LAMP: Data Provenance for Graph Based Machine Learning Algorithms Through Derivative Computation

Anonymous Author(s)

ABSTRACT

Data provenance tracking determines the set of inputs related to a given output. It enables quality control and problem diagnosis in data engineering. Most existing techniques work by tracking program dependencies. They cannot quantitatively assess the importance of related inputs, which is critical to machine learning algorithms, in which an output tends to depend on a huge set of inputs while only some of them are of importance. In this paper, we propose LAMP, a provenance computation system for machine learning algorithms. Inspired by automatic differentiation (AD), LAMP quantifies the importance of an input for an output by computing the partial derivative. LAMP separates the original data processing and the more expensive derivative computation to different processes to achieve cost-effectiveness. In addition, it allows quantifying importance for inputs related to discrete behavior, such as control flow selection. The evaluation on a set of real world programs and data sets illustrates that LAMP produces more precise and succinct provenance than program dependence based techniques, with much less overhead. Our case studies demonstrate the potential of LAMP in problem diagnosis in data engineering.

1 INTRODUCTION

We are entering an era of data engineering. Compared to traditional software engineering, the complexity of data engineering largely lies in data and models. For example, many data processing programs, such as well-known machine learning programs, have a small size. But the data processed by these programs and the models generated are often large and complex. It poses new challenges to engineers such as how to validate outputs and how to diagnose problems. Note that faults may likely reside in data and models, while they are more likely present in programs in traditional engineering scenarios. Graph based machine learning (GML) is an important kind of data processing with increasing popularity. GML algorithms are widely used in various applications such as search engines, social network analysis and recommendation systems. Provided with an input graph model and initial weight values, GML algorithms generate an updated model. Most these algorithms are iterative. In each iteration, a vertex communicates with its neighbors and updates its value until all the values converge. Through multiple iterations, a vertex can affect other vertices that are many edges away. This is called the *rippling effect*. Due to the nature of such computation, it is highly challenging to determine the correctness of the generated models as a fault may get propagated through many steps and faulty states may get accumulated/obfuscated during propagation. Even if the user suspects the incorrectness of final outputs, she can hardly diagnose the procedure to identify the root cause, which could be present in the input graph model, the initial weight values, or even in the GML algorithm itself.

Data provenance is an important approach to addressing the problem. It identifies the input-output dependencies and/or records the operation history. There are a number of existing efforts [36–38, 47, 60] that aim to provide general frameworks for collecting data provenance for GML algorithms. Most of them focus on selectively collecting intermediate results at run time in an effort to provide crash recovery mechanisms, debugging support, and so on. However, these techniques can hardly reason about input-output dependencies. Dynamic information flow tracking, or tainting [23, 54], computes the set of inputs related to a given output, by monitoring program dependencies. However, it cannot quantify the importance of individual inputs. Due to the rippling effect of GML algorithms, an output tends to be dependent on a huge set of inputs even though most of them have negligible impact on the output.

In this paper, we propose LAMP, a technique to quantitatively reason about input importance. Inspired by *automatic differentiation* (AD) techniques [18, 19, 24, 30–33, 43, 53, 63], LAMP works by computing output derivatives regarding inputs. A large derivative indicates the input is of high importance for the output. A zero derivative indicates the input has no impact on the output. However, existing AD techniques cannot be directly used in GML provenance tracking due to the following reasons. (1) Derivative computation is closely coupled with the original computation, leading to high overhead for production runs; (2) Derivatives cannot be used to quantify the importance of inputs related to discrete behavior, such as the inputs whose changes may lead to control flow changes. They are common in GML algorithms; (3) AD techniques typically compute derivatives for a small number of inputs. However in GML, all inputs need to be considered. LAMP addresses these challenges. In particular, it considers the initial weight values (of vertices/edges) as the input, even though they may be initialized to constants and do not come from the program input. The iterative procedure then aggregates, propagates, and updates these values, driven by the graph structure. As such, variations to the initial weight value of a vertex have impact to the final outputs proportional to the structural importance of the vertex. Therefore, derivatives regarding these initial (constant) values reflect the importance of vertices. LAMP features a novel design that separates the original data processing from provenance computation so that very little overhead is introduced during production runs. It quantifies input importance for those that may induce control flow variations by spawning processes to concretely determine the output variations. The workflow of LAMP is as follows. It first collects a lightweight trace during production run that contains branch outcomes and a small set of states. The trace allows the decoupling of the original computation and the provenance computation. It transforms the original program to a new program that solely focuses on provenance computation. The transformed program takes the original inputs and the trace, and produces the provenance (i.e., the importance measurement for each input).

We make the following contributions.

- We formally define the problem of provenance computation for GML algorithms, which has the key challenge of quantifying input importance. We propose a novel solution based on computing the partial derivatives of each output variable with respect to the related input variables.
- We propose a novel design that decouples original data processing from provenance computation.
- We propose an execution based approach to quantify input importance for those related to control flow.
- We develop a prototype LAMP. Our evaluation on a set of real world GML algorithms and large data sets show that LAMP substantially outperforms program dependence tracking based approaches in terms of accuracy and efficiency, producing provenance sets that are orders of magnitude smaller with overhead that is 3-6 times lower. Our case studies demonstrate the potential of LAMP in data engineering, for helping the development process and finding bugs in input data, graph models, and even in GML algorithm implementations.

2 MOTIVATION

PageRank [52] is one of the most popular GML algorithms. It computes a numerical weight for each vertex in a graph to measure its relative importance within the graph, called the *rank value*. It is widely used in practice such as in search engines and social network analysis. Based on the assumption that a vertex becomes popular when it has many links with other popular vertices, the algorithm iteratively computes the rank value for a vertex by aggregating the rank values from its predecessor vertices. During each iteration, the rank value of a vertex is computed by the following equation:

$$PR(u) = \frac{1-d}{|V|} + d * \sum_{j \in B_u} \frac{PR(j) * k_j}{c_j} \tag{1}$$

, where B_u contains all the vertices which have at least one link/edge to u, c_j is the sum of outgoing edge weights of vertex j, and k_j is the weight of the edge from j to u. $|V|$ is the number of vertices in the graph, and d is a user defined *damping* coefficient to ensure fairness in computation for leaf vertices.

An intuitive way of understanding PageRank is the following. Assume a user wants to surf the Internet. She chooses one page to start randomly with some probability (initial rank), and follows the outgoing links to other pages. After a long period of time, the time spent on each web page represents its rank value, suggesting the importance of the page.

Figure 1 presents a sample PageRank implementation (in Python). It first assigns the initial rank values to all vertices in lines 3-4, and then updates the ranks inside a loop. In each iteration, the algorithm traverses all vertices, collects ranks from the parent nodes and updates the ranks accordingly (lines 16-24). After updating all the vertices, it calculates the difference between the current and the previous ranks to check if it has reached a relatively stable state, by comparing *delta* with an *epsilon* threshold (line 30). If the difference is smaller enough, the algorithm returns the ranks (line 31). Otherwise, it continues computing until the condition is satisfied or a maximum number of iterations is reached (line 7).

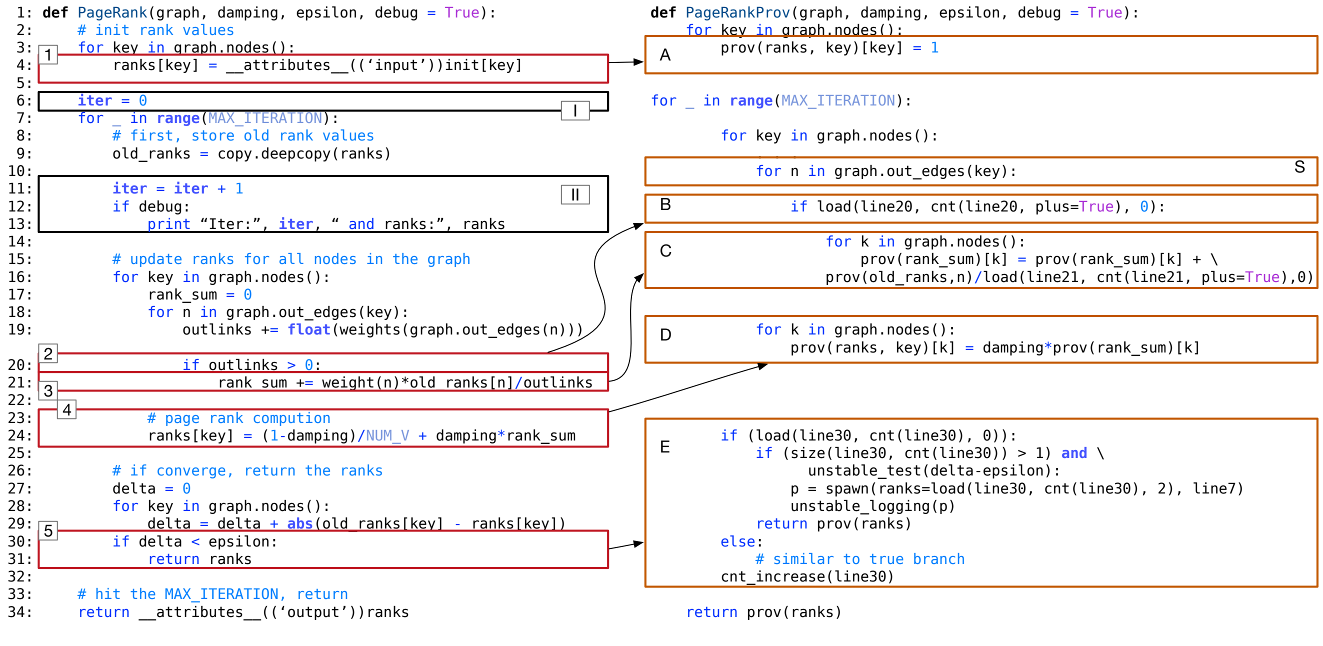
Motivation Case. We apply the algorithm to rank the accounts in the Weibo social network [7]. In the data set, a node represents an account, and an edge denotes that an account follows another account. Input edge weights are used to represent the number of actions (comment, re-tweet etc.) between followers and followees. In particular, an edge weight is calculated as the number of actions multiplied by a constant τ, and subtracted by the number of tweets for which the follower does not do anything. We cross-check the ranking results with those by other methods [22, 40]. We observe that most results are consistent and popular accounts have high ranks. In Figure 2a, we show the provenance graph for a popular account `Kai-Fu Lee`, who is an IT celebrity. The node size represents the rank value of the node, and the edge width denotes the impact of a node on another node. As we can see, many accounts, including popular ones like `NewsPlus` and unpopular ones, are following `Kai-Fu Lee` and actively commenting and retweeting his tweets. Effectively, the red node in the center (representing `Kai-Fu Lee`) has many edges of different weights from many other nodes of various sizes, which represents a typical provenance graph for popular accounts.

