Towards Scalable Critical Alert Mining

Bo Zong[1] Yinghui Wu[1] Jie Song[2] Ambuj Singh[1]
Hasan Cam[3] Jiawei Han[4] Xifeng Yan[1]
[1]UC Santa Barbara [2] LogicMonitor [3]Army Research Lab [4] UIUC
{xyan, bzong, yinghui, ambuj}@cs.ucsb.edu jsong@logicmonitor.com
hasan.cam.civ@mail.mil hanj@cs.uiuc.edu

ABSTRACT

Performance monitor software for data centers typically generates a great number of alert sequences. These alert sequences indicate abnormal network events. Given a set of observed alert sequences, it is important to identify the most critical alerts that are potentially the causes of others. While the need for mining critical alerts over large scale alert sequences is evident, most alert analysis techniques stop at modeling and mining the causal relations among the alerts.

This paper studies the *critical alert mining* problem: Given a set of alert sequences, it is to find a set of critical alerts such that the number of alerts potentially triggered by them is maximized. The problem, although desirable, is intractable; therefore, we resort to approximation and heuristic algorithms. First, we develop a $1 - \frac{1}{e}$ approximation algorithm in quadratic time, and propose pruning techniques to improve its performance. Moreover, we show a faster approximation exists, when the alerts follow certain causal structure. Second, we propose two fast heuristic algorithms based on tree sampling techniques. On real-life data, these algorithms identify a critical alert from up to $270,000$ mined causal relations in 5 seconds; meanwhile, they preserve more than 80% of solution quality, and are up to $5,000$ times faster than their approximation counterparts.

1. INTRODUCTION

System monitoring and analysis in data centers and cyber security applications produces alert sequences to capture abnormal events. For example, performance metrics are posed on hosts in data centers to capture alerts such as high CPU usage, memory overflow, or service errors. Understanding the causal and dependency relations among these alerts is critical for data center management [10, 24], cyber security [15], and device network diagnosis [22], among others.

While there exist a variety of approaches for modeling and deriving causal relations [2, 30, 31], another important step is to efficiently suggest critical alerts from a huge amount of observed alerts. Intuitively, these critical alerts indicate the "root causes" that account for the observed alerts, such that if fixed, we may expect a great reduction of other alerts without blindly addressing them one by one. We consider several real-life applications below.

Data centers. System monitoring and analysis providers seek efficient and reliable techniques to understand a large number of system performance alerts in data centers. According to LogicMonitor[1], an SaaS network monitor company, a data center of 122 servers generates more than $20,000$ alerts per day. While it is daunting for domain experts to manually check these alerts one by one, it is desirable to automatically suggest a small set of alerts that are potentially causes for a large amount of alerts, for further verification. These critical alerts also help in determining key control points for data center infrastructures [24].

Intrusion detection [1, 14]. State-of-the-art intrusion detection systems produce a large amount of alerts from cyber network monitors and sensors, over tens of thousands of metrics. As suggested in [14], it is observed that a few critical alerts generally account for over 90% of the alerts that an intrusion detection system triggers. By handling only a small number of critical alerts, a huge amount of efforts and recourse can be reduced. On the other hand, critical alerts can be used to reduce a great number of "false alerts" and improve the alarm quality [1].

Network performance diagnosis [21]. Large-scale IP networks (*e.g.,* North America IPTV network) contain millions of devices, which generate a great number of performance alarms from customer call records and provider logs. Scalable mining of critical alerts for a given set of symptom events benefits fast network diagnosis [21].

These highlight the need for efficient algorithms to mine critical alerts, given the sheer size of observed ones. In this work, we investigate efficient critical alert mining techniques. We focus on general framework with desirable performance guarantees on alert quality and scalability.

(1) We formulate the *critical alert mining* problem: Given a set of alerts and a number k, it aims to find a set of k critical alerts, such that the number of alerts that are potentially caused by them is maximized. We introduce a generic framework for mining critical alerts. The framework learns and maintains an *alert graph*, a graph representation of causal relations among alerts. Upon request, top critical alerts are mined from alert graphs and returned to users.

[1]http://www.logicmonitor.com/

(2) Although desirable, we show that the critical alert mining problem is NP-complete. Nonetheless, we provide an algorithm with approximation ratio $1-\frac{1}{e}$, in time $O(k|V||E|)$, where $|V|$ and $|E|$ are the number of alerts and their causal relations, respectively. To further improve the efficiency of the algorithm, we propose a bound and pruning algorithm that effectively reduces the size of alerts to be verified as critical ones. In addition, we identify a special case: when alert graphs are trees, it is in $O(k|V|)$ time to find k critical alerts, with the same approximation ratio.

(3) We further propose two fast time heuristics for large-scale critical alert mining. These algorithms induce trees that preserve the most probable causal relations from large alert graphs, and estimate top critical alerts and their impact by only accessing the trees. The first one induces a single tree, while the second algorithm balance alert quality and mining efficiency with multiple sampled trees.

(4) We experimentally verify our critical alert mining framework. Over real-life data center datasets, our algorithms effectively identify critical alerts that trigger a large number of other alerts, as verified by domain experts. We found that our approximation algorithms mine a top critical alert from up to $270,000$ causal relations (one day's alert sequences) in 5 seconds. On the other hand, while our heuristics preserve more than 80% of solution quality, they are up to $5,000$ times faster. The heuristics also scale well over large synthetic alert graphs, with up to in total 1 million alerts and 10 million relations.

Our work aims to provide an efficient platform for large-scale critical alert mining. We distinguish our methods from causality modeling and fast causality mining approaches. In addition, we do not assume the luxury of accessing rich semantics from the alerts that helps in improving mining efficiency, although our methods immediately benefit from the semantics in specific applications [14, 21, 24], as well as domain experts. Taken together with domain knowledge and causality mining tools, these algorithms are one step towards large-scale critical alert analysis for data centers, intrusion detection systems, and network diagnose systems.

2. PROBLEM DEFINITION

We start with the notions of alert sequences and alert graphs. Then we introduce the critical alert mining problem.

Performance metrics. A performance metric measures an aspect of system performance. For data centers, common types of performance metrics include CPU and memory usage for virtual machines, error rate of disk writes for a service, or communication time between two hosts. The same type of metrics over different hosts, virtual machines, or services are considered as distinct performance metrics.

In practice, system service providers like LogicMonitor may cope with as much as 2 million performance metrics from a data center with $5,000$ hosts. These metrics could correlate with and cause each other due to functional or resource dependencies.

