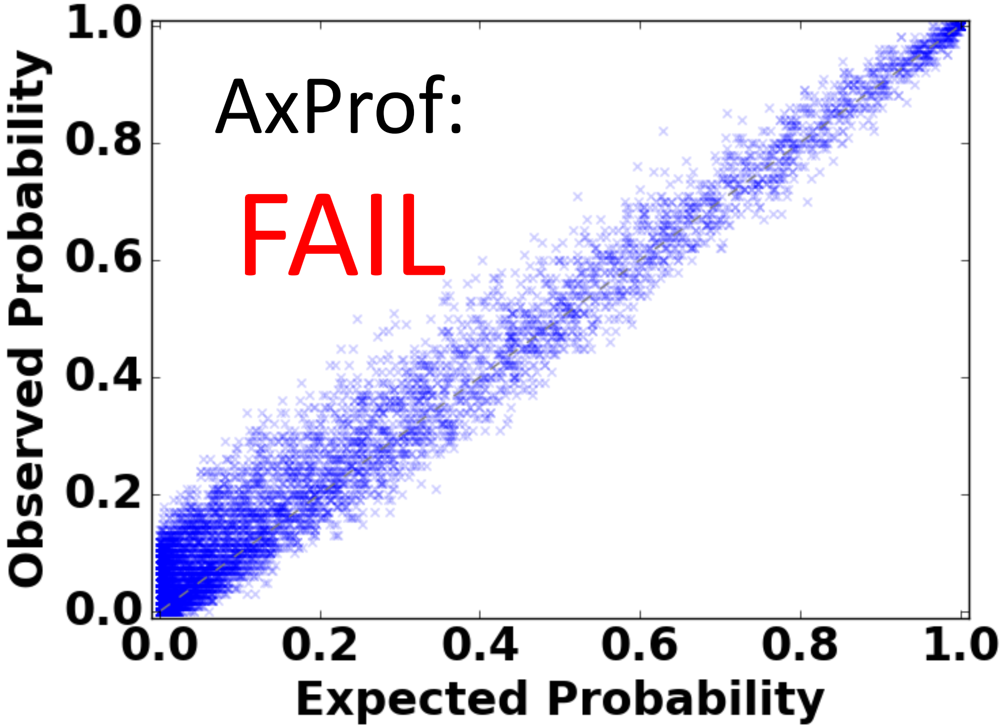
Obtained by running TarsosLSH multiple times