However, we noticed that an unknown account, `Yangqi`, is mysteriously ranked very high (i.e., within top 50), while it has less than 200 followers, far less than popular accounts. Given the complexity of the dataset, It is difficult to diagnose why `Yangqi` ranks so high, especially that his followers may have their own followers that (transitively) contribute to his rank. Thus, we employ LAMP to investigate this case. Figure 2b presents the provenance graph for `Yangqi`. Observe that it is quite different from `Kai-Fu Lee`'s graph. It suggests that the rank of `Yangqi` is highly influenced by another unpopular account `Qing Fei`. Note that the node for `Qing Fei` has a very small size but its edge is very heavy, implying that although `Qing Fei`'s rank value is small, it substantially inflates the rank of `Yangqi`. Further inspection shows that `Qing Fei` has many negative weights on its outgoing edges, which makes the sum of its outgoing edge weights, i.e., c_j in Equation (1), far smaller than the edge weight from `Qing Fei` to `Yangqi`, i.e., k_j in Equation (1). As a result, k_j/c_j is much larger than 1 such that the rank of `Yangqi`, $PR(u)$, is essentially the rank of `Qing Fei`, $PR(j)$, multiplied by a very large factor according to Equation (1). The root cause is that when the dataset was generated (by other researchers), the parameter τ was not well-defined, which has led to negative weights. To fix this problem, we redefine the value of τ to preclude negative values. The updated rank of `Yangqi` correctly represents its unpopularity. Its provenance graph is shown in Figure 2c. This case illustrates a typical fault in data engineering. More cases can be found in §7.

Table 1: Provenance by Tainting

Graph	Node	Edges	Avg	Max	Run	Mem
StanfordWeb	1,000	10,948	723	1,000	825%	33.25%
GoogleWeb	34,546	421,578	6,194	21,349	1103%	42.19%
TencentUser	10,970	170,327	1,293	8,306	1023%	48.24%
TencentMsg	83,306	2,516,122	12,342	82,194	1345%	39.25%
Twitter	96,401	482,834	32,593	72,294	1483%	29.38%

Dependency Analysis based Provenance Tracking. A traditional way of collecting provenance for programs is by tracking



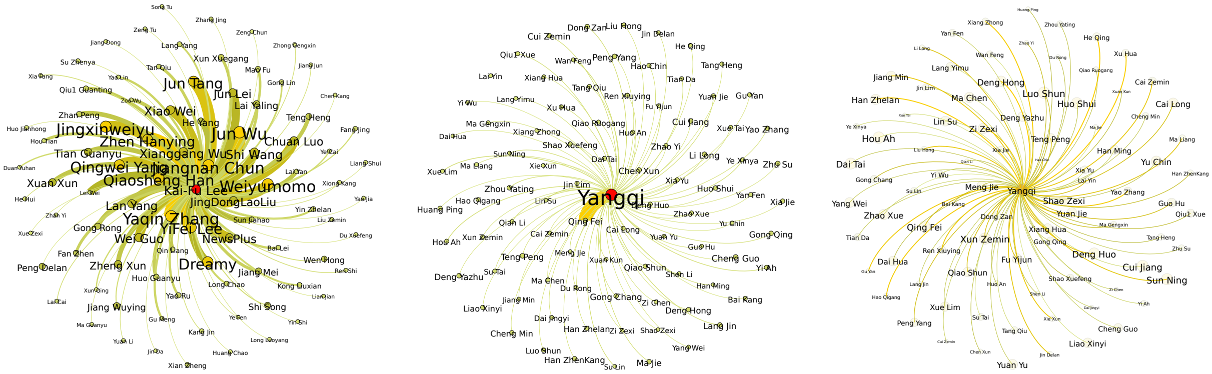
```
 1: def PageRank(graph, damping, epsilon, debug = True):
 2:     # init rank values
 3:     for key in graph.nodes():
 4:         ranks[key] = __attributes__((‘input’))init[key]
 5:
 6:     iter = 0
 7:     for _ in range(MAX_ITERATION):
 8:         # first, store old rank values
 9:         old_ranks = copy.deepcopy(ranks)
10:
11:         iter = iter + 1
12:         if debug:
13:             print “Iter:”, iter, “ and ranks:”, ranks
14:
15:         # update ranks for all nodes in the graph
16:         for key in graph.nodes():
17:             rank_sum = 0
18:             for n in graph.out_edges(key):
19:                 outlinks += float(weights(graph.out_edges(n)))
20:                 if outlinks > 0:
21:                     rank_sum += weight(n)*old_ranks[n]/outlinks
22:
23:             # page rank compution
24:             ranks[key] = (1-damping)/NUM_V + damping*rank_sum
25:
26:         # if converge, return the ranks
27:         delta = 0
28:         for key in graph.nodes():
29:             delta = delta + abs(old_ranks[key] - ranks[key])
30:         if delta < epsilon:
31:             return ranks
32:
33:     # hit the MAX_ITERATION, return
34:     return __attributes__((‘output’))ranks
```

(a) PageRank Code

```
def PageRankProv(graph, damping, epsilon, debug = True):
    for key in graph.nodes():
        prov(ranks, key)[key] = 1

for _ in range(MAX_ITERATION):

        for key in graph.nodes():

            for n in graph.out_edges(key):
                if load(line20, cnt(line20, plus=True), 0):
                    for k in graph.nodes():
                        prov(rank_sum)[k] = prov(rank_sum)[k] + \
                    prov(old_ranks,n)/load(line21, cnt(line21, plus=True),0)

            for k in graph.nodes():
                prov(ranks, key)[k] = damping*prov(rank_sum)[k]

        if (load(line30, cnt(line30), 0)):
            if (size(line30, cnt(line30)) > 1) and \
                  unstable_test(delta-epsilon):
                p = spawn(ranks=load(line30, cnt(line30), 2), line7)
                unstable_logging(p)
            return prov(ranks)
        else:
            # similar to true branch
        cnt_increase(line30)

    return prov(ranks)
```

(b) Transformed PageRank Code

Figure 1: PageRank and the corresponding transformed code



(a) Kai-Fu Lee **(b) Yangqi** **(c) Yangqi (New Data)**

Figure 2: Weighted PageRank Data Debugging Example

data and control dependencies [23]. However in GML, an output tends to be (transitively) dependent on a large set of inputs through program dependencies due to the rippling effect. Such huge provenance sets for outputs are hardly useful as the importance of individual input values cannot be distinguished. A lot of inputs in the provenance set have very little impact on the output. Furthermore, taint propagation for individual operations is usually implemented as set operations (e.g., set unions), which are very expensive on provenance sets. We run PageRank for 10 iterations on a few data sets while collecting provenance using dynamic tainting. The results are shown in Table 1, which presents the input data set (column 1), the number of vertices/edges (column 2/3) in the graph, the average/maximum size of provenance set (column 4/5), and the runtime/memory overhead (column 6/7). Observe that even though we only run it for a small number of iterations, the provenance sets are already of large size, which cause substantial overhead.

In comparison, we show the distribution of derivatives computed by LAMP when running the PageRank algorithm on the Tencent Message data set in Figure 3. The x-axis represents the logarithm of derivative value. and the y-axis represents the number of occurrences. As we can see, most derivatives are small, meaning that the rank of a vertex is mostly determined by a small number of vertices.

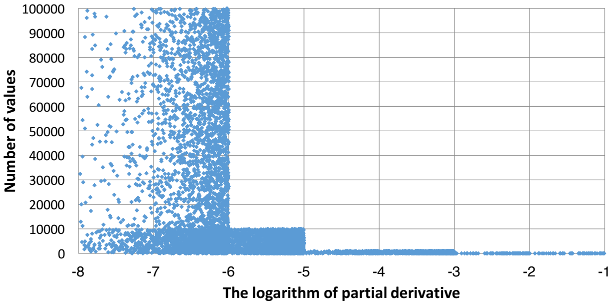


Figure 3: The distribution of impact values for the Tencent Message graph

3 PROBLEM STATEMENT

Our goal is to compute data provenance for GML programs.

Definition 1. *A Graph based Machine Learning (GML) program P takes a graph model G and its nodes' initial weight values $I = \{x_0, ...x_m\}$, where x_i is the initial weight of node i, and produces updated weight values F without changing the graph structure.*

The initial weights of a given graph's nodes may be explicitly provided as part of the input (e.g., as in Bayesian Networks [45], Belief Propagation [59]) or using a pre-defined constant (e.g., as in the original PageRank [52]). Edge weights are handled no differently from node weights as both are represented as program variables. Hence for simplicity, we only assume node weights in our discussion. LAMP can be applied to many ML algorithms that fall in our GML definition.

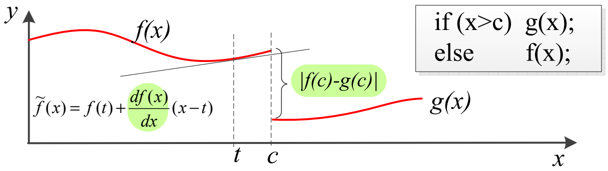
Problem Statement 1. *Consider the output weight for a node n as a function over the initial weights, denoted as $F_n(x_0, ..., x_m)$. For each node i with an initial weight x_i, and a given execution $x_0 = a_0$, ..., $x_m = a_m$, we aim to compute the partial derivative $\frac{d\ F_n(a_0,..,a_m)}{dx_i}$, if the partial function $F_n(a_0, ..., a_{i-1}, x_i, a_{i+1}, ..., a_m)$ is continuous at $x_i = a_i$. Otherwise, we compute $|F_n(a_0, ..., a_i + \epsilon, ..., a_m) - F_n(a_0, ..., a_i - \epsilon, ..., a_m)|$ with ϵ being an infinitely small value.*

Our goal is to compute the partial derivative that represents the impact of each input on each output. Intuitively, if a vertex is important (i.e., by having a high initial weight or by being connected to many other vertices), a small perturbation of its initial value will change the values of all the connected vertices and eventually lead to substantial output changes. However, while mathematical functions are largely continuous, GML programs have a lot of discrete behaviors. As a result, the F_n functions are usually discontinuous. Note that in a discontinuous function, an arbitrarily small input variation does not lead to arbitrarily small output variation. As a result, the derivative is infinite. In this case, derivatives do not represent the impact of input variations. Therefore, we report the output variations instead.