Alert and alert sequences. For a set of performance metrics $\mathbb{P}$, alerts are determined by aggregating the metric values of interests. For example, in data centers, an alert is raised when the value of a performance metric (*e.g.,* CPU usage) goes beyond a pre-defined threshold (*e.g.,* $> 75\%$). In this work, we define an alert as a triple $u = (p_u, t_u, w_u)$, where $p_u \in \mathbb{P}$ is a performance metric u corresponds to, t_u denotes the timestamp when the alert u happened, and w_u is the weight of u, representing the benefit if u is fixed.

We use a sequence of alerts to characterize abnormal events for a specific performance metric. Indeed, in practice the performance metrics are typically periodically monitored. We denote as $\vec{s}_p$ an alert series (an ordered sequence of alerts following their timestamps), for a specific performance metric $p \in \mathbb{P}$. Each entry of $\vec{s}_p$ is either 0 (normal) or 1 (alert).

To characterize causal relations between two alerts, we next introduce a notion of *dependency rule*. We also introduce *alert graph* as an intuitive graph representation for multiple dependency rules.

Dependency rule. Let p and q be two distinct performance metrics. A dependency rule $p \xrightarrow{l_{pq}} q$ denotes an alert issued on q at some time t is caused by an alert issued on p at $t' \in [t-l_{pq}, t-1]$, where l_{pq} is a *lag* from p to q (*e.g.,* 5 minutes). Note that we do not specify the time t, as a dependency rule describes a statistical rule for all the observed alerts. Intuitively, a dependency rule indicates that q occurs if and only if p occurs as the cause of q. If certain trouble shooting action is taken to fix p, q is addressed accordingly [1, 21].

Dependency rules can be automatically learned from alert series [2, 30]. They can also be suggested by experts and exiting knowledge bases [8]. To smoothen the noise or error brought by rule generation process, we associate an uncertainty to each dependency rule. In particular, we denote the uncertainty by $\Pr(p \xrightarrow{l_{pq}} q)$, which is the probability that the corresponding dependency rule holds.

Alert graph. An alert graph over a set of alerts V is a directed acyclic graph $G = (V, E, f_e)$:

- V is the set of vertices in G, where each vertex $v \in V$ is an alert from V.
- E is a set of edges in G. Let $u = (p_u, t_u, w_u)$ and $v = (p_v, t_v, w_v)$ be two alerts in V. There is an edge $(u, v) \in E$ if and only if there exists a dependency rule $p_u \xrightarrow{l_{p_u p_v}} p_v$, where $t_u < t_v$, and $t_v - t_u \leq l_{pq}$.
- f_e is a function that assigns for each edge (u, v) the probability that u *causes* v, *i.e.,* $\Pr(p_u \xrightarrow{l_{p_u p_v}} p_v)$.

We shall use the following notations. Abusing the notions from tree topology, we say u (resp. v) is a parent (resp. child) of v (resp. u) if $(u, v) \in E$, and the edge (u, v) is an incoming edge of v. The *topological order* r of an alert u in G is defined as follows. (a) $r(u) = 0$ if u has no parent, and (b) $r(u) = 1 + \max r(v)$, for all its parents v.

Following the convention of causal relation and cascading models [28], we assume that an alert is caused by a single alert issued earlier, if any. Intuitively, a path from an alert u to another alert v in the alert graph indicates a potential "causal chain" from u to v.

Critical alerts. We next introduce a metric to characterize critical alerts, in terms of how many alerts are potentially caused by them via a cascading effect (and hence are addressed if the critical ones are fixed). Given $G = (V, E, f_e)$, a set of fixed alerts $S \subseteq V$, and an alert $u \in V$, we use a

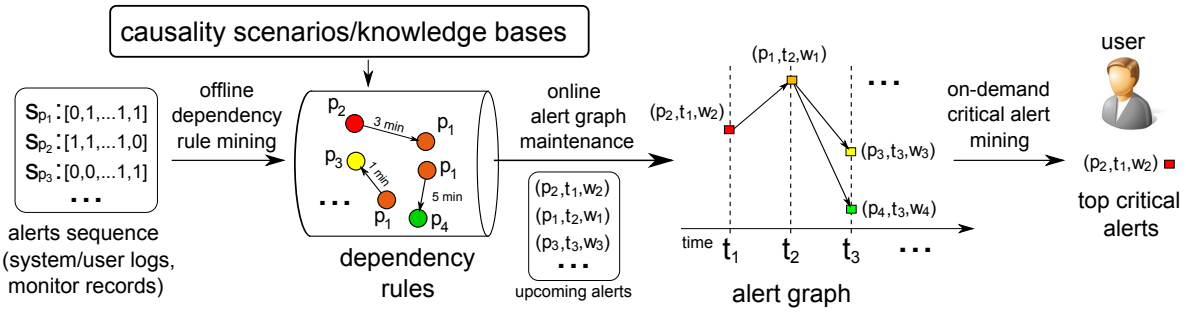


Figure 1: Critical alert mining: pipeline

notion of *alert-fixed probability* P_f to characterize the probability that u is fixed if S is fixed. More specifically,

- $P_f(S,u) = 1$ if $u \in S$,
- otherwise,

$$P_f(S,u) = 1 - \prod_{(u',u)\in E} \Big(1 - P_f(S,u') f_e(u',u)\Big),$$

Based on the alert-fixed probability, we next define a set function, denoted as Gain, to characterize critical alerts. Given an alert graph $G = (V, E, f_e)$ and $S \subseteq V$, the gain of S is a set function

$$\mathsf{Gain}(S) = \sum_{u\in V} w_u \cdot P_f(S,u).$$

Intuitively, $\mathsf{Gain}(S)$ computes the total expected benefits induced via fixing a set of alerts S and subsequently addressing the alerts caused by S. The larger $\mathsf{Gain}(S)$ is, the more "critical" S is.

We next introduce the critical alert mining problem.

DEFINITION 1. *Given an alert graph G and an integer k, the critical alert mining problem (referred to as* CAM*) is to find a set of k critical alerts $S \subset V$ such that* $\mathsf{Gain}(S)$ *is maximized.*

It is desirable to find k best alerts with the maximized gain. The problem is, nevertheless, nontrivial.