Obtained from specification

We found and fixed 3 faults and ran AxProf again

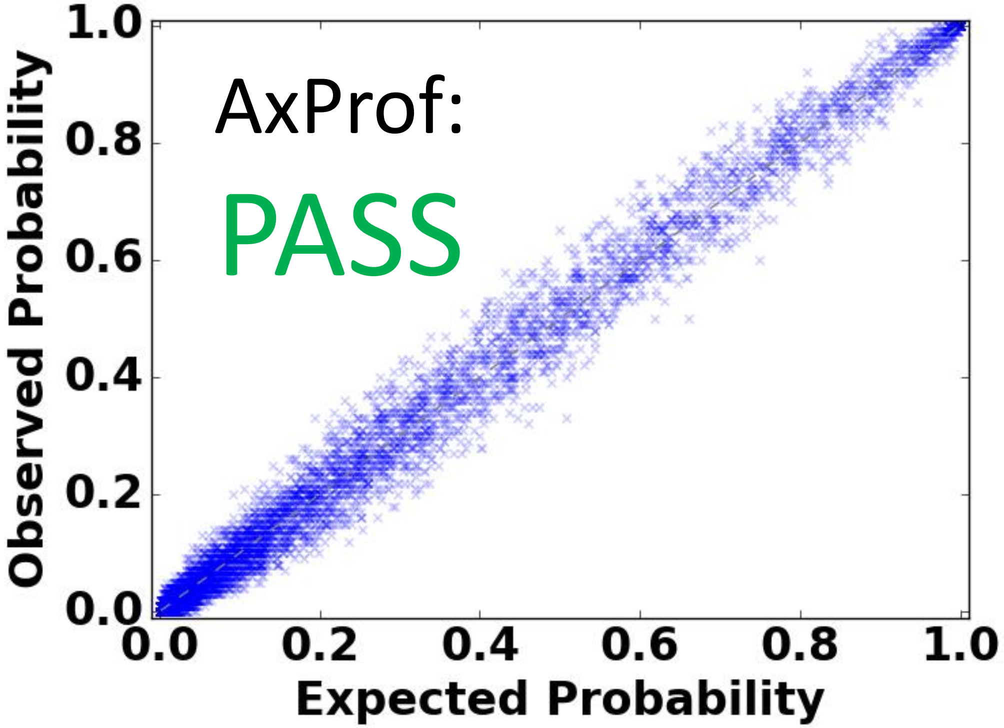
TarsosLSH Failure Visualization 2



Contains 1 subtle fault

Visual analysis not sufficient!

Visualization of Corrected TarsosLSH



AxProf Accuracy Specification Language

Handles a wide variety of algorithm specifications

AxProf language specifications appear very similar to mathematical specifications

Expressive:

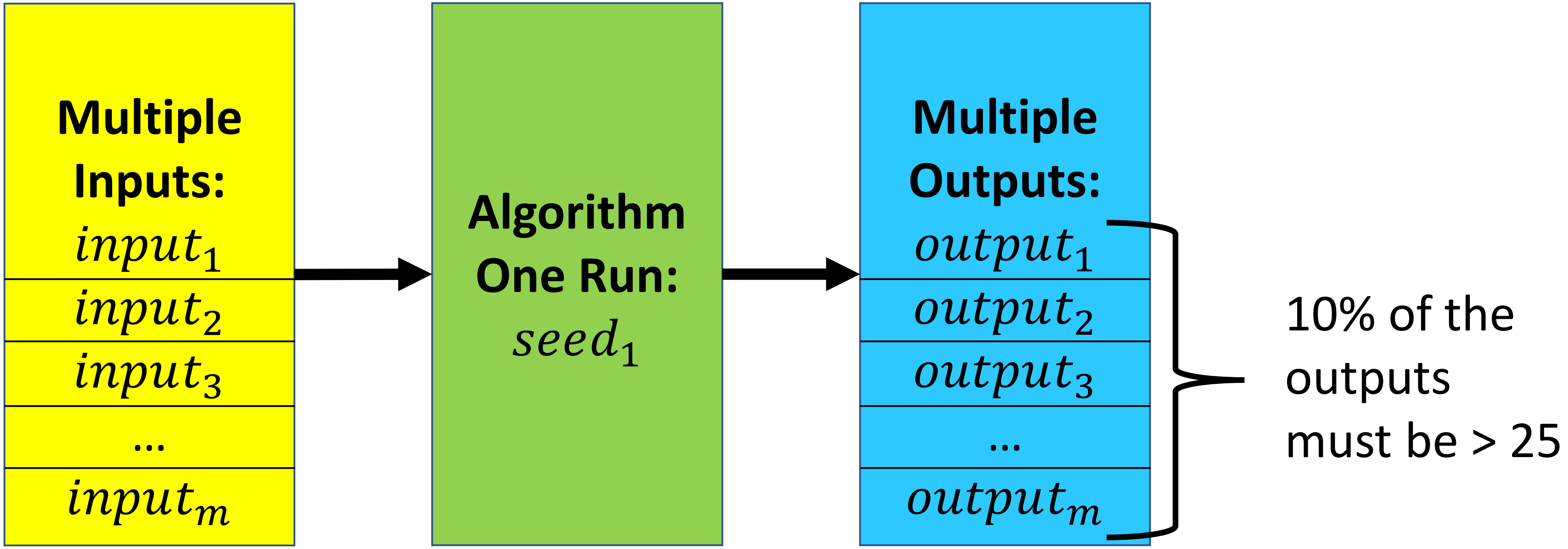
- Supports list, matrix, and map data structures
- Supports probability and expected value specifications
- Supports specifications with universal quantification over input items

Unambiguous:

- Explicit specification of probability space – over inputs, runs, or input items