Figure 4 depicts an illustrative example. The program, presented in the small box, behaves as follows, based on the input x:

$$F(x) = \begin{cases} f(x) & x <= c \\ g(x) & otherwise \end{cases}$$

$F(x)$ is discontinuous at $x = c$. For any $x = t$ and $t < c$, LAMP computes $\frac{d\,f(t)}{dx}$, which is the slope of the tangent line to $f(x)$ at $x = t$, denoting how the output varies at the neighborhood of $x = t$. Upon $x = c$, the derivative is not informative due to the discontinuity. Hence, LAMP computes $|f(c) - g(c)|$ instead, which gauges the impact of input variation.



```
if (x>c)  g(x);
else       f(x);
```

Figure 4: Essence of LAMP

In this paper, we do not consider floating point rounding errors. Since we reason about input variations at a much larger scale compared to rounding errors, the effects of rounding errors are largely shadowed. We will leave a thorough study of the interference of rounding errors to the future.

4 OVERVIEW

Basic Idea. LAMP computes the partial derivatives for each variable on the fly. Given a statement $y = f(t_1, ..., t_m)$ with t_1, ..., t_m the operands, LAMP computes the partial derivatives of y regarding each initial weight, leveraging the derivative chain rule. It is a rule to compute the derivative of function composition:

$$\frac{d}{dx}[f(u)] = \frac{d}{du}[f(u)]\frac{du}{dx} \tag{2}$$

, where $f(u)$ is the final output, u is an intermediate result and x is the input variable. Intuitively, it says that the derivative of a function regarding its input can be computed from the derivative of the function regarding an intermediate result and the derivative of the intermediate result regarding the input. Leveraging the chain rule, derivative computation can be done locally to a statement, based on the operand values and their partial derivatives that were computed when the operands were defined. Upon a predicate, LAMP checks if a small variation to any initial weight value can cause the predicate to take a different branch outcome. This can be done by linear approximation using the computed partial derivative of the predicate expression. If so, LAMP spawns a new process to take the other branch. At the end, output variations are derived by comparing the outputs from all the processes. According to the problem statement in §3, the partial derivatives and the output variations caused by discontinuity are the resulting data provenance that gauges the impact of inputs on outputs.

Consider the code example on the left side of Figure 5. The program takes two initial weight values: x_1 and x_2. The predicate at line 4 makes the output weights discontinuous functions. The next two columns in the same figure show the conceptual provenance computation regarding x_1 and x_2, respectively. At line 1, the partial derivatives of p with regards to x_1 and x_2 are 2 and 3, respectively. At line 2, the partial derivatives of q with regards to x_1 and x_2 are 0 and $2 * x_2$, respectively. At line 3, according to the chain rule, the derivatives of r are computed from the values and the previously

Code	Derivatives w.r.t. x_1	Derivatives w.r.t. x_2
1 $p=2x_1+3x_2;$	① $p'_{x1}=2;$	$\boxed{1}$ $p'_{x2}=3;$
2 $q=x_2*x_2;$	② $q'_{x1}=0;$	$\boxed{2}$ $q'_{x2}=2*x_2;$
3 $r=p*q$	③ $r'_{x1}=p'_{x1}*q+p*q'_{x1};$ //2*x_2*x_2	$\boxed{3}$ $r'_{x2}=p'_{x2}*q+p*q'_{x2};$ //$4x_1x_2$+$9x_2x_2$
4 if $(r>100)$	④$_T$ if $(r>100$ && $r-\lvert r'_{x1}*\Delta\rvert<=100)$ /*spawn a process to execute s2*/	$\boxed{4_T}$ if $(r>100$ && $r-\lvert r'_{x2}*\Delta\rvert<=100)$ /*spawn a process to execute s2*/
5 $s1;$		
6 else $s2;$	④$_F$ if $(r<=100$ && $r+\lvert r'_{x1}*\Delta\rvert>100)$ /*spawn a process to execute s1*/	$\boxed{4_F}$ if $(r<=100$ && $r+\lvert r'_{x2}*\Delta\rvert>100)$ /*spawn a process to execute s1*/

Figure 5: Example to Illustrate the Basic Idea. Symbol p'_{x_1} denotes $\frac{d\,p}{d\,x_1}$.

```
Program     P  ::= s*
Statement   s  ::= x =ℓ e
               |   x =ℓ y op z
               |   while (x >ℓ 0) : s
               |   if x >ℓ 0 then s1 else s2
Expr        e  ::= v  | x  | x^input  | y^output
Operator    op ::= +  | -  | *  | /  | ...
Value       v  ::= {'True', 'False', 0, 1, ...}
Label       ℓ  ::= {ℓ1, ℓ2, ...}
     Variable  x, y, z ∈ Identifier
```

Figure 6: Language Abstraction

computed derivatives of p and q. Note that the symbolic value of r regarding x_1 and x_2 is $r(x_1,x_2) = 2x_1x_2^2 + 3x_2^3$, $\frac{d\,r(x_1,x_2)}{d\,x_2} = 4x_1x_2 + 9x_2^2$, which is exactly the value computed through the three steps $\boxed{1}$ – $\boxed{3}$. Upon the predicate, ④$_T$ indicates that if the original execution takes the true branch (i.e., $r > 100$) but the linear approximation of r's variation with a small variation Δ of x_1 leads to the opposite branch outcome, a new process is spawned to take the false branch so that the outputs along this different path can be computed and contrasted with the original output.

Workflow. The aforementioned procedure is just a conceptual explanation. In practice, it is too expensive to perform provenance computation during production runs. An important design choice of LAMP is hence to decouple the original computation from the provenance computation. Observe that in the provenance computation, the values computed in the original run are usually not needed. For instance, at steps ① and $\boxed{1}$, the values of p, x_1, or x_2 are not needed at all. In fact, the operand values are only needed in multiplication and division operations, which are relatively rare compared to additions and subtractions. Therefore, LAMP records the needed variable values during production runs, such as values p and q at line 3.

LAMP then transforms the original program to a new one dedicated to provenance computation, which is triggered on demand. During provenance computation, the logged values are used to avoid most of the original computation.

5 DESIGN

To facilitate discussion, we introduce a simple language in Figure 6. We use superscripts to indicate input and output variables. Note that here input variables are the initial weight values, which may be loaded from input files, or initialized to some constant (e.g., line 4 in Figure 1). Each statement is identified by a label ℓ. The language is just for discussion. Our implementation supports Python.

5.1 Run time Information Collection

During production runs, LAMP conducts very lightweight tracing to collect branch outcomes and the results of some operations (e.g., multiplications). The tracing semantics are explained in Figure 7. The expression rules are standard. According to the evaluation context E, expressions are first evaluated to values before the statement rules are applied. Statement evaluation has the configuration of σ, ω, and C: σ is the store; ω is the tracing log that consists of a sequence of trace entries, each containing a statement label, the execution counter value of the statement, and a set of values; C records the current counter value for each statement.

The evaluation rules of most statements are standard and hence elided. Rules [MUL-LOG] and [MUL-LOG-Y] indicate that LAMP may log the operand values for multiplications because such values are needed in derivative computation in the later provenance computation phase (§4). In particular, if the compiler statically determines that both operand variables y and z are related to annotated input variables, their values are logged by attaching an entry to ω. The counter is also increased. Similarly, if only one operand is input related, the other operand value is logged. When neither operand is input variable related, LAMP does not need to compute the derivatives and hence the operand values are not recorded (Rule [MUL-NoLOG]).

Upon a conditional statement, LAMP determines if the predicate is related to input variables. If so, it further detects if the branch outcome may be unstable by function *unstable*(). We say that a branch outcome is unstable if a small input perturbation Δ flips the branch outcome. Specifically, for a predicate v>0 and a related input x_i, the predicate is unstable if $|\frac{d\ v}{d\ x_i}| \times \Delta > |v|$. However during production run, we do not know the derivative. Hence, we test if v/Δ is smaller than the pre-defined maximum partial derivative. If so, the predicate may be unstable and we log the branch outcome, the value v, and the values of critical state variables (Rule [IF-UNTABLE-T]). We determine if the predicate is truly unstable during provenance computation. If so, a process is spawned to take the other branch. Logging the critical state variables is to support the child process. For example in PageRank (Figure 1), the comparison at line 30 is input related and possibly unstable when $|delta - epsilon|/\Delta$ is smaller than $MAXD$. Thus, $ranks$[] values are recorded, which are sufficient for execution along the other branch. The set of critical variables at a program point ℓ, $CS(\ell)$, is

Definitions:

$\omega \in$ Log: $\langle Label, Index, Value^{*}\rangle$ $\quad\sigma \in Store: Variable \rightarrow Value$ $\quad \mathrm{C} \in StmtIndex: Label \rightarrow Index$

$input\_rel(x)$: if x is (transitively) data dependent on any input

$CS(\ell)$: the set of critical state variables at ℓ

$unstable(v) = |v/\Delta| < MAXD$, with Δ the input variation bound and $MAXD$ the upper bound of partial derivatives. It determines if the predicate may potentially take the opposite branch in the presence of input variation.