THEOREM 1. *For a given alert graph G and an integer k, the problem* CAM *is NP-complete.*

PROOF. We prove the NP-completeness of the decision version of CAM as follows. (1) CAM is in NP. Indeed, given an alert graph $G = (V, E)$ and a set of vertices $S \subseteq V$, one can evaluate Gain by computing $P_f(S, v)$ of each alert v in polynomial time. (2) To show that CAM is NP-hard, we construct a reduction from the maximum coverage problem, which is known to be NP-hard [34]. An instance of a maximum coverage problem consists a set of sets $\mathcal{S}$ and an integer k. It is to select at most k of these sets such that the number of elements that are covered is no less than a bound B. A maximum coverage instance can be constructed as a bipartite alert graph, with each "upside" node as a set in $\mathcal{S}$, each "downside" node a distinct element in these sets, and there is an edge from upside node to downside node if the corresponding element is in the set denoted by the upside node. In addition, the weights on edges are uniformly 1. Given the bound B, one may verify that there is a solution for the maximum converage problem if and only if there is a set S of k critical alerts with $\mathsf{Gain}(S) \geq B$. Therefore, CAM is at least as hard as maximum coverage problem, and is NP-hard. Hence, CAM is NP-complete. □

3. MINING FRAMEWORK

In this section, we present a framework for critical alert mining. It consists of three components as illustrated in Figure 1: (1) offline dependency rule mining; (2) online alert graph maintenance; and (3) on-demand critical alert mining.

Offline dependency rule mining. Given a set of observed alert sequences, the system mines the alerts of interests and their causal relations offline, and represent them as a set of dependency rules. As there are a variety of methods to model a causal relation, in this work we adopt Granger causality [2, 30], which can naturally be represented by dependency rules. An alert sequence X is said to Granger-cause another sequence Y if it can be shown, via certain statistic tests on lagged values of X and Y, that the values of X provide statistically significant information to predicate the future values of Y. More specifically,

(1) We collect alert sequences for all performance metrics of interest as training data, following two criteria as follows: (a) the alerts in training data should be the latest ones such that the latest dependency patterns among performance metrics can be captured; and (b) the alert information should be rich enough such that learned dependency rules would be more robust. In our work, we treat the latest one week alert data as the training data.

(2) We apply existing Granger causality analysis tools [30] to mine the dependency rules, and apply conditional probabilities to estimate the uncertainty of the rules [18].

The learned dependency rules are stored in knowledge bases to support online alert graph maintenance. Moreover, existing knowledge bases such as event causality scenarios [8] can also be "plugged in" in our critical causal mining framework. The dependency rules are then shipped to the next stage in the system to maintain alert graphs.

Online alert graph maintenance. Using dependency rules, our system constructs and maintains an alert graph G online from a range of newly issued alerts. Upon an alert u from performance metric q is detected at time t, it first marks u as a new alert in G. It then checks (1) if there exists dependency rules in the form of $p \xrightarrow{l_{pq}} q$, and (2) whether there are alerts detected on performance metric p during the time period $[t - l_{pq}, t)$. If there exists such an alert v on p, an directed edge from v to u is inserted, and

the rule uncertainty $\Pr(p \xrightarrow{l_{pq}} q)$ is associated to the edge (v, u). Following the above steps, it maintains G online for newly detected alerts.

On-demand critical alert mining. The major task (and the focus of this work) in the pipeline is to identify k critical alerts from alert graphs. In practice, a user may specify a time window of interests, which induces an alert graph from the maintained alert graph. It contains all the alerts detected during the time window. However, the induced alert graphs can still be huge.

In this paper, we propose three algorithms to address the scalability issue: (1) a quadratic time approximation algorithm with performance guarantees on the quality of critical alerts, (2) a faster approximation algorithm, which guarantees the alert quality for tree-structured alert graphs; and (3) sampling-based heuristics which can be *tuned* to balance the alert quality and response time. The critical alerts are then returned to users for further analysis and verification.

4. BOUND AND PRUNING ALGORITHM

Theorem 1 tells us that it is unlikely to find a polynomial time algorithm to find the best k alerts with the maximum gain. All is not lost: we can find polynomial time algorithms that approximately identifies the most critical alerts. The main result in this section is as follows.

THEOREM 2. *Given an alert graph $G = (V, E, f_e)$ and an integer k, (1) there exists an algorithm in $O(k|V||E|)$ time with approximation ratio $1 - \frac{1}{e}$, and (2) there exists a $1 - \frac{1}{e}$ approximation algorithm in $O(k|V|)$ time, when G is a tree.*

Denote the optimal k alerts as S^*, we present an efficient algorithm to identify k alerts S where $\mathsf{Gain}(S) \geq (1 - \frac{1}{e})\mathsf{Gain}(S^*)$, in quadratic time.

We start with a greedy algorithm, denoted as Naive.

Naïve greedy algorithm. Given an alert graph $G = (V, E, f_e)$ and an integer k, Naive finds k critical alerts in k iterations as follows. (1) It initializes a set S_0 to store the selected alerts. (2) At the ith iteration, Naive checks each alert in V, and greedily picks the alert s_i that maximizes the *incremental gain* $\mathsf{Gain}(S_{i-1} \cup \{s_i\})$, where S_{i-1} is the set of critical alerts found at iteration $i - 1$. (3) It repeats the above step until k alerts are identified.

One may verify that Naive is a $1 - \frac{1}{e}$ approximation algorithm. To see this, observe that the set function $\mathsf{Gain}(\cdot)$ is a *monotonically submodular function*. A function $f(S)$ over a set S is called *submodular* if for any subset $S_1 \subseteq S_2 \subset S$ and $x \in S \setminus S_2$, $f(S_1 \cup \{x\})$ - $f(S_1) \geq f(S_2 \cup \{x\})$ - $f(S_2)$. It is known that for maximizing a submodular function, a greedy strategy achieves $1 - \frac{1}{e}$ approximation ratio [26]. Hence it suffices to show that the function Gain is a monotonically submodular function. Indeed, (1) one may verify that Gain is monotonic: for any $S_1 \subseteq S_2 \subseteq V$, $\mathsf{Gain}(S_1) \leq \mathsf{Gain}(S_2)$; (2) the diminishing return of Gain can be shown by mathematical reduction. We provide the detailed proof in the appendix.

For complexity, Naive requires k iterations, and in each iteration, it scans all the vertices u and computes $P_f(S_{k-1}, u)$, which takes in total $O(k|V||E|)$ time.