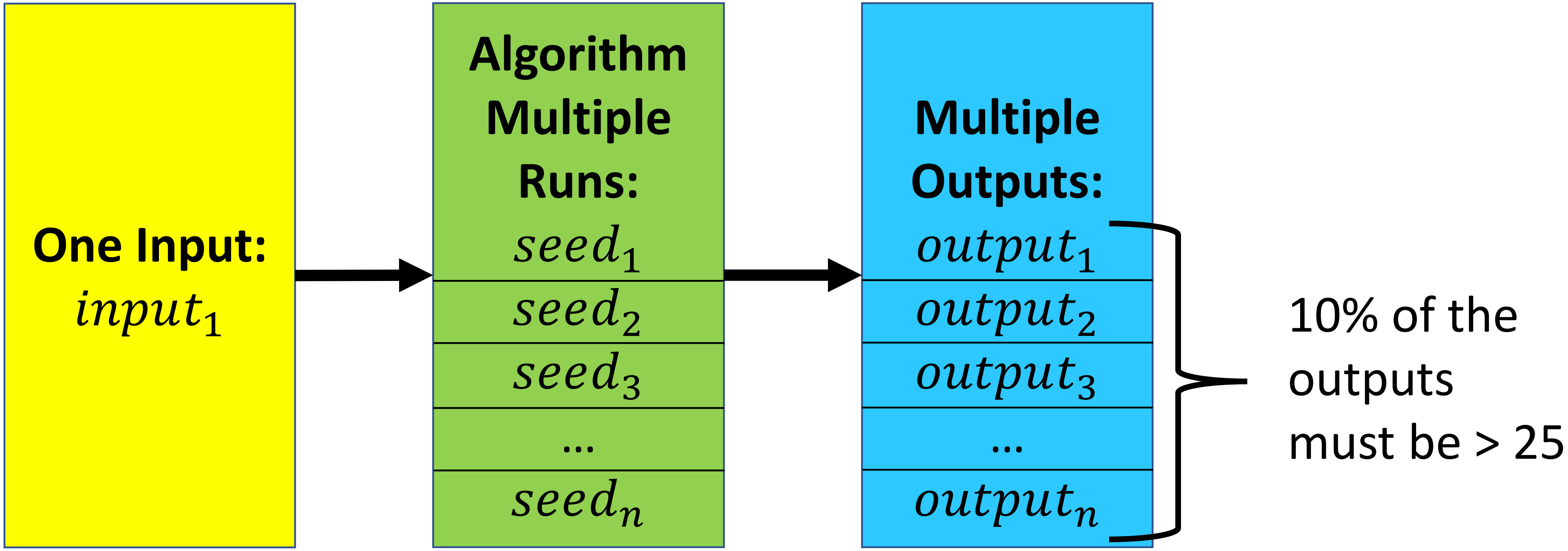
Accuracy Specification Example 1: Probability over inputs

```
Probability over  inputs  [Output > 25] == 0.1
```



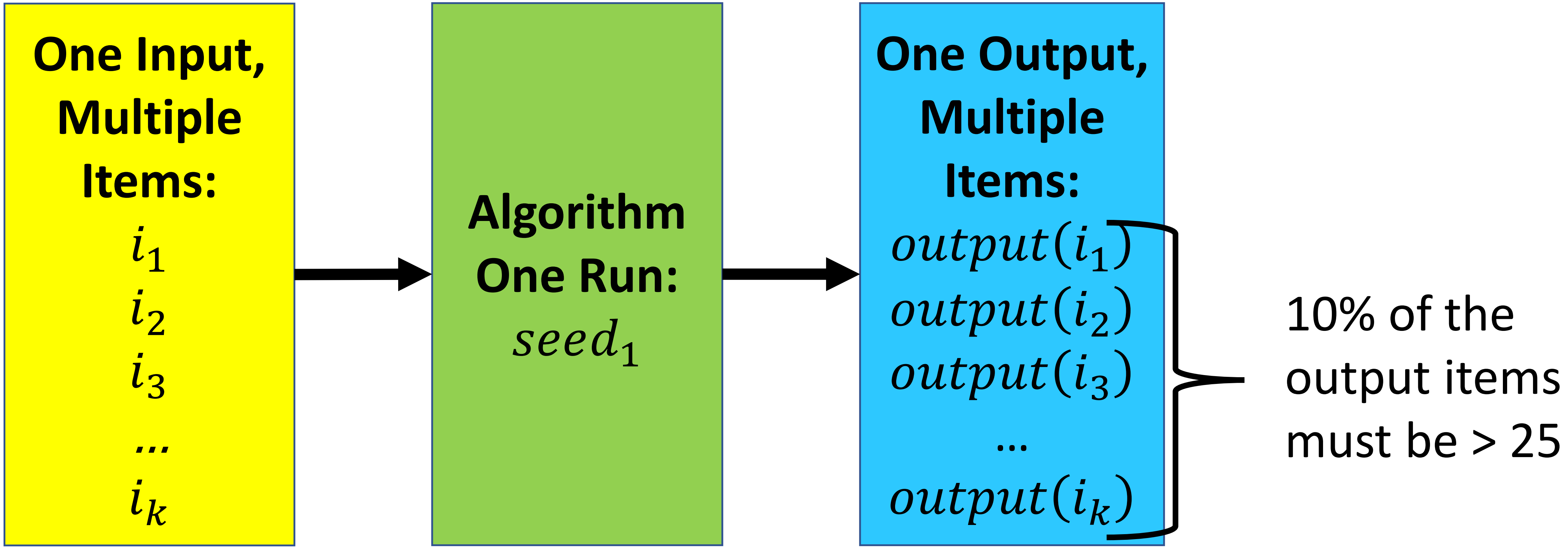
Accuracy Specification Example 2: Probability over runs

```
Probability over    runs    [Output > 25] == 0.1
```