Semantic rules:

$$\mathrm{E} ::= \mathrm{E;s} \mid [\cdot]_s \mid \mathrm{x}{=}[\cdot]_e \mid \mathrm{x}{=}[\cdot]_e \ \mathrm{op}\ e \mid \mathrm{x}{=}v\ \mathrm{op}\ [\cdot]_e \mid \mathrm{if}\ [\cdot]_e > y\ \mathrm{then}\ s_1\ \mathrm{else}\ s_2 \mid \mathrm{if\ v} > [\cdot]_e\ \mathrm{then}\ s_1\ \mathrm{else}\ s_2$$

Expression Rule: $\sigma : e \xrightarrow{e} v$

$\sigma : v \xrightarrow{e} v$ [E-CONST] $\qquad \sigma : x \xrightarrow{e} \sigma(x)$ [E-VAR]

Statement Rule: $\sigma, \omega, \mathrm{C} : s \xrightarrow{s} \sigma', \omega', \mathrm{C}', s'$

$\sigma, \omega, \mathrm{C} : x =^{\ell} v_y * v_z \xrightarrow{s} \sigma[x \rightarrow v_y * v_z],\ \omega \cdot \langle \ell, \mathrm{C}(\ell), v_y, v_z\rangle,\ \mathrm{C}[\ell \rightarrow \mathrm{C}(\ell)+1],\ \mathrm{skip}$, if $input\_rel(y) \wedge input\_rel(z)$ [MUL-LOG]

$\sigma, \omega, \mathrm{C} : x =^{\ell} v_y * v_z \xrightarrow{s} \sigma[x \rightarrow v_y * v_z],\ \omega \cdot \langle \ell, \mathrm{C}(\ell), v_y\rangle,\ \mathrm{C}[\ell \rightarrow \mathrm{C}(\ell)+1],\ \mathrm{skip}$,if $\neg input\_rel(y) \wedge input\_rel(z)$ [MUL-LOG-Y]

$\sigma, \omega, \mathrm{C} : x =^{\ell} v_y * v_z \xrightarrow{s} \sigma[x \rightarrow v_y * v_z],\ \omega,\ \mathrm{C},\ \mathrm{skip}$,otherwise [MUL-NoLOG]

$\sigma, \omega, \mathrm{C} : \mathbf{if}\ v >^{\ell} 0\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \xrightarrow{s} \sigma,\ \omega \cdot \langle \ell, \mathrm{C}(\ell), True, v, \sigma(CS(\ell))\rangle,\ \mathrm{C}[\ell \rightarrow \mathrm{C}(\ell)+1],\ s_1$
,if $v > 0 \wedge input\_rel(x) \wedge unstable(v)$ [IF-UNSTABLE-T]

$\sigma, \omega, \mathrm{C} : \mathbf{if}\ v >^{\ell} 0\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \xrightarrow{s} \sigma,\ \omega \cdot \langle \ell, \mathrm{C}(\ell), True)\rangle,\ \mathrm{C}[\ell \rightarrow \mathrm{C}(\ell)+1],\ s_1$
,if $v > 0 \wedge (\neg input\_rel(x) \vee \neg unstable(v)$ [IF-STABLE-T]

Global Rules:

$$\frac{\sigma : e \xrightarrow{e} v}{\sigma, \omega, \mathrm{C}, E[e]_e \rightarrow \sigma, \omega, \mathrm{C}, v}\ \text{[G-EXPR]} \qquad \frac{\sigma, \omega, \mathrm{C} : s \xrightarrow{s} \sigma', \omega', \mathrm{C}', s'}{\sigma, \omega, \mathrm{C}, E[s]_s \rightarrow \sigma', \omega', \mathrm{C}', E[s']_s}\ \text{[G-STMT]}$$

Figure 7: Semantic rules

pre-computed by the compiler. In our experience, *CS* sets are small in GML programs and the number of unstable predicates at run time is very small, thus the space overhead is low (§6). If a predicate is not input related or is stable, LAMP simply logs the branch outcome (Rule [IF-STABLE-T]). The branch outcomes will be reused during provenance computation to ensure the same control flow is followed.

5.2 Code Transformation

LAMP transforms the original program to a new program, which takes the original input graph and the log generated in the tracing phase, and performs provenance computation.

Figure 8 describes the set of transformation rules. A number of terms and helper functions are defined in the top of the figure. In particular, two global variables are declared in the transformed program: Γ that maps a variable to its partial derivatives (regarding input variables), and *cnt* that maps a statement to its current execution count.

Rule [T-INPUT-ASGN] specifies that for a statement (in the original program) that copies an input variable y^{input} to x, statements are added to the transformed program to set the derivative of x regarding y^{input} to 1 and the derivatives for other input variables to 0. In all the rules, the transformed statements are boxed. Note that the original assignment is precluded from the transformed program. For an addition statement, the transformed statement adds the corresponding derivatives (Rule [T-ADD]). Rule [T-MUL-Y] specifies that given a multiplication statement $x = y * z$, if only y is input related, the transformed statement computes the partial derivative of x by multiplying the derivative of y and the recorded value of z from the log ω. When both y and z are input related, the multiplication is transformed to statements that compute the derivative of x from both the derivatives and the values of y and z (Rule [T-MUL-YZ]).

When the variable in predicate is not input related, the statement is transformed to loading the branch outcome from the log (Rule [T-IF-NOINPUT]). According to Rule [T-IF-INPUT], when the variable is input related, the following statements are added to the transformed program. Line 1 tests if the recorded branch outcome is true. If so, line 2 further tests if the log entry contains additional information (i.e., $|\omega[\ell, cnt(\ell)]|{>}1$), which indicates the predicate is potentially unstable, and if a small input variation Δ induces a value change on x larger than the (recorded) value, leveraging the partial derivative. If so, the branch outcome can be flipped. Hence, LAMP spawns a process to continue execution along the other branch *in the original program* (lines 3). Before executing the branch, LAMP restores the critical state. The parent process continues the derivative computation in the true branch (line 5). Line 4 is to log the annotated input variables whose variations may flip the branch outcome, and the child process id. At the end of computation, for each input that causes unstable predicates, LAMP collects the values of an output variable z across all the associated processes with z_{max} and z_{min} the maximum and the minimum z, which denote the impact of the input on z.

Example. In Figure 1, box 1 contains a statement that copies the annotated input variables to $ranks[key]$. Following Rule [T-INPUT-ASGN], we set the derivatives of $ranks[key]$ for each input to 1, as shown in box A. We handle the addition statement in box 3 by applying Rule [T-ADD] to compute the partial derivatives for variable $rank\_sum$. For $old\_ranks[n]/outlinks$, we apply Rule [T-MUL-Y], invoking function $load()$ to read the value of $outlinks$ from the log file, and using it in the operation. The corresponding code is simplified and shown in box C. Box 4 contains a statement that includes $(1 - damping)/NUM\_V$. We apply Rule [T-MUL] to skip the computation, since it is not related to the annotated input variable, as shown in box D. Box 2 in Figure 1 shows a case where the comparison is not related to input. Note that $outlinks$

Definitions:

$\Gamma[x]$: the array storing the derivatives of x regarding inputs
$cnt[\ell]$: the execution counter for statement ℓ
$\omega[\ell, n, 0]$: the first recorded value for the log entry of the nth instance of statement ℓ
$spawn(s, \ell)$: spawn a process that first executes s and then original code starting from ℓ with the tracing semantics
$log(p, t)$: record that process p is spawned because of instability caused by input t
IV : the set of annotated input variables

Transformation Rule: $s \xrightarrow{ctx} s'$

$$x = y^{input} \xrightarrow{ctx} \boxed{for\ (t\ in\ IV)\ \Gamma[x, t] = 0;\ \Gamma[x, y^{input}] = 1} \quad \texttt{[T-INPUT-ASGN]}$$

$$x = y \xrightarrow{ctx} \boxed{\Gamma[x] = \Gamma[y]} \quad \texttt{[T-ASGN]}$$

$$x = y + z \xrightarrow{ctx} \boxed{for\ (t\ in\ IV)\ \Gamma[x, t] = \Gamma[y, t] + \Gamma[z, t]} \quad \texttt{[T-ADD]}$$

$$\frac{\neg input\_rel(y) \qquad \neg input\_rel(z)}{x =^{\ell} y * z \xrightarrow{ctx} \boxed{skip}} \quad \texttt{[T-MUL]}$$

$$\frac{input\_rel(y) \qquad \neg input\_rel(z)}{x =^{\ell} y * z \xrightarrow{ctx} \boxed{\begin{array}{l} for\ (t\ in\ IV)\ \ \Gamma[x, t] = \Gamma[y, t] * \omega[\ell, cnt[\ell], 0]; \\ cnt[\ell]++; \end{array}}} \quad \texttt{[T-MUL-Y]}$$

$$\frac{input\_rel(y) \qquad input\_rel(z)}{x =^{\ell} y * z \xrightarrow{ctx} \boxed{\begin{array}{l} for\ (t\ in\ IV)\ \ \Gamma[x, t] = \Gamma[z, t] * \omega[\ell, cnt[\ell], 0] + \Gamma[y, t] * \omega[\ell, cnt[\ell], 1]; \\ cnt[\ell]++; \end{array}}} \quad \texttt{[T-MUL-YZ]}$$

$$\frac{\neg input\_rel(x) \qquad s_1 \xrightarrow{ctx} s_1' \qquad s_2 \xrightarrow{ctx} s_2'}{if\ \ (x >^{\ell} 0)\ \ then\ \ s_1\ \ else\ \ s_2 \xrightarrow{ctx} \boxed{if\ \ (\omega[\ell, cnt(\ell)++, 0])\ \ then\ \ s_1'\ \ else\ \ s_2'}} \quad \texttt{[T-IF-NOINPUT]}$$

$$\frac{input\_rel(x) \qquad s_1 \xrightarrow{ctx} s_1' \qquad s_2 \xrightarrow{ctx} s_2'}{if\ \ (x >^{\ell} 0)\ \ then\ \ s_1^{\ell_1}\ \ else\ \ s_2^{\ell_2} \xrightarrow{ctx} \boxed{\begin{array}{l} if\ \ (\omega[\ell, cnt(\ell), 0])\ then \\ \quad if\ \ |\omega[\ell, cnt(\ell)]| > 1\ \&\&\ \exists d \in \Gamma[x]\ |d \times \Delta| > |\omega[\ell, cnt(\ell), 1]|)\ then \\ \qquad p = spawn(CS[\ell] = \omega[\ell, cnt(\ell), 2], \ell_2); \\ \qquad for\ (t \in IV\ s.t.\ \ |\Gamma[x, t] \times \Delta| > |\omega[\ell, cnt(\ell), 1]|)\ \ log(p, t); \\ \quad s_1'; \\ else\ \ .../*!\omega[\ell, cnt(\ell), 0]*/ \\ cnt(\ell)++; \end{array}}} \quad \texttt{[T-IF-INPUT]}$$

Figure 8: Transformation Rules

is a computed value, but not from annotated variables. Thus we apply Rule [T-IF-NOINPUT] to get the branch outcome from the log (box B). In box 5, *delta* is computed from *ranks*, which has data dependencies on the annotated inputs. We thus apply Rule [T-IF-INPUT]. We check if the log entry contains information additional to the branch outcome. If so, the predicate instance was determined as potentially unstable during the production run. We use function *unstable_test* to check if ($delta - epsilon$) has a partial derivative that is large enough such that a small input variation can flip the branch outcome. If so, function *spawn*() spawns a child that first loads the values of *ranks* from the log and then executes from line 7. The corresponding simplified code is shown in box E. At the end of all processes, the differences of their *ranks* arrays are computed as the observed output variations. ⊡

5.3 Discussion

Property 1. *If an output is a continuous function of an (annotated) input variable within a range, the partial derivative computation of LAMP is precise in the range.*

For example in Figure 4, the derivative computation in the ranges $x < c$ and $x > c$ is precise. This is because LAMP strictly follows the mathematical rules of derivative computation. However, since LAMP does not model derivative variation with regard to input variation. As such, if a derivative varies substantially (e.g., when an output function oscillates rapidly), the derivatives computed by LAMP may not be a good indicator for input impact. In practice, most GML algorithms are iterative algorithms that have very slow derivative variation, as supported by our experiments in §6.