Naive provides a polynomial time algorithm to approximate CAM within $1 - \frac{1}{e}$. Nevertheless, the scalability issue of Naive makes it difficult to use in practice for large alert graphs. For instance, when an alert graph of around $20K$ vertices and $200K$ edges, Naive mines 6 critical alerts in more than 800 seconds. We next present a faster approximation algorithm with the same approximation ratio. By using pruning and verification, the algorithm is 30 times faster than Naive, as verified in our experimental study.

4.1 Pruning and verification

To select a most promising alert at each iteration, Naive evaluates the incremental gain for each alert in $V \setminus S$, and then selects the one of the highest incremental gain, which runs in $O(|V||E|)$. Instead of blindly processing every alert, we may efficiently filters "unpromising" alerts using estimated gain, and then evaluates the exact gain for the remaining vertices. In particular, at each iteration i, for two alerts v' and $v \in V \setminus S_{i-1}$, we compute upper bounds U'_v, U_v and lower bounds L'_v, L_v for $\mathsf{Gain}(S_{i-1} \cup \{v\})$ and $\mathsf{Gain}(S_{i-1} \cup \{v'\})$, respectively. If v' is already not a critical alert, all the alerts v with $L'_v > U_v$ can be safely skipped without losing the alert quality.

We next derive an upper and lower bound for $\mathsf{Gain}(\cdot)$, and present algorithms to compute them efficiently. Instead of visiting each alert and causal relation in G, these algorithms compute the bounds by visiting only local information of each alert in G. This enables a fast estimation for $\mathsf{Gain}(\cdot)$.

4.2 Upper bound

We introduce a notion of *sum gain* (denoted as SGain) to characterize the upper bound for $\mathsf{Gain}(\cdot)$. Given an alert graph $G = (V, E, f_e)$, an alert $v \in V$, and a set of selected critical alerts $S \subseteq V$, an upper bound is computed as $\mathsf{SGain}(S \cup \{v\}) = \sum_{u \in V} w_u \cdot \hat{P}_f(S \cup \{v\}, u)$, where

- $\hat{P}_f(S \cup \{v\}, u) = 1$, if $u \in S$;
- $\hat{P}_f(S \cup \{v\}, u) = \sum_{(u', u) \in E} \hat{P}_f(S \cup \{v\}, u') f_e(u', u)$, if $u \notin S$.

The sum gain SGain (as illustrated in Fig. 2) is an upper bound for $\mathsf{Gain}(\cdot)$. Better still, it can be efficiently computed.

PROPOSITION 1. *Given an alert graph $G = (V, E, f_e)$, a set of critical alert $S \subseteq V$, and an alert $u \in V \setminus S$, (1) $\mathsf{Gain}(S \cup \{u\}) \leq \mathsf{SGain}(S \cup \{u\})$; and (2) SGain can be computed for* all *alerts in V in $O(|E|)$ time.*

We first prove Proposition 1 (1). We remark that SGain is built upon the following generalization of Bernoulli's inequality [23]. Given $x_i \leq 1$, we have

$$1 - \prod_{i=1}^{n} (1 - x_i) \leq \sum_{i=1}^{n} x_i.$$

We next conduct a mathematical induction over the topological order (Section 2) of the alerts in G as follows.

- Consider the alerts $u_1 \in V$ with topological order 0: (1) if $u_1 \in S$, $\hat{P}_f(S, u_1) = 1$ and $P_f(S, u_1) = 1$; (2) otherwise, $\hat{P}_f(S, u_1) = 0$ and $P_f(S, u_1) = 0$, since u_1 has no parents. In both cases, $P_f(S, u_1) \leq \hat{P}_f(S, u_1)$.
- Assume that alert $u_i \in V$ with topological order i sat-

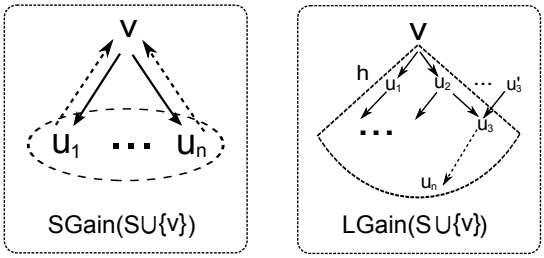


Figure 2: Algorithm BnP: Upper and lower bound

isfies $P_f(S, u_i) \leq \hat{P}_f(S, u_i)$. For an alert $u_{i+1} \in V$,

$$\begin{aligned} P_f(S, u_{i+1}) &= 1 - \prod_{(u', u_{i+1}) \in E} \big(1 - P_f(S, u') f_e(u', u_{i+1})\big) \\ &\leq \sum_{(u', u_{i+1}) \in E} P_f(S, u') f_e(u', u_{i+1}) \\ &\leq \sum_{(u', u_{i+1}) \in E} \hat{P}_f(S, u') f_e(u', u_{i+1}) \\ &= \hat{P}_f(S, u_{i+1}) \end{aligned}$$

Therefore, for any $u \in V$, $P_f(S, u) \leq \hat{P}_f(S, u)$. By definition, $\mathsf{Gain}(S \cup \{u\}) \leq \mathsf{SGain}(S \cup \{u\})$. Hence, SGain is indeed an upper bound for $\mathsf{Gain}(\cdot)$.

Upper bound computation. As a constructive proof for Proposition 1 (2), we present a procedure (denoted as computeUpperBound) for SGain to compute the upper bounds for all vertices in $O(|E|)$ time.

The algorithm (not shown) follows a "bottom up" computation, starting from the alerts with the highest topological order in G. (1) It first computes the topological order for all the alerts in G. (2) Starting from the alert with the highest topological order, it computes SGain for each alert $u \in V \setminus S$ as follows: (a) $\mathsf{SGain}(S \cup \{u\}) = \mathsf{SGain}(S \cup \{u\}) + w_u$, and (b) for each $u' \in N_i(u)$, it updates $\mathsf{SGain}(S \cup \{u'\})$ by $\mathsf{SGain}(S \cup \{u'\}) + f_e(u', u)\mathsf{SGain}(S \cup \{u\})$. (3) It repeats step (2) until all the alerts are processed.

It takes $O(|E|)$ time for computeUpperBound to obtain the topological order by depth-first search in step (1). Each edge in G is visited exactly once in step (2) and (3). Therefore, the algorithm runs in $O(|E|)$ time.

The above analysis completes the proof of Proposition 1.