Accuracy Specification Example 3: Probability over input items

```
Probability over i in Input [Output[i] > 25] == 0.1
```



Accuracy Specification Example 4: Expectation

```
Expectation over inputs [Output] == 100
```

```
Expectation over runs [Output] == 100
```

```
Expectation over i in Input [Output[i]] == 100
```

Accuracy Specification Example 5: Universal quantification

```
forall i in Input:
    Probability over runs [Output [i] > 25] == 0.1
```

One Input, Multiple Items:
i_1
i_2
…
i_k

→

Algorithm Multiple Runs:

$seed_1$
$seed_2$
…
$seed_n$

→

Multiple Outputs, Multiple Items:
$output_{1\ldots n}(i_1)$
$output_{1\ldots n}(i_2)$
…
$output_{1\ldots n}(i_k)$

→

Multiple Outputs per Item:

$output_1(i_1)$
…
$output_n(i_1)$

10% of the outputs **<u>for every input item</u>** must be > 25

Accuracy Specification Testing

AxProf generates code to fully automate specification testing:

1. **Generate** inputs with varying properties
2. **Gather** outputs of the program from multiple runs/inputs
3. **Test** the outputs against the specification with a statistical test
4. **Combine** the results of multiple statistical tests, if required
5. **Interpret** the final combined result (PASS/FAIL)

LSH: Choosing a Statistical Test

AxProf accuracy specification for LSH:

```
forall a in Input, b in Input :
  Probability over runs [[a, b] in Output] == 1-(1-(p_ab(a,b))^k)^l
```

Must compare values of $p_{a,b}$ for every a, b in input
Then combine results of each comparison into a single result

AxProf uses the non-parametric **binomial test** for each probability comparison

- Non-parametric – does not make any assumptions about the data

For `forall`, AxProf combines individual statistical tests using **Fisher's method**

LSH: Choosing the Number of Runs

Number of runs for the binomial test depends on desired level of confidence:

- $\boldsymbol{\alpha}$**:** Probability of incorrectly assuming a correct implementation is faulty (Type 1 error)
- $\boldsymbol{\beta}$**:** Probability of incorrectly assuming a faulty implementation is correct (Type 2 error)
- $\boldsymbol{\delta}$**:** Minimum deviation in probability that the binomial test should detect

Formula for calculating the number of runs: $\left(\frac{z_{1-\frac{\alpha}{2}}\sqrt{p_0(1-p_0)}+z_{1-\beta}\sqrt{p_a(1-p_a)}}{\delta}\right)^2$

We choose $\alpha = 0.05, \beta = 0.2, \delta = 0.1$ (commonly used values)

- AxProf calculates that **200 runs** are necessary

LSH: Generating Inputs

```
Input list of (vector of real);
forall a in Input, b in Input :
  Probability over runs [[a, b] in Output] == 1-(1-(p_ab(a,b))^k)^l
```

There is an implicit requirement that this specification should be satisfied for every input