Property 2. *Assuming the variation of an output function $f(x)$ within an input variation bound of $(0, \Delta)$ is bounded by ϵ. In the presence of discontinuity (caused by control flow), the error of data provenance computed by LAMP is bounded by ϵ.*

Recall upon discontinuity, LAMP computes output variations, instead of derivatives (§3). Take Figure 4 as an example. At $x = c$, ideally LAMP should compute $f(c) - g(c)$. However, according to our semantics, when $x \in (c, c + \Delta)$, the branch outcome may be flipped, LAMP hence computes $f(x) - g(x)$. Note that although $f(x)$ is undefined in $(c,\ c + \Delta)$ (in the original program), LAMP essentially approximates it in this range by spawning a process to take the else branch. According to our assumption, the computed $f(x) - g(x)$ has a bounded error when compared to $f(c) - g(c)$.

5.4 Optimization

Copy-on-write: Our derivative computation semantics dictates that LAMP may need to copy the Jacobian matrices, which could be very large. For instance, an 18 MB GraphML file (10970 nodes and 170327 edges) may require 10GB memory in copying the Jacobian matrix. For instance, If we directly transform the PageRank program Figure 1, we will need to create another matrix to keep *old_ranks* with the same size as the computed Jacobian matrix, which is far larger than the original graph matrix. In fact, for an 18 MB GraphML file (10970 nodes and 170327 edges), keeping a copy of the Jacobian matrix and results requires about 10GB memory space. Besides, performing a deep copy on the matrix is also very expensive. Inspired by the copy-on-write strategy in the operating system design, we use copy-on-write for Jacobian matrices. In particular, LAMP does not copy any data until two copies begin to differ, i.e. data entries are modified. As the matrix is large, we only keep records of the different parts.

Dynamic allocation: For a strongly connected graph, the Jacobian matrix is a dense matrix as each node will depend on almost all other nodes. For a star-like graph, the Jacobian matrix is sparse with a dense row/column. If most of the nodes are isolated, the Jacobian matrix is a sparse matrix. Maintaining a full Jacobian matrix (with lots of zeros) may be inefficient. We tackle this problem by dynamically allocating space to store only non-zero values.

For example, lines 3-4 in Figure 1 assign deterministic values to the variable *ranks*. By following the original transformation rule, we initialize the Jacobian matrix with all values set to 0. Now, under the dynamic allocation approach, we only create the variable to store the Jacobian matrix and do not store the 0 values until updated.

Zeroing out minor values: The rippling effect dictates that many vertices have negligible impacts, reflected by very small derivative values. Such small impacts can only degrade over time. To avoid computing and maintaining such small derivatives, we reset them to 0 if they are small than a threshold. To support such a mechanism, LAMP takes a user specified threshold to determine if the impact is large enough to be stored. If the impact on a target vertex is smaller than the threshold, LAMP assumes that a dependency does not exist and thus the partial derivative is set to be 0.

Trace Compression: We use running encoding to compress multiple identical trace entries to a number. This is particularly effective in compressing branch outcomes. As discussed in §5, during the execution of the original program, we need to log some values in order to avoid extra computations in the provenance analysis. While the concrete values (e.g. Rule [S-MUL-LOG]) have to be logged, we do not need to log every result of the predication (e.g. Rule [S-IF-T-LOG]) for optimization purposes. To illustrate this, line 30 in Figure 1 shows an if-statement that checks whether the results have converged. Since the program converges only once, the if-statement will return *False* in all cases except in the last iteration. We leverage such an observation to reduce the logging overhead. We log the predicate result when the program first hits one predicate, however; for subsequent hits, we write a new log only when the predicate result is different from the previous one. This approach entails caching a few values (2 for Figure 1), and avoids logging most entries.

6 EVALUATION

LAMP is implemented on [1], a python analysis platform we developed which can perform static analysis and instrumentation. We have also developed a number of optimizations such as using copy-on-write to store derivatives and zeroing out derivatives that are trivial. In this section, we report the evaluation results regarding efficiency and effectiveness. In addition, we provide three case studies to show how the computed provenance can be used in problem diagnosis. All experiments were conducted on a machine with 4 cores and 64 GB memory.

Table 2: Code Size of Transformed Program

Algorithms	Original	Transformed
PageRank [52]	144	171
Weighted PageRank [57]	168	182
Undirected PageRank [34]	142	170
Visit of Links based PageRank [44]	224	242
Visit of Links based Weighted PageRank [56]	256	270
Personalized PageRank [20]	152	184
Collusionrank [27]	124	135
SimRank [39]	111	158
ASCOS [21]	126	172
TextRank [50]	348	382
Belief Propagation [59]	384	462
Gibbs Sampling [25]	265	296

Table 2 shows the algorithms list used in our evaluation. We have 12 real world GML programs. Weighted PageRank computes rank values in weighted graphs; Visit of Links based PageRank requires the visit information to rank web pages; Personalized PageRank requires a user input vector (i.e., user preference) to calculate the rank values for individual users. SimRank and ASCOS calculate the similarity of two nodes in a graph with different methods. Belief propagation inferences the marginal distribution for unobserved nodes, conditional on observed nodes. Gibbs sampling is a well-known Markov Chain Monte Carlo (MCMC) sampling method. The 2nd and 3rd columns show the sizes of the original program and the transformed program. Note that although these programs are not large, they are the typical GML programs used in practice. For example, popular graph analysis tools (e.g., NetworkX [9], graph-tool [6]) implement PageRank in less than 80 LOCs. Our adaption of GML programs (of such sizes) as benchmarks is also consistent with experiment setup in the literature (e.g. [48, 49, 58, 61]). The common use pattern in data engineering tends to have small programs and large data sets.

6.1 Efficiency

Run Time: Different programs may require data sets in different formats. For 6 programs, we used the same 14 real world data sets. For the remaining programs, we used 7 other data sets. The data sets are acquired from public sources (e.g. Stanford Large Network Dataset Collection [46] and MovieLens [35]) or crawled by Scrapy [12]. The sources and data set sizes (number of vertices/number of edges, and text file size for Gibbs Sampling) are reported in Table 3. Some data sets (e.g., Tencent Messages) are huge so that we can only use part of them due to our memory capacity. Here, we only report the overhead of provenance computation. The tracing overhead is less than 1% for all the programs due to the limited instrumentation and the small logs generated. As shown in Table 3, provenance computation takes 2 to 3 times the original computation time. LAMP's provenance computation is optimized. The optimizations reduce the overhead by an average factor of 3. While we consider the overhead reasonable, it cannot be afforded during production runs. This strongly supports our design that separates the original computation from the provenance computation.

Table 3: Provenance Computation Overhead

Algorithm	Web Graph-1	Tencent Message	Cit-HepPh	Tencent User	ego-Twitter	p2p-GnuTella08	Wikipedia Link
	1K/1M	382K/13M [16]	34K/421K [46]	438K/6M [16]	81K /2M [46]	6K/21K [46]	4K/823K [17]
PageRank	372.24%	121.06%	274.81%	126.98%	298.34%	309.38%	194.20%
Per PageRank	322.73%	182.23%	276.39%	123.48%	248.39%	327.38%	231.28%
Collusionrank	328.48%	128.28%	263.48%	138.56%	192.39%	382.47%	183.38%
SimRank	273.38%	124.68%	251.37%	183.45%	183.48%	325.27%	182.84%
ASCOS	273.48%	198.28%	174.27%	184.53%	263.59%	385.73%	108.83%
Belief Prop	283.34%	201.28%	273.47%	128.38%	294.38%	294.58%	192.38%
Algorithm	Twitter	p2p-GnuTella09	wiki-Vote	MovieLen 10M	web-Google	soc-Slashdot0922	email-EuAll
	696K/2M [27]	8K/26K [46]	7K/104K [46]	72K/10M [35]	876K/5M [46]	82K/948K [46]	265K/420K [46]
PageRank	121.06%	284.38%	273.27%	185.37%	284.73%	284.92%	283.25%
Per PageRank	204.37%	274.37%	239.37%	183.58%	274.85%	238.26%	294.38%
Collusionrank	197.35%	263.46%	263.84%	183.38%	385.27%	304.28%	301.58%
SimRank	194.35%	263.23%	273.38%	174.28%	302.38%	329.38%	318.85%
ASCOS	237.75%	284.58%	284.37%	173.58%	326.48%	329.48%	333.62%
Belief Prop	206.58%	274.38%	284.58%	184.27%	385.27%	274.59%	321.38%
Algorithm	LDA Sample-1	LDA Sample-2	LDA Sample-3	LDA Sample-4	LDA Sample-5	LDA Sample-6	LDA Sample-7
	121M	143MB	147MB	100MB	137MB	125MB	134MB
GibbsSamp	123.29%	128.34%	124.56%	119.35%	120.35%	118.23%	116.23%
Algorithm	Web Graph-1	Web Graph-2	Web Graph-3	Web Graph-4	Web Graph-5	Web Graph-6	Web Graph-7
	1,000/998,996	5,000/723,283	2,283/374,382	1,927/428,283	6,293/823,283	8,238/823273	7,238/824,837
VOLPRank	293.56%	192.42%	246.33%	364.21%	352.26%	314.27%	356.25%
WVOLPRank	325.42%	243.56%	236.43%	326.47%	295.63%	324.36%	316.32%
Algorithm	MovieLen 100k	MovieLen 20M	MovieLen 1M	MovieLen 10M	ego-Facebook	ego-Amazon	com-DBLP
	1K/100K [35]	138K/20M [35]	6K/1M [35]	72K/10M [35]	4K/88K [46]	335K/924K [46]	317K/1M [46]
UndirPRank	183.85%	173.47%	173.24%	173.85%	364.28%	285.37%	372.38%
SimRank	124.38%	137.53%	127.38%	136.43%	385.28%	295.37%	336.49%
Collusionrank	173.43%	174.28%	163.48%	172.48%	382.59%	264.69%	317.48%