4.3 Lower bound

To compute the lower bound of $\mathsf{Gain}(\cdot)$, we introduce a notion *local gain* (denoted as LGain). Given an alert graph $G = (V, E)$, an alert $v \in V$, a set of selected alerts $S \subseteq V$, and an integer h, LGain of $S \cup \{v\}$ is defined as follows.

$$\mathsf{LGain}(S \cup \{v\}) = \sum_{u \in V_v^h} w_u \cdot P_f(S \cup \{v\}, u),$$

where h is a tunable integer, and $V_v^h \subseteq V$ is a set of vertices that can be reached from v in no more than h hops. Intuitively, LGain estimates a lower bound of $\mathsf{Gain}(S)$ with the impact of an alert to its local "nearby" alerts in G (as illustrated in Fig.2). One may verify the following.

PROPOSITION 2. *Given $G = (V, E)$, $S \subseteq V$, for any alert $u \in V \setminus S$, (1)* $\mathsf{Gain}(S \cup \{v\}) \geq \mathsf{LGain}(S \cup \{v\})$*, and (2)* LGain *can be computed in $O(\sum_{v \in V} |E_v^h|)$ time, where E_v^h is the set of incoming edges in G of the alerts in V_v^h.*

We present a procedure computeLowerBound to compute LGain. For each alert $v \in V \setminus S$ (*e.g.,* u_3 in Fig 2), the algorithm visits the alerts in V_v^h and their incoming edges (*e.g.,* (u'_3, u_3)) once, and computes LGain following the definition, in $O(\sum_{v \in V} |E_v^h|)$ time.

4.4 Algorithm BnP

Based on the upper and lower bounds, we propose an efficient approximation algorithm, denoted as BnP, for the problem CAM. BnP enables faster critical alert mining while achieves the approximation ratio $1 - \frac{1}{e}$. The algorithm BnP follows Naive's greedy strategy: given an integer k, it conducts k iterations of search, each determines a top critical alert. The difference is that in each iteration, it invokes a procedure Prune to identify a set C of candidate alerts for consideration.

The procedure Prune (as illustrated in Fig. 3) invokes computeUpperBounds and computeLowerBounds to dynamically updates the lower and upper bounds for each alert by accessing their local information (lines 1-2), and filters the alerts that are not possible to be critical ones:

1. it scans the lower bounds LGain of each alert, and find the maximum one as *bar* (line 3);
2. it scans the upper bounds SGain of each alert, and prunes those with $\mathsf{SGain}(u) < bar$, adding the rest to a candidate alert set C.

Input: An alert graph $G = (V, E, f_e)$;
a set of critical alert S.
Output: a set of candidate alerts C.

1. computeUpperBound (G, S);
2. computeLowerBound (G, S);
3. set bar as the largest LGain over alerts in $V \setminus S$;
4. $C \leftarrow \emptyset$;
5. **for each** $u \in V \setminus S$
6. **if** $\mathsf{SGain}(u) \geq$ bar
7. $C \leftarrow C \cup \{u\}$;
8. **return** C;

Figure 3: The pruning procedure Prune

Correctness and Complexity. The algorithm BnP achieves approximation ratio $1 - \frac{1}{e}$, as it follows the same greedy strategy as Naive. Note that the pruning procedure Prune does not affect the approximation ratio.

For complexity, let C_m be the maximum set of candidate sets in all the iterations after pruning. For the alerts in C_m, it takes BnP $O(|C_m||E|)$ time to find a best alert. The total time for pruning is $O(k(\sum_{u \in V} |E_u^h| + |E|))$. Hence, it takes BnP in total $O(k(\sum_{u \in V} |E_u^h| + |C_m||E|))$ time. Moreover, $|E_u^h|$ is typically small, and is *tunable* by varying h, as indicated by Proposition 2. For example, when $h = 1$, LGain can be computed in $O(d_m|E|)$ for all the alerts, where d_m is the largest in-degree in G. As h gets larger, the computation complexity gets higher, leading to tighter lower bound LGain. In our experimental study, by setting $h = 3$, 95% of the alerts are pruned, which makes BnP 30 times faster than Naive without losing alert quality.

Mining Alert Trees. When G is a directed tree, the algorithm BnP identifies k critical alert in $O(k|V|)$ as follows. (1) Starting from the alerts $u \in V$ of the highest topological

order, it computes $\mathsf{Gain}(u) = \mathsf{Gain}(u) + w_u$, and makes an update by $\mathsf{Gain}(u') = \mathsf{Gain}(u') + f_e(u', u)\mathsf{Gain}(u)$, if u' is the parent of u. (2) It repeats (1) on the alerts following the decreasing topological order, until all the alerts are processed. One iteration over (1) and (2) identifies a critical alert. (3) BnP repeats (1) and (2) to find k critical alerts.

Following the correctness analysis, BnP preserves the approximation ratio $1-\frac{1}{e}$ over trees. Moreover, each edge in G is visited once in a single iteration. Hence, it takes $O(k|V|)$ time of BnP over G as trees. Theorem 2 (2) hence follows.

5. TREE APPROXIMATION

The performance of algorithm BnP requires all the candidates and their causal relations to be processed, which may not be efficient for a large amount of alert sequences. In extreme cases where few alert is pruned, BnP degrades to its naïve greedy counterpart.

As indicated by Theorem 2(2), fast approximation exists for alert graphs as trees. Following this intuition, we may make large alert graphs "small", by sparsifying them into directed trees, which "preserves" most of alert dependency information in an alert graph. This enables both fast algorithms and low quality loss. In this section, we introduce fast heuristic algorithms for CAM.

5.1 Single-tree approximation

We start by introducing a heuristic algorithm ST. The basic idea is to induce a *maximum directed tree* (forest) T from a given alert graph G, such that for any set of alerts S in G, $\mathsf{Gain}(S)$ in T is "close" as much as possible to $\mathsf{Gain}(S)$ in G, and a fast approximation can be performed over T without much quality loss.

Maximum directed tree. Given an alert graph $G = (V, E, f_e)$, a maximum directed tree of G is a spanning tree $T = (V, E')$, where $E' \subset E$, such that (1) for any $u \in V$, u has at most one incoming edge, and (2) $\sum_{\langle u,v\rangle \in E'} f_e(u, v)$ is maximized. Intuitively, T depicts a "skeleton" of an alert graph G, where causal relations always follow the most likely dependency rules.

Algorithm ST. Given an alert graph $G = (V, E, f_e)$, the single-tree approximation ST mines k critical alerts as follows. (1) ST first finds the maximum directed tree T. To construct T, an algorithm simply selects, for each alert u in G, the incoming edge (u', u) with the maximum $f_e(u', u)$ among all its incoming edges. (2) ST searches the k critical alerts following the algorithm BnP over T.

