Statistical Algorithmic Profiling for Randomized Approximate Programs

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

ICSE 2019
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* – nearby vectors are more likely to have the same hash

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.
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)
- Appropriate Statistical Test
- Multiple Test Inputs
- Implementation Runner
- Number of Times to Run LSH
- Visualization Script
Testing an LSH Implementation With AxProf

To test with AxProf, the developer must provide:

<table>
<thead>
<tr>
<th>Accuracy / Performance Specification (math notation)</th>
<th>Input and Output Types (for LSH: list of vectors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm Parameters (for LSH: range of $k, l$ values)</td>
<td>Implementation Runner</td>
</tr>
<tr>
<td>Appropriate Statistical Test</td>
<td>Number of Times to Run LSH</td>
</tr>
<tr>
<td>Multiple Test Inputs</td>
<td>Visualization Script</td>
</tr>
</tbody>
</table>
Testing an LSH Implementation With AxProf

To test with AxProf, the developer must provide:

- **Accuracy / Performance Specification (math notation)**
- **Algorithm Parameters**
- **Appropriate Statistical Test**
- **Multiple Test Inputs**
- **Implementation Runner**
- **Input and Output Types (vectors / matrices / maps)**
- **Number of Samples (runs / inputs)**
- **Visualization Script**
Math Specification: A vector pair \((a, b)\) appears in the output if LSH considers them neighbors. This should occur with probability \(p_{sim} = 1 - (1 - p_{a,b}^k)^l\).

AxProf specification:

\[
\text{Input list of (vector of real);} \\
\text{Output list of (pair of (vector of real));} \\
\text{forall } a \text{ in Input, } b \text{ in Input :} \\
\quad \text{Probability over runs } \left[ \left[ a, b \right] \text{ in Output } \right] = 1 - (1 - (p_{ab}(a, b))^k)^l
\]

\(p_{ab}\) 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 $\ell_1$ distance metric

*https://github.com/JorenSix/TarsosLSH
TarsosLSH Failure Visualization 1

AxProf: FAIL

Obtained by running TarsosLSH multiple times

Obtained from specification

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

We found and fixed 3 faults and ran AxProf again
TarsosLSH Failure Visualization 2

AxProf: FAIL

Contains 1 subtle fault

Visual analysis not sufficient!
Visualization of Corrected TarsosLSH

AxProf: PASS
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

\[ \text{Probability over inputs} \quad \text{[Output} > 25\text{]} = 0.1 \]

Multipl e Inputs: 
\( \text{input}_1 \)
\( \text{input}_2 \)
\( \text{input}_3 \)
\( \ldots \)
\( \text{input}_m \)

Algorithm One Run: 
\( \text{seed}_1 \)

Multipl e Outputs: 
\( \text{output}_1 \)
\( \text{output}_2 \)
\( \text{output}_3 \)
\( \ldots \)
\( \text{output}_m \)

10% of the outputs must be > 25
Accuracy Specification Example 2: Probability over runs

Probability over runs

\[ \text{Output} > 25 \] == 0.1

One Input:

\( \text{input}_1 \)

Algorithm Multiple Runs:

- seed_1
- seed_2
- seed_3
- ... seed_n

Multiple Outputs:

- output_1
- output_2
- output_3
- ... output_n

10% of the outputs must be > 25
Accuracy Specification Example 3: Probability over input items

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

One Input, Multiple Items:
- $i_1$
- $i_2$
- $i_3$
- ...
- $i_k$

Algorithm

One Run:
- seed

One Output, Multiple Items:
- output($i_1$)
- output($i_2$)
- output($i_3$)
- ...
- output($i_k$)

10% of the output items must be > 25
Accuracy Specification Example 4: Expectation

Expectation over inputs \([\text{Output}] == 100\)

Expectation over runs \([\text{Output}] == 100\)

Expectation over \(i\) in Input \([\text{Output}[i]] == 100\)
Accuracy Specification Example 5: Universal quantification

forall \( i \) in Input:
Probability over runs \([\text{Output} [i] > 25] == 0.1\)

One Input, Multiple Items:
- \( i_1 \)
- \( i_2 \)
- ... 
- \( i_k \)

Algorithm Multiple Runs:
- \( seed_1 \)
- \( seed_2 \)
- ...
- \( seed_n \)

Multiple Outputs, Multiple Items:
- \( output_{1...n}(i_1) \)
- \( output_{1...n}(i_2) \)
- ...
- \( output_{1...n}(i_k) \)