AxProf provides flexible input generators for various input types

- User can provide their own input generators

LSH: Generating Inputs

For LSH, AxProf can generate a list of input vectors with adjustable properties:

- Average distance between vectors
- Number of vectors in input

AxProf determines which input properties affect the accuracy of the algorithm using the Maximal Information Coefficient (MIC)*:

- The average distance affects LSH accuracy
- The number of input vectors does not affect LSH accuracy

*See paper for more details

Performance Specification Testing

The AxProf language also supports time and memory specifications

Time specification for LSH:
Asymptotic notation: $O(kln)$ AxProf: `k*l*size(Input)`

Memory specification for LSH:
Asymptotic notation: $O(ln)$ AxProf: `l*size(Input)`

Like accuracy specifications, AxProf tests performance specifications via statistical tests

Performance Specification Testing

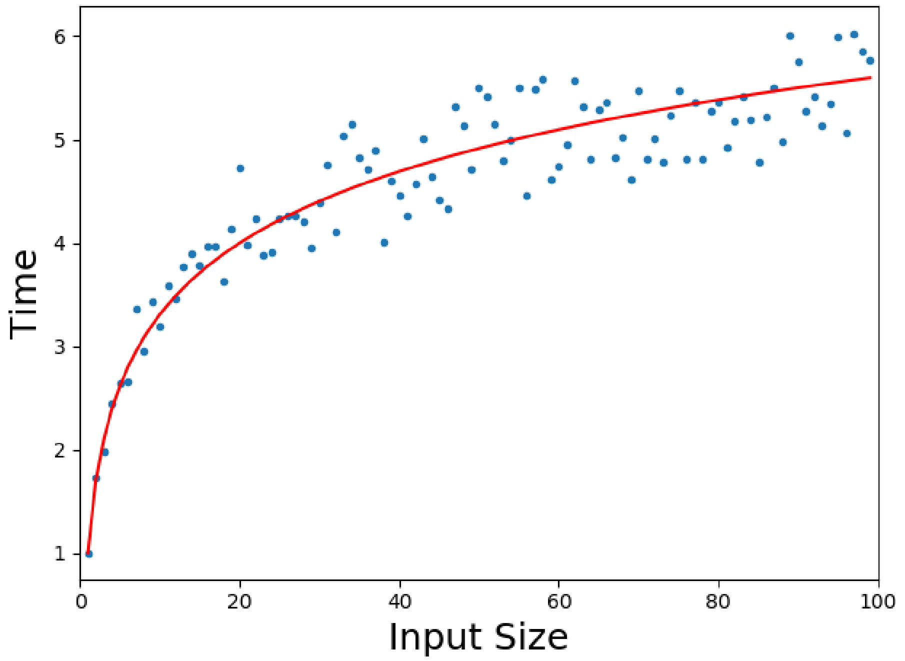
AxProf gathers performance data across multiple runs and algorithm parameter values

AxProf fits a curve and compares it to the specification (like algorithmic profilers*)

To check for conformance: R^2 metric

If R^2 is lower than a threshold, AxProf reports a failure

Expected time complexity: $O(\log(n))$
Fitted curve:



*D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," and E. Coppa et al., "Input-sensitive profiling," both in PLDI, 2012.

Research Questions

- **Research Question 1:** Can AxProf find accuracy bugs in approximate algorithm implementations?
- **Research Question 2:** Can AxProf identify input parameters that affect algorithm accuracy?

See Paper

- **Research Question 3:** Can AxProf find performance anomalies in algorithm implementations?

Tested Algorithms

Algorithm
Locality Sensitive Hashing (LSH)
Bloom Filter
Count-Min Sketch
HyperLogLog
Reservoir Sampling
Approximate Matrix Multiply
Chisel/blackscholes
Chisel/sor
Chisel/scale

5 Big Data Algorithms

1 Approximate Numerical Computation Algorithm

3 Algorithms Running on Imprecise Hardware

Tested Algorithms

Algorithm	**Algorithm Parameters**
Locality Sensitive Hashing (LSH)	No. hash functions and hash tables
Bloom Filter	Capacity and maximum false positive probability
Count-Min Sketch	Error factor and error probability
HyperLogLog	Number of hash values
Reservoir Sampling	Reservoir size
Approximate Matrix Multiply	Sampling rate
Chisel/blackscholes	Reliability factor
Chisel/sor	Reliability factor and no. iterations
Chisel/scale	Reliability factor and scale factor