One may verify that it is in $O(|E|)$ time to construct T. From Theorem 2(2), it is in $O(k|V|)$ time to find k critical alerts in T (as either a tree or a forest). Hence, the algorithm ST takes in total $O(|E| + k|V|)$ time. Note that the induced T can be a set of disjoint trees, where the above complexity still holds.

5.2 Multi-tree sampling

Single-tree approximation provides fast mining method for large scale alerts. On the other hand, using induced trees to approximate causal structures may lead to biased results. For example, more dependency information could be lost for alerts with more incoming edges. To rectify this, we propose a heuristic, denoted as MTS, based on multi-tree sampling.

Algorithm MTS. The algorithm MTS is as illustrated in Fig. 4. Given an alert graph $G = (V, E, f_e)$, integer k and a sample number N, MTS starts by initializing a set S_0 as $\emptyset$, the alert-fixed probability for each node as 0 (line 1), and identify the topological orders of the alerts in G.

Input: Alert graph $G = (V, E, f_e)$,
integer k, the number of sampled trees N.
Output: A set S of k critical alerts.

1. $S \leftarrow \emptyset$; initializes $P_f(\cdot)$; $i \leftarrow 0$;
2. **while** $i \leq k$ **Do**
3. **for each** alert u in G **Do**
4. update $P_f^{(i-1)}(S_{i-1}, u)$;
5. $l \leftarrow 0$;
6. **while** $l \leq N$ **Do**
7. $T_{i,l} \leftarrow \mathsf{sampleTree}(G)$;
8. **for each** alert u in G **Do**
9. $\mathsf{Gain}(S_{i-1} \cup \{u\}) \leftarrow \mathsf{Gain}(S_{i-1}) + \frac{\sum_{l=1}^{N} D(u,l)}{N}$;
10. select u with the maximum $\mathsf{Gain}(S_{i-1} \cup \{u\})$;
11. $S_i \leftarrow S_{i-1} \cup \{u\}$;
12. **return** S_k;

Figure 4: Algorithm MTS

Algorithm MTS then finds a set S critical alerts in k iterations as follows. Denote the selected critical set at iteration $i-1$ as S_{i-1}. At each iteration i, (1) MTS updates the alert-fixed probability $P_f^{(i-1)}(S_{i-1}, u)$ for each alert $u \in V$ in G, fixing S_{i-1} as the critical alerts (lines 3-4). (2) It then invokes procedure sampleTree to sample N trees from G (lines 6-7), according to the updated alert-fixed probability in (1). (3) For each alert u, MTS computes the weighted sum of u's descendants $D(u, l)$ in each sampled tree $T_{u,l}$, and takes the average $D(u, l)$ over all sampled tress as an estimation of $\mathsf{Gain}(S_{i-1} \cup \{u\})$ (lines 8-9). It selects the alert u that introduces the maximum improvement, and update S_{i-1} as S_i by adding u (lines 10-11), which is used to update $P_f(\cdot)$ in G in the next iteration.

Procedure sampleTree. Given an alert graph G and an integer N, the procedure sampleTree (line 7) samples N trees (forest) from G at iteration i. More specifically, it generates a single tree (or forest) $T_{i,l}$ as follows. (1) It first samples a set of alerts $V_{i,l}$ as the nodes for tree $T_{i,l}$ following Bernoulli distributions. For each alert $u \in V$ and the updated $P_f^{(i-1)}(u)$, MTS selects u with probability $1-P_f^{(i-1)}(u)$, and inserts it to $V_{i,l}$. (2) MTS then samples an edge for each alert $u \in V_{i,l}$. It randomly order u's parents. Starting from the first parent, u tries to build an edge (u', u) to its parent with probability $f_e(u', u)$, where u' ranges over all the parents of u, until an edge is selected (and attached to u), or all the parents are visited. (3) MTS repeats (2) until all the alerts $u \in V_{i,l}$ are visited.

It takes in total $O(k*N|E|)$ time for MTS to find k critical alerts. (a) MTS takes in total $O(k|E|)$ time to update P_f in G; (b) the total sampling time is in $O(k * N|E|)$; and (c) it takes in total $O(k * N|V|)$ time to select the critical alerts.

In contrast to its single-tree counterpart, MTS leverages sampling to reduce the bias: alerts with more parents and larger probability are more likely to have a parent in a sampled tree. In addition, it synthesizes the gain estimation from multiple trees, such that the noise from a single tree is smoothed. Indeed, we found that using only 300 samples, MTS finds top 6 critical alerts with $\mathsf{Gain}(\cdot)$ 90% as good as Naive, and is 80 times faster. It reduces 10% more alert quality loss compared with ST (see Section 6).

6. EXPERIMENT

We applied both real-life and synthetic data to evaluate our algorithms. We first provide a case study (Section 6.2). Using real-life data, we next investigate (1) the efficiency and effectiveness of our algorithms (Section 6.3), (2) the impact of the number of explored hops to the performance of BnP (Section 6.4), and (3) how the number of samples affects MTS (Section 6.5). In addition, we evaluate the scalability of our algorithms, over large synthetic data (Section 6.6).

6.1 Setup

Real-life data. We use real-life data center performance data (referred to as LM), from LogicMonitor, an SaaS network monitoring company. The data spans 53 days from Nov. 23, 2013 to Jan. 14, 2014. It contains the sequences for 50,772 performance metrics from 9,956 services residing in 122 servers. Each metric is reported every 2 minutes. The alerts are identified by specified rules provided by LogicMonitor, where we assign a weight 1.0 to all the metrics.

Dependency rules and alert graphs. Dependency rules were mined from data collected in 7 consecutive days, and are used to construct alert graphs using the data from the following days. We used the tool developed by [30] to mine the Granger causality among performance metrics as dependency rules (with the p-value set to be 0.01 [30]). We then applied conditional probability to estimate the uncertainty of the rules [18]. From the dataset LM, we mined 46 sets of dependency rules, where each set contains on average $2,082$ rules. Each set of rules were mined in less than 60 minutes.

By applying the sets of dependency rules on the alert detected in the next single day, we obtained 46 alert graphs, following the online alert graph construction (Section 3). The number of alerts (resp. edges) ranges from $20,248$ to $25,057$ (resp. $162,000$ to $270,370$) for a single graph.