Multiple Outputs per Item:
- \( output_1(i_1) \)
- ...
- \( output_n(i_1) \)

10% of the outputs for every input item 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:
\[
\text{forall } a \text{ in Input, } b \text{ in Input :}
\]
\[
\text{Probability over runs } [[a, b] \text{ in Output}] = 1-(1-(p_{ab}(a,b))^k)^l
\]

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 \textbf{binomial test} for each probability comparison
• Non-parametric – does not make any assumptions about the data

For \texttt{forall}, AxProf combines individual statistical tests using \textit{Fisher’s method}
**LSH: Choosing the Number of Runs**

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

- **α**: Probability of incorrectly assuming a correct implementation is faulty (Type 1 error)
- **β**: Probability of incorrectly assuming a faulty implementation is correct (Type 2 error)
- **δ**: Minimum deviation in probability that the binomial test should detect

Formula for calculating the number of runs:

\[
\left( \frac{z_{1-\alpha \sqrt{p_0(1-p_0)}+z_{1-\beta \sqrt{p_a(1-p_a)}}}{\delta} \right)^2
\]

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: $O(k \ln) \text{ AxProf: } k \times l \times \text{size(Input)}$

Memory specification for LSH:
Asymptotic notation: $O(ln) \text{ AxProf: } l \times \text{size(Input)}$

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

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?

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

See Paper
Tested Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>5 Big Data Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality Sensitive Hashing (LSH)</td>
<td>1 Approximate Numerical Computation Algorithm</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td></td>
</tr>
<tr>
<td>Count-Min Sketch</td>
<td></td>
</tr>
<tr>
<td>HyperLogLog</td>
<td></td>
</tr>
<tr>
<td>Reservoir Sampling</td>
<td></td>
</tr>
<tr>
<td>Approximate Matrix Multiply</td>
<td></td>
</tr>
<tr>
<td>Chisel/blackscholes</td>
<td></td>
</tr>
<tr>
<td>Chisel/sor</td>
<td></td>
</tr>
<tr>
<td>Chisel/scale</td>
<td></td>
</tr>
</tbody>
</table>
## Tested Algorithms

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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Algorithm Parameters</th>
<th>Accuracy Specification Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality Sensitive Hashing (LSH)</td>
<td>No. hash functions and hash tables</td>
<td>Probability over runs with universal quantification</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>Capacity and maximum false positive probability</td>
<td>Probability over input items</td>
</tr>
<tr>
<td>Count-Min Sketch</td>
<td>Error factor and error probability</td>
<td>Probability over input items</td>
</tr>
<tr>
<td>HyperLogLog</td>
<td>Number of hash values</td>
<td>Probability over input items</td>
</tr>
<tr>
<td>Reservoir Sampling</td>
<td>Reservoir size</td>
<td>Probability over runs with universal quantification</td>
</tr>
<tr>
<td>Approximate Matrix Multiply</td>
<td>Sampling rate</td>
<td>Probability over runs</td>
</tr>
<tr>
<td>Chisel/blackscholes</td>
<td>Reliability factor</td>
<td>Probability over runs</td>
</tr>
<tr>
<td>Chisel/sor</td>
<td>Reliability factor and no. iterations</td>
<td>Probability over runs</td>
</tr>
<tr>
<td>Chisel/scale</td>
<td>Reliability factor and scale factor</td>
<td>Expectation over runs</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Implementation</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>Locality Sensitive Hashing</td>
<td>TarsosLSH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java-LSH</td>
<td></td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>libbbf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BloomFilter</td>
<td></td>
</tr>
<tr>
<td>Count-Min Sketch</td>
<td>alabid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>awnystrom</td>
<td></td>
</tr>
<tr>
<td>HyperLogLog</td>
<td>yahoo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ekzhu</td>
<td></td>
</tr>
<tr>
<td>Reservoir Sampling</td>
<td>yahoo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sample</td>
<td></td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>RandMatrix</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mscs</td>
<td></td>
</tr>
<tr>
<td>blackscholes</td>
<td>Chisel</td>
<td></td>
</tr>
<tr>
<td>sor</td>
<td>Chisel</td>
<td></td>
</tr>
<tr>
<td>scale</td>
<td>Chisel</td>
<td></td>
</tr>
</tbody>
</table>

From GitHub (except Chisel)

Selection factors:
- No. of stars on GitHub
- Repository activity
- GitHub search rank
- Java / Python / C / C++
AxProf detected statistical test failures in six implementations.

After manual inspection – found faults in five implementations.

One false positive (*) for ekzhu HyperLogLog.

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