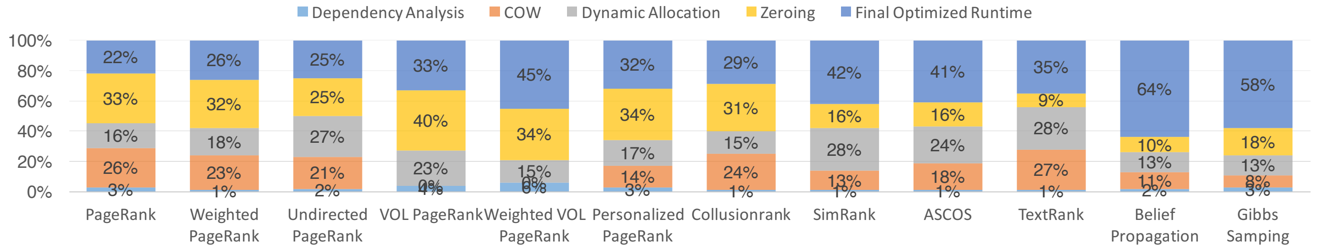


Figure 9: Optimization effects on different algorithms

In Figure 9, we show the run time effects of various optimizations discussed in §5.4. We use the computation time without optimization as the base line (100%), and then measure the percentage of computation time reduction by each optimization. For example, adopting copy-on-write on PageRank reduces the computation time by 16% (the gray part in the bar). Observe that the effects of applying different optimizations vary across benchmarks. For instance, VOL PageRank does not benefit from copy-on-write, since its implementation relies only on a maximum number of iterations to terminate and does not copy the old matrix in any case. In general, the optimization reduces half or more of the original program run time overhead.

Memory Consumption: We measure the memory consumption of provenance computation using Python memory profiler [11] that samples every 0.1 second. Figure 10 presents the results. As illustrated, the memory consumption follows a similar trend. In the first few minutes, it grows very fast. This is attributed to two reasons.

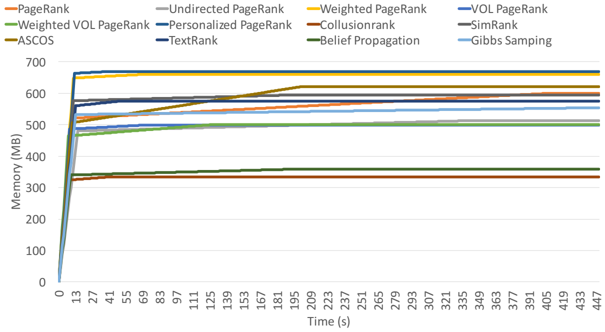


Figure 10: Memory usage of provenance computing with different algorithms

First, the process first needs to load the whole graph. Second, during the first a few iterations, the number of inputs related to a variable rapidly grows. After this, For most programs, our optimizations can reduce the memory consumption by a factor of 2.

In Figure 11, we show the effects of applying optimization techniques on the memory consumption. For these algorithms, we show the maximal memory consumption during the execution for using different techniques. For most of the programs, we can reduce the memory consumption by a factor of 2. We also demonstrate in Figure 11 the effect of optimization on the memory consumption of LAMP. We run the provenance computation on various algorithms and show the maximum memory consumption while using different techniques.

Log Space: As LAMP needs to log some variables and branch outcomes, we also measure the log size. For the data sets used (with millions nodes and edges), the log size ranges from 3-7KB. A key reason is that the running encoding optimization is very effective in compressing branch outcomes. The results are shown in Table 4. Column 2 shows the number of data sets we tested on. Column 3 and 4 show the maximum number of nodes and edges in our tested data set, and column 5 shows the corresponding log file size.

Table 4: Storage Overhead

Algorithms	#Datasets	# Nodes	# Edges	Log(KB)
PageRank	20	875,713	5,105,039	4
Weighted PageRank	7	334,863	925,872	6
Undirected PageRank	15	36,692	183,831	4
VOL PageRank	8	34,546	421,578	7
Weighted VOL PageRank	8	34,546	421,578	3
Personalized PageRank	20	875,713	5,105,039	4
Collusionrank	20	875,713	5,105,039	4
SimRank	20	36,692	183,831	5
ASCOS	20	875,713	5,105,039	3
TextRank	20	36,692	183,831	5
Belief Propagation	8	875,713	5,105,039	5
Gibbs Sampling	8	File size	121MB	4

As shown, the log files are less than a few KBs, thus can be cached in memory and would not require to be written to disk frequently. This also explains why LAMP incurs a low run time overhead as writing to disk is avoided. The low storage is attributed to that we use running encoding to compress consecutive branch outcomes to a number.

Comparison with Tainting: To demonstrate the advantage of LAMP over tainting, we compare the efficiency of the two techniques. Our experiments show that the average runtime and space overheads for tainting are 1133.17% and 41.16%, whereas those for LAMP are 250.16% and 19.71%, respectively. This is because tainting has to manipulate large sets. The concrete results are shown in Table 5. The first column shows the algorithms, the second and third columns show the comparison of runtime for the two approaches, and the last two columns show the memory consumption of the two approaches.

Unstable Predicates: The stability of predicates varies according to the input variation threshold Δ. We conduct experiments with Δ = 10%, 20%, and 40%. We observe totally 2, 8 and 20 forks, respectively, for our test data sets. For a single execution, the maximum number of forks we observe is 2. Most cases have only one fork. Recall that LAMP forking a child process means that the input variation flips a branch outcome, leading to output variations that cannot be described by derivatives.

Table 5: Comparison of LAMP and tainting

Algorithm	Run Time		Memory	
	LAMP	Tainting	LAMP	Tainting
PageRank	121%	1345%	17.28%	39.25%
Weighted PageRank	235%	964%	16.63%	42.35%
Undirected PageRank	364%	1486%	24.85%	68.23%
VOL PageRank	294%	1028%	21.84%	36.35%
Weighted VOL PageRank	244%	1123%	19.24%	34.27%
Personalized PageRank	276%	1024%	12.28%	28.25%
Collusionrank	139%	917%	15.27%	36.42%
SimRank	183%	824%	25.37%	53.26%
ASCOS	386%	1332%	13.26%	32.46%
TextRank	354%	1283%	22.57%	46.34%
Belief Propagation	283%	1294%	26.32%	42.24%
Gibbs Sampling	123%	978%	21.56%	34.48%

6.2 Effectiveness

Table 6: Distribution of partial derivatives

Algorithm	$>10^{-3}$	$>10^{-4}$	$>10^{-5}$	$>10^{-6}$	$<10^{-6}$
PageRank	0.08%	0.52%	1.40%	6.00%	92.00%
Weighted PageRank	0.04%	0.86%	2.10%	8.00%	89.00%
Undirected PageRank	0.03%	0.57%	1.40%	4.00%	94.00%
VOL PageRank	0.06%	0.54%	1.40%	2.00%	96.00%
WeightedVOLPageRank	0.03%	0.57%	1.40%	8.00%	90.00%
Personalized PageRank	0.04%	0.56%	1.40%	7.00%	91.00%
Collusionrank	0.08%	0.52%	1.40%	5.00%	93.00%
SimRank	0.05%	0.55%	1.40%	11.00%	87.00%
ASCOS	0.05%	0.55%	1.40%	9.00%	89.00%
TextRank	0.07%	0.53%	1.40%	7.00%	91.00%
Belief Propagation	0.07%	0.63%	1.30%	10.00%	88.00%
Gibbs Sampling	0.07%	0.63%	1.30%	8.00%	90.00%

We study the distribution of the derivatives for all the data sets. Our results show that only 0.03−0.08% of derivatives are larger than 10^{-3}, 0.52 − 0.86% larger than 10^{-4}, and 87 − 96% are smaller than 10^{-6}. The results are reported in Table 6. The results are averaged over all the data sets. we calculate all the partial derivatives for different algorithms, and show their distribution in Table 6. The first column lists the name of the algorithms, and the rest columns show the value ranges. We partition the derivative values into 5 categories based on the value ranges. , values that are larger than 10^{-3} (column 2), values between 10^{-3} and 10^{-4} (column 3), 10^{-4} and 10^{-5} (column 4), 10^{-5} and 10^{-6} (column 5), and values that are smaller than 10^{-6} (column 6). As shown in the table, Observe that most values are smaller than 10^{-6} and less than 0.1% of the values are larger than 10^{-3}. This clearly indicates that most of the related inputs are insignificant. Unfortunately, traditional tainting based approaches would report all these insignificant inputs.

We also perform another experiment to validate the correctness of the computed derivatives. For each data set, we randomly select an input whose derivative is larger than 10^{-3}. We then mutate the input value by 10%, run the program again with the mutated data, and measure the output differences. We then compute the *observed derivative* as the ratio between the observed output difference and the input variation, and compare it with the reported derivative by LAMP. We repeat it 500 times (i.e., randomly selecting 500 different inputs) for each data set and report the average. The results are

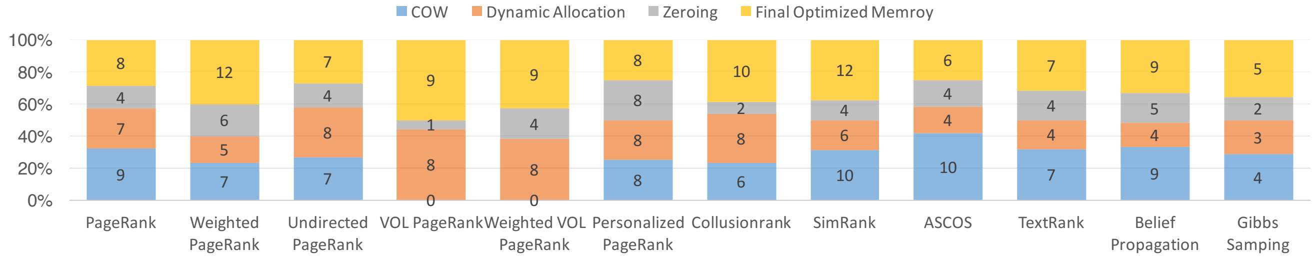


Figure 11: Memory usage of applying different optimization techniques

reported in Table 7. The 3rd column reports The last column reports the standard deviation. The table lists the name of the algorithm, the graph we use, and also the maximum difference of all 500 experiments. The table shows the name of the algorithm, the graph we use, and also the maximal difference of all 500 experiments. The differences are very small, indicating that the derivatives by LAMP can precisely measure input importance. This supports our assumption that derivatives change slowly with input changes (§5.3).