Synthetic alert graphs. For scalability tests over large alert graphs, we applied the graph model proposed in [16] to generate large synthetic alert graphs (referred to as SYN). In particular, the node degree and edge weights follow the empirical distributions [33] learned from alert graphs over the real-life data LM. We ranged the number of alerts from 100K to 1M, and the average degree of SYN graphs is 9.

Evaluation. To measure the quality of the critical alerts identified by an algorithm A, we investigate a metric *loss ratio* of A defined as

$$\textbf{loss ratio}(A) = 1 - \frac{Gain(S_A)}{Gain(S_{\mathsf{Naive}})},$$

where S_{Naive} (resp. S_A) is the set of critical vertices returned by the algorithm Naive (resp. algorithm A). As Naive guarantees the alert quality within a bound, loss ratio suggests how "close" the quality of the alerts from heuristic algorithms and the optimal ones is. The less, the better.

Implementation. In addition to the proposed algorithms BnP, ST, and MTS, we implemented the following baseline algorithms: (1) Naive, the greedy algorithms without pruning strategy; (2) $\mathsf{BnP}_{\mathsf{UB}}$, a simplified version of BnP, which only uses upper bound to filter unpromising alerts: it skips those alerts with upper bound smaller than an alert with computed $\mathsf{Gain}(\cdot)$ in each iteration (Section 4). (3) MaxDeg, a simple strategy that returns the top k alerts with the largest weighted sum of outgoing edges.

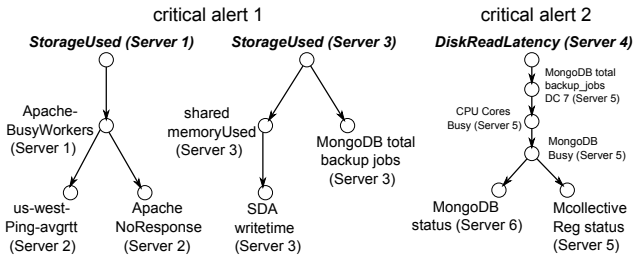


Figure 5: Critical alerts over LM

All the algorithms were implemented in C++, and all experiments were executed on a machine powered by an Intel Core i7-2620M 2.7GHz CPU and 8GB of RAM, using Ubuntu 12.10 with GCC 4.7.2. Each experiment was run 10 times, and their average results are presented.

6.2 Case study

Using real-world data LM, our algorithms suggest reasonable critical alerts that are indeed the source of a range of large amount of alerts, as verified by the domain experts from LogicMonitor. We illustrate three "causality patterns" induced by top two critical alerts and their descendants following the weighted dependency rules in Fig. 5. (1) Our algorithms suggest that StorageUsed, a critical alert that indicates insufficient memory, leads to poor performance of Web servers (Apache), which typically triggers delayed Ping round-trip time (Ping-avgrtt) from other servers. In another set of hosts, it leads to insufficient shared memory over a range of servers, which typically triggers slower Shared Data Access write time (SDA writetime) on their own. (2) A second critical alert DiskReadLatency suggests I/O bottleneck for a range of abnormal status of database applications. The disk access speed alert often triggers the unsolved back up requests from another server, which leads to poor performance of CPU and database servers, and further affects a range of database related requests from more outside servers. These causal patterns are consistent with the workflow of data centers at LogicMonitor.

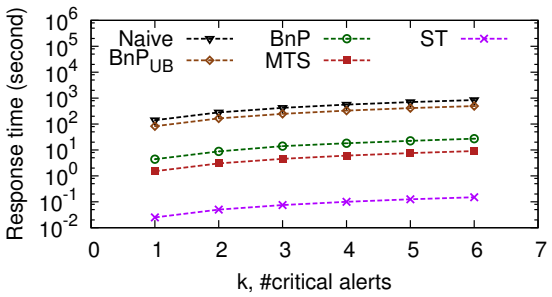
Our algorithms do not assume prior domain knowledge. On the other hand, external knowledge and rules enable our algorithms to further improve the quality of the critical alerts and causal patterns.

6.3 Overall performance evaluation

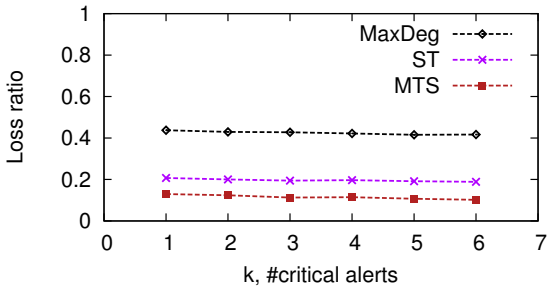
We first investigate the efficiency and effectiveness of the proposed algorithms, using alert graphs from LM. In the following tests, we fixed the number of explored hops in BnP as 3, and the number of sampled trees in MTS as 300.

As illustrated in Figure 6(a), the proposed algorithms BnP, MTS, and ST consistently outperform the baseline algorithms Naive and $\mathsf{BnP}_{\mathsf{UB}}$ in efficiency, while varying k, the number of required critical alerts. They introduce different levels of efficiency improvement. Compared with Naive and $\mathsf{BnP}_{\mathsf{UB}}$, BnP is 30 times and 17 times faster, respectively, without quality loss on solutions. With some quality loss, ST is 5000 times and 3000 times faster than Naive and $\mathsf{BnP}_{\mathsf{UB}}$, respectively, and MTS results in 80 times and 50 times speedup. In addition, all the algorithms take more time when k varies from 1 to 6, as expected.

Figure 6(b) shows the loss ratio of ST and MTS, where k varies from 1 to 6. Compared with MaxDeg, MTS and ST

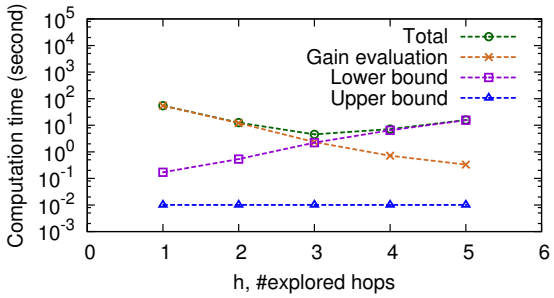


(a) Mining efficiency on different algorithms

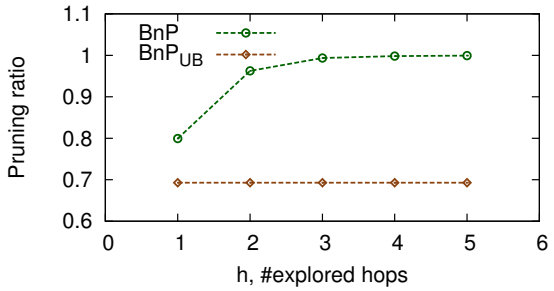


(b) Loss ratio comparison

Figure 6: Mining performance on LM alert graphs



(a) Computation time on different components



(b) Pruning ratio comparison

Figure 7: BnP performance on LM alert graphs

obtain significant improvement on loss ratio. As k increases, the loss ratio of MaxDeg is consistently more than 0.4; meanwhile, the loss ratio of MTS and ST is around 0.1 and 0.2, respectively. Compared with MTS, ST receives higher efficiency at the cost of solution quality loss. When the number of required critical alerts varies from 1 to 6, MTS and ST share the same trend: the loss ratio decreases. Compared with BnP that returns critical alerts without quality loss, ST and MTS are 180 and 3 times faster, respectively, at the cost of small quality loss.