Each parameter can take multiple values

We chose ranges of parameter values to test based on algo. author recommendations

A particular combination of parameter values is an *algorithm configuration*

Tested Algorithms

Algorithm	Algorithm Parameters	Accuracy Specification Type
Locality Sensitive Hashing (LSH)	No. hash functions and hash tables	Probability over runs with universal quantification
Bloom Filter	Capacity and maximum false positive probability	Probability over input items
Count-Min Sketch	Error factor and error probability	Probability over input items
HyperLogLog	Number of hash values	Probability over inputs
Reservoir Sampling	Reservoir size	Probability over runs with universal quantification
Approximate Matrix Multiply	Sampling rate	Probability over runs
Chisel/blackscholes	Reliability factor	Probability over runs
Chisel/sor	Reliability factor and no. iterations	Probability over runs
Chisel/scale	Reliability factor and scale factor	Expectation over runs

Algorithm	Implementation
Locality Sensitive Hashing	TarsosLSH java-LSH
Bloom Filter	libbf BloomFilter
Count-Min Sketch	alabid awnystrom
HyperLogLog	yahoo ekzhu
Reservoir Sampling	yahoo sample
Matrix Multiplication	RandMatrix mscs
blackscholes	Chisel
sor	Chisel
scale	Chisel

From GitHub (except Chisel)

Selection factors:

- No. of stars on GitHub
- Repository activity
- GitHub search rank
- Java / Python / C / C++

Implementation	Tested Configurations	Configurations w/ Accuracy Failures
TarsosLSH	12	**12**
java-LSH	4	**4**
libbf	60	0
BloomFilter	60	0
alabid	90	**90**
awnystrom	90	**81**
Yahoo (HyperLogLog)	40	0
ekzhu	40	**2***
Yahoo (Reservoir)	100	0
sample	100	0
RandMatrix	243	**30**
mscs	16	0
Chisel (blackscholes)	3	0
Chisel (sor)	108	0
Chisel (scale)	20	0

AxProf detected statistical test failures in six implementations

After manual inspection – found faults in five implementations

One false positive (*) for ekzhu HyperLogLog

Errors in Implementations

We submitted a pull request for each faulty implementation:

- Four faults were caused by the use of incorrect hash functions
- One fault was caused by incorrect sampling to improve efficiency

Four pull requests were accepted – one is still pending

Developer feedback:

“Hi, I am the creator of TarsosLSH and I have just seen your paper, especially the parts relevant to TarsosLSH… I would like to thank you for your work and for the well documented merge requests.”

False Warning for HyperLogLog (ekzhu)

The correctness of AxProf depends on the correctness of the specification

Some specifications fail to capture fine details – may cause failures in AxProf's statistical tests for specific inputs

HyperLogLog applies error correction if the output is below a certain threshold

AxProf found failures when the output size is very close to the threshold

Algorithm	Implementation	Tested Configurations	Configurations w/ Accuracy Failures	Time/Memory Spec. Test Results
Locality Sensitive Hashing	TarsosLSH	12	12	Pass
	java-LSH	4	4	Pass
Bloom Filter	libbf	60	0	Fail
	BloomFilter	60	0	Fail
Count-Min Sketch	alabid	90	90	Pass
	awnystrom	90	81	Fail†
HyperLogLog	yahoo	40	0	Fail
	ekzhu	40	2*	Pass
Reservoir Sampling	yahoo	100	0	Fail
	sample	100	0	Fail†
Matrix Multiplication	RandMatrix	243	30	Pass
	mscs	16	0	Pass
blackscholes	Chisel	3	0	Pass
sor	Chisel	108	0	Pass
scale	Chisel	20	0	Pass

†False positives: measurement noise

Why Were Developer-Written Tests Inadequate?

- Focusing only on performance testing
- Running the implementation only once
- Running on only one input
- Running on only one algorithm configuration

AxProf alleviates these inadequacies via an easy to use framework

Conclusion

AxProf is a tool for accuracy and performance profiling

Automates many tasks for testing the implementations of emerging randomized and approximate algorithms

With AxProf, we found five faulty implementations from a set of 15 implementations

Check out AxProf at axprof.org