Table 7: Correctness Verification

Algorithm	Data Set	Avg	STDEV
PageRank	Tencent Message	1.02E-05	6.42E-07
Weighted PageRank	MovieLen	1.23E-06	3.42E-07
Undirected PageRank	ego-Facebook	1.64E-06	5.80E-08
VOL PageRank	Web Graph-1	1.32E-05	3.42E-06
Weighted VOL PageRank	Web Graph-2	6.32E-07	7.23E-08
Personalized PageRank	Cit-HepPh	1.34E-06	4.32E-07
Collusionrank	Tencent User	1.42E-06	3.23E-07
SimRank	ego-Vote	1.32E-05	5.32E-07
ASCOS	p2p-GnuTella08	1.78E-06	2.12E-07
TextRank	Wiki	3.25E-07	5.42E-08
Belief Prop	Tencent User	1.38E-06	3.21E-07
Gibbs Sampling	LDA Sample-1	2.64E-06	4.32E-07

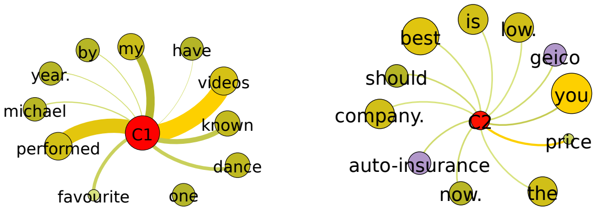
7 CASE STUDIES

In this section, we provide a few case studies to demonstrate how LAMP can help programmers in data engineering.

7.1 Model Evolution

ML models are often generated in an incremental fashion due to the cost of training and the availability of new training data. We use a Bayesian classifier [4] to demonstrate how LAMP can facilitate data engineers in this process. The classifier implements Paul Graham's algorithm [10] to classify spam comments. The original model was trained on 30,000 manually labeled YouTube comments. Given a new comment, it outputs a score (between 0 and 1) to predict if the comment is a spam. The score is computed from the scores of individual words in the comment. For example, the word `click`, which appears frequently in spam comments and rarely in benign ones, strongly hints a spam comment. We applied the original model [4] to a data set collected from other YouTube videos and observed many mis-classifications. We show two examples in the following.

- C1 (Benign): `Billie Jean, this is one of the my favorite videos I have ever seen this year. It is performed by Michael Jackson. The dance is known as the Moonwalk.`



(a) C1 (b) C2

Figure 12: Provenance graphs for C1 and C2

- C2 (Spam): `Geico is the best auto-insurance company. The price is low. You should apply now.`

The computed score of C1 is 0.999721671845, indicating a spam. But it is a false positive. We generate its provenance graph Figure 12a using LAMP. The size of a node indicates the probability of the word/comment being spam, while the weight of an edge represents the partial derivative. As shown in the graph, C1's score is determined by 11 words. It is strange that the word `videos`, which is very common in benign YouTube comments, has a high score and is the most influential node in the classification decision. Moreover, the stop word `my` is the 3rd most influential word while we expected such words should have been filtered out.

The classifier produces a score of 0.129468956748 for C2, which is a false negative. The provenance graph in Figure 12b indicates that the words `Geico` and `auto-insurance` are assigned a predefined score value of 0.4, which the model assigns to any word not appearing in the training set. However, we argue that the default value undermines the influence of these two words, which are strong indicators of spam. In fact, when cross-checking the words used in benign and spam comments with the most commonly used 5,000 English words, we find that spam comments are more likely to use uncommon words such as company names.

Table 8: Models for spam comments detection

	Original		New		Total
	Spam	Benign	Spam	Benign	
Spam	523	214	724	13	737
Benign	2026	2902	23	4905	4928
Total	2549	3116	747	4918	5665

Based on the above analysis, we improve the model by adding a pre-processing step to filter highly common words and stop words, and initializing the default score of unknown words to 0.8. Table 8 shows the performance before and after our improvement. The first row reads as follows: 523 spams are classified as spam and 214 spams are classified as benign by the original model, and the numbers become 724 and 13, respectively by the new model. Observe that the new model has much smaller false positive and false negative rates. We also use LAMP on a few other public spam filtering models to improve performance. The results are shown Table 9. For these models, we can improve their accuracy from 80%- to 90%+.

Table 9: Model Evolution

Model	Accuracy		Model	Accuracy	
	Original	New		Original	New
SpamFilter [14]	78.46%	96.27%	BayesSpam [3]	72.34%	94.32%
NBSF [15]	74.82%	98.28%	BayesianFilter [5]	57.23%	92.35%
AntiSpam [2]	69.23%	96.37%	SMS Filter[13]	60.27%	98.27%

7.2 Model Error Debugging

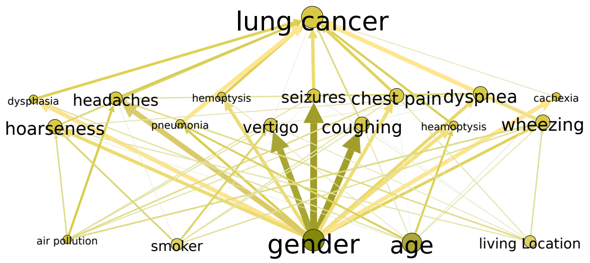
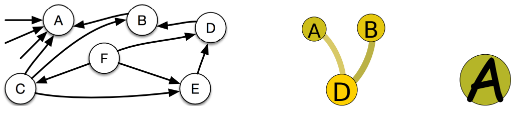


Figure 13: Provenance for model bugs

Bayesian networks are widely used in decision making such as diagnosing cancer [26, 41, 45, 55]. In this case, we use a lung cancer diagnosing network [55] with 413 nodes to demonstrate how LAMP can be used in debugging faults in models. We inject a fault to the model by changing a few conditional probabilities related to gender (e.g., $P(coughing|gender = \text{male}, \ldots)$ from 0.01 to 0.9). We then apply the model to a public cancer data set [8]. We encounter a number of misdiagnosis cases with the faulty model. Note that due to the model complexity, it is really difficult to spot the fault by inspecting the model. For instance, the `lung cancer` node is connected to 64 nodes, which are further connected to 278 nodes and `gender` is one of these nodes. We then use LAMP to generate the provenance for the misdiagnosis in Figure 13. From the graph, the decision is evenly attributed to many nodes. Nothing seems suspicious. But when we further investigate the provenance of the nodes connected to `lung cancer`, we find that a few of them have (unexpectedly) heavy influence from `gender`, indicated by the thick arrows, in comparison with other second layer nodes such as `age`. This suggests that the faulty conditional probabilities of `gender` (transitively) lead to the wrong decision. We inject 12 other bugs into the model by changing the network structure or modifying the probabilities for other nodes/edges. The provenance graphs by LAMP are always able to point to the faulty places.

7.3 Debugging GML Implementation



(a) Network (b) B (c) A

Figure 14: PageRank Debugging Example

Debugging ML programs can be hard because sometimes, we do not have clear constraints to determine if the returned results are correct. In this case, we show how LAMP is used to discover that a popular Python PageRank implementation on Github (with 50+ forks and 60+ stars) is buggy. While the implementation produces very reasonable ranking results for some popular data sets, we observed some seemingly incorrect results on some of our own data sets. We apply LAMP to further analyze the suspicious results. Figure 14a shows an example input network connection graph. The PageRank implementation generates a very low rank for node A, which does not seem right. The provenance for A's rank (Figure 14c) indicates that other nodes have no impact on A, which is buggy. In contrast, Figure 14b shows that node D has influence from nodes A and B, but not from C, E or F, to which it is also connected. From the two provenance graphs, it becomes clear that the algorithm fails to consider incoming edges in rank computation. It hence does not work properly for directed graphs. We reported the bug to the developer.

Here we will show some other bugs reported in StackOverflow or Github, and use them to demonstrate how LAMP can help identify the root causes.

Bug 1: In this report, the user is writing a PageRank algorithm. But she found that the program does not converge. We run the program with LAMP, and terminate the program after running for a while. Then, we generate the provenance graph for a desired output node, and we find that the node is connected with many nodes with unknown labels. By further investigation, we find that this is because when the algorithm is using a wrong graph. When one node tries to get rank data, it is fetching data from the wrong data structure. Thus all the dependencies are wrong in this graph.

Bug 2: The user is trying to implement the `TextRank` algorithm, and she was trying to re-use the existing NetworkX package to construct the graph. After running the algorithm, she found that the results are wrong. We reproduced this process with her code, and apply LAMP on it. The results show that many of the desired dependencies are missing in the graph. And further investigate the problem, we find that this is because the user is using a direct graph constructor to generate the graph, and the `TextRank` algorithm requires undirected graphs. Thus most of the edges just have a one-direction relationship, but missed the other one. To fix this problem, we add one line to convert the directed graphs to undirected graphs by adding edges with reversed direction.

Bug 3: This is a reported bug for the Bayespy package, which is a widely used Python package for applications using the Bayesian theorem. The user found that many of the output results are `nan`. We reproduced the bug, and generate the provenance graphs for these values. By using the layered provenance graph, we find that the original variable that takes this value does not have any provenance. This means that it is not initialized, which by default will have the default value and propagate it through computation and leads to the nan output in the final results.

Bug 4: In this post, the user gets undesired results from her algorithm, and as usually the computation involves many steps of computing and a large amount of data. Traditional approaches will not help a lot. We apply LAMP on the same algorithm and data set to help investigate the problem. The output provenance graph is smaller than desired, and many of the dependencies are missing. By further investigation, we find that this algorithm requires a normalization step, which will lead to a large provenance graph. The missing edges and nodes in the provenance graph is because the implementation is missing such a normalization step.

Bug 5: In Github, one user reported that the Pgmpy package is giving weird results from the Bayesian Model. The user uses a Bayesian network to demonstrate this. We apply the same network on this model with provided code, and use LAMP to generate the corresponding provenance graph. We find that most of the dependencies are wrong in the graph. Similar to the PageRank case, the dependencies are in a reversed fashion: independent variables are connected while the dependent variables are not. The developers of this project confirmed this bug and verifies our observation.