In all cases, we observe that the total Gain$(\cdot)$ increases with larger k with diminish return (not shown). This is consistent with its submodularity.

6.4 Performance evaluation of BnP

In this set of experiments, we focus on the impact of the number of hops h (for lower bound computation) to the performance of BnP. We fixed k as 1. Besides running time, we investigate the *pruning ratio* of BnP, defined as $\frac{|V|-|C|}{|V|}$, where $|V|$ is the total alert number in an alert graph G, and $|C|$ is the average size of the candidate set C (Section 4) after pruning, for all the k iterations.

Figure 7(a) and Figure 7(b) illustrate how the computation time of different components in BnP varies, and how the pruning ratio varies, respectively, while the number of explored hops h varies from 1 to 5. The result tells us the following. (1) When h increases from 1 to 3, the response time of BnP drops. Indeed, as observed from Figure 7(b), the efficiency improvement comes from the increasing number of pruned alerts. With more alerts pruned, the amount of time taken on Gain evaluation, which is the dominating cost, drops accordingly. (2) When the number of hops increases from 3 to 5, the response time of BnP increases. As the number of hops grows from 3 to 5, we can see that the pruning ratio of BnP marginally is improved from Figure 7(b); however, the amount of computation time for lower bound in BnP dramatically increases, which becomes the dominating computation cost. According to our result, when the number of explored hops is set to be 3, BnP achieves the best performance on LM alert graphs.

In addition, as shown in Figure 7(b), BnP consistently outperforms BnP$_{\text{UB}}$ in terms of pruning ratio, since the upper and lower bounds in BnP introduce more powerful pruning to reduce unnecessary computation.

6.5 Performance evaluation of MTS

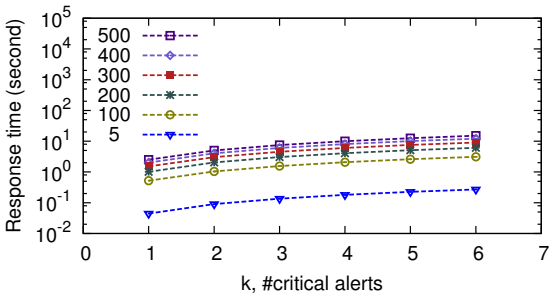
In this set of experiments, we demonstrate how the number of sampled trees affect the performance of MTS.

Figure 8(a) tells us the following. (1) While the number of required critical alerts is fixed, the response time of MTS is proportional to the number of sampled trees (varies from 5 to 500). (2) When the number of samples is fixed, the response time of MTS grows linear to the number of required critical alerts. In all cases, MTS takes no more than 15 seconds.

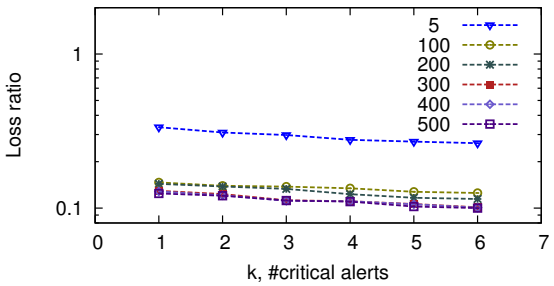
Figure 8(b) illustrates how the number of sampled trees influences the effectiveness of MTS. When the number of sampled trees increases, the loss ratio of MTS decreases, while the reduction of loss ratio diminishes. As the number of sampled trees changes from 5 to 100, the loss ratio of MTS is significantly improved; meanwhile, as the number of sampled trees changes from 100 to 500, the loss ratio is marginally improved. In addition, fixing the number of sampled trees, when the number of required critical alerts is increased, the loss ratio of all MTS variants decreases.

6.6 Scalability

On SYN alert graphs, we fixed the number of required critical alerts to be 3, and evaluate the scalability of BnP, MTS, ST, Naive, and BnP$_{\text{UB}}$. Note that the number of hops explored in BnP is fixed to be 3, and the number of sampled trees in MTS is fixed to be 300.



(a) MTS efficiency on varying #sampled trees



(b) MTS effectiveness on varying #sampled trees

Figure 8: MTS performance on LM alert graphs

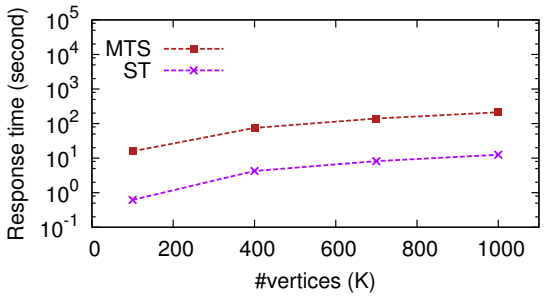


Figure 9: Scalability results on SYN graphs

Figure 9 reports the scalability results. When the number of alerts in SYN graphs increases from 100K to 1000K, the response time of MTS and ST linearly grows. In particular, when a SYN graph has 1M alerts and more than 90M edges, MTS and ST return 3 critical alerts in 4 minutes and 13 seconds, respectively. On the other hand, Naive, BnP_{UB}, and BnP cannot finish the computation in an hour, even for alert graphs with $100K$ alerts (hence are not shown). Indeed, the efficiency of BnP relies on the amount of alerts it can prune. In the worst case, it works as slow as Naive. In contrast, MTS and ST are much less sensitive to the growth of graph size, and are more promising for large alert graphs.

6.7 Summary

We summarize the experimental results as follows. (1) BnP can effectively prune unnecessary computation on real-life data, and outperforms baseline algorithms in terms of computation efficiency up