8 RELATED WORK

Automatic Differentiation (AD): LAMP is inspired by AD, but differs from AD. First, AD [18, 19, 24, 30–33, 43, 53, 63] computes derivatives alongside with the original computation whereas LAMP decouples the two so that provenance computation can be activated on demand. Second, AD cannot reason about output variations caused by control flow differences while LAMP can. Third, AD typically computes derivatives regarding some inputs but LAMP considers all inputs. LAMP also has a number of optimizations specific to provenance computation.

Machine Learning Algorithms Provenance: In this paper, we focus on GML algorithms. For many other ML algorithms, e.g. decision trees, the model itself reflects the dependence relationships. Gleich et al. [28] studied the PageRank algorithm sensitivity with respect to the damping parameter. A few projects study the behavior of various PageRank algorithms for multiple values of the damping parameter [29, 42]. Zhang et al. [62] infer spam pages by investigating PageRank with different damping parameters. Their argument is that spam pages should be sensitive to a given damping parameter, thus changing it will disclose them.

Other approaches such as [37, 38, 47, 51] aim to support data provenance in DISC systems. For example, [37] provides a general wrapper for MapReduce jobs providing data provenance capabilities. Matteo et al. [38] proposed Titian, a general provenance collection system for the Spark system, that enables data scientists to interactively trace through the intermediate data of a program execution and identify the input data at the root cause of a potential outlier or bug. While the intermediate results define the provenance for Titian, LAMP calculates the partial derivative as provenance, which can quantitatively measure the output sensitivity with regards to the input and produce precise and succinct dependence relationships with low overhead. These proposed approaches are system specific. LAMP provides a solution that is not bound to any data processing systems.

9 CONCLUSION

In this paper, we propose LAMP, a data provenance computation technique for GML algorithms. It features the capability of quantifying input importance. It is inspired by AD techniques and goes beyond them, by decoupling derivative computation from the original computation and supporting control flow path variations. The experimental results show that LAMP is much more efficient and effective than program dependence tracing based techniques. The results by LAMP are used in debugging a widely used PageRank implementation and detecting spam networks.

REFERENCES

[1] Anonymized for submissioin.
[2] *Anti Spam.* https://github.com/dinever/antispam.
[3] *Bayes Spam.* https://github.com/SunnyMarkLiu/NaiveBayesSpamFilter.
[4] *Bayesian classifier.* https://github.com/browning/comment-troll-classifier.
[5] *Bayesian Spam Filter.* https://github.com/aashishsatya/Bayesian-Spam-Filter.
[6] *Graph Tool.* https://graph-tool.skewed.de.
[7] *KDD Cup 2012 Track1.* https://www.kaggle.com/c/kddcup2012-track1.
[8] *National Cancer Database - American College of Surgeons.* https://www.facs.org/quality%20programs/cancer/ncdb.
[9] *NetworkX.* https://networkx.github.io.
[10] *A Plan For Spam.* http://www.paulgraham.com/spam.html.
[11] *Python Memory Profiler.* https://pypi.python.org/pypi/memory_profiler.
[12] *Scrapy.* https://scrapy.org/.
[13] *SMS Spam Filter.* https://github.com/revantkumar/SMS-Spam-Classifier.
[14] *Spam Filter.* https://github.com/jameshwang2013/SpamFilter.
[15] *Spam Filter.* https://github.com/SunnyMarkLiu/NaiveBayesSpamFilter.
[16] *Tencent 2012 KDD Cup.* http://www.kddcup2012.org/.
[17] *Wikipedia Data Set.* https://dumps.wikimedia.org/enwiki/20160601/.
[18] Joel Andersson, Johan Åkesson, and Moritz Diehl. 2012. CasADi: A symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation.* Springer, 297–307.
[19] Christian Bischof, Lucas Roh, and Andrew Mauer-Oats. 1997. ADIC: an extensible automatic differentiation tool for ANSI-C. *Urbana* 51 (1997), 61802.
[20] Soumen Chakrabarti. 2007. Dynamic personalized pagerank in entity-relation graphs. In *Proceedings of the 16th international conference on World Wide Web.* ACM, 571–580.
[21] Hung-Hsuan Chen and C Lee Giles. 2013. ASCOS: an asymmetric network structure context similarity measure. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining.* ACM, 442–449.
[22] Tianqi Chen, Linpeng Tang, Qin Liu, Diyi Yang, Saining Xie, Xuezhi Cao, Chunyang Wu, Enpeng Yao, Zhengyang Liu, Zhansheng Jiang, and others. 2012. Combining factorization model and additive forest for collaborative followee recommendation. *KDD CUP* (2012).
[23] Juan José Conti and Alejandro Russo. 2010. A taint mode for python via a library. In *Nordic Conference on Secure IT Systems.* Springer, 210–222.
[24] David A Fournier, Hans J Skaug, Johnoel Ancheta, James Ianelli, Arni Magnusson, Mark N Maunder, Anders Nielsen, and John Sibert. 2012. AD Model Builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software* 27, 2 (2012), 233–249.
[25] Edward I George and Robert E McCulloch. 1993. Variable selection via Gibbs sampling. *J. Amer. Statist. Assoc.* 88, 423 (1993), 881–889.
[26] Olivier Gevaert, Frank De Smet, Dirk Timmerman, Yves Moreau, and Bart De Moor. 2006. Predicting the prognosis of breast cancer by integrating clinical and microarray data with Bayesian networks. *Bioinformatics* 22, 14 (2006), e184–e190.
[27] Saptarshi Ghosh, Bimal Viswanath, Farshad Kooti, Naveen Kumar Sharma, Gautam Korlam, Fabricio Benevenuto, Niloy Ganguly, and Krishna Phani Gummadi. 2012. Understanding and Combating Link Farming in the Twitter Social Network. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12).* ACM, New York, NY, USA, 61–70. DOI:http://dx.doi.org/10.1145/2187836.2187846

[28] David Francis Gleich and Michael Saunders. 2009. Models and algorithms for pagerank sensitivity. __, *Sept* (2009).

[29] G. H. Golub and C. Greif. 2006. An Arnoldi-type algorithm for computing page rank. *BIT Numerical Mathematics* 46, 4 (2006), 759–771. DOI:http://dx.doi.org/10.1007/s10543-006-0091-y

[30] Andreas Griewank. 1991. Achieving Logarithmic Growth Of Temporal And Spatial Complexity In Reverse Automatic Differentiation. (1991).

[31] Andreas Griewank. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[32] Andreas Griewank and others. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989), 83–107.

[33] Andreas Griewank, David Juedes, and Jean Utke. 1996. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)* 22, 2 (1996), 131–167.

[34] Vince Grolmusz. 2015. A note on the pagerank of undirected graphs. *Inform. Process. Lett.* 115, 6 (2015), 633–634.

[35] GrouhLens. *MovieLens Datasets*. http://grouplens.org/datasets/movielens/.

[36] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 784–795.

[37] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for Generalized Map and Reduce Workflows. In *CIDR*.

[38] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 216–227. DOI:http://dx.doi.org/10.14778/2850583.2850595

[39] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 538–543.

[40] Meng Jiang, Peng Cui, Rui Liu, Qiang Yang, Fei Wang, Wenwu Zhu, and Shiqiang Yang. 2012. Social contextual recommendation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 45–54.

[41] Charles E Kahn, Linda M Roberts, Katherine A Shaffer, and Peter Haddawy. 1997. Construction of a Bayesian network for mammographic diagnosis of breast cancer. *Computers in biology and medicine* 27, 1 (1997), 19–29.

[42] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. 2004. Adaptive methods for the computation of PageRank. *Linear Algebra Appl.* (2004).

[43] Gershon Kedem. 1980. Automatic differentiation of computer programs. *ACM Transactions on Mathematical Software (TOMS)* 6, 2 (1980), 150–165.

[44] Gyanendra Kumar, Neelam Duhan, and AK Sharma. 2011. Page ranking based on number of visits of links of Web page. In *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*. IEEE, 11–14.

[45] S. L. Lauritzen and D. J. Spiegelhalter. 1990. Readings in Uncertain Reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems, 415–448. http://dl.acm.org/citation.cfm?id=84628.85343

[46] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[47] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 17.

[48] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-based Memory Management for Distributed Data Processing Systems. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 936–947. DOI:http://dx.doi.org/10.14778/2994509.2994513

[49] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 281–292. DOI:http://dx.doi.org/10.14778/2735508.2735517

[50] Rada Mihalcea and Paul Tarau. 2004. TextRank: Bringing order into texts. Association for Computational Linguistics.

[51] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1099–1110. DOI:http://dx.doi.org/10.1145/1376616.1376726

[52] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).

[53] Louis B Rall. 1981. Automatic differentiation: Techniques and applications. (1981).

[54] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 31–54.

[55] M Berkan Sesen, Ann E Nicholson, Rene Banares-Alcantara, Timor Kadir, and Michael Brady. 2013. Bayesian networks for clinical decision support in lung cancer care. *PloS one* 8, 12 (2013), e82349.

[56] Neelam Tyagi and Simple Sharma. 2012. Weighted Page Rank algorithm based on number of visits of Links of web page. *International Journal of Soft Computing and Engineering (IJSCE) ISSN* (2012), 2231–2307.

[57] Wenpu Xing and Ali Ghorbani. 2004. Weighted pagerank algorithm. In *Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on*. IEEE, 305–314.

[58] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. *Proc. VLDB Endow.* 9, 5 (Jan. 2016), 420–431. DOI:http://dx.doi.org/10.14778/2876473.2876477

[59] Jonathan S Yedidia, William T Freeman, Yair Weiss, and others. 2000. Generalized belief propagation. In *NIPS*, Vol. 13. 689–695.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[62] Hui Zhang, Ashish Goel, Ramesh Govindan, Kahn Mason, and Benjamin Van Roy. 2004. Making Eigenvector-Based Reputation Systems Robust to Collusion.. In *WAW (Lecture Notes in Computer Science)*, Stefano Leonardi (Ed.), Vol. 3243. Springer, 92–104.

[63] Xiao Jian Zhou and Ting Jiang. 2016. Enhancing Least Square Support Vector Regression with Gradient Information. *Neural Processing Letters* 43, 1 (2016), 65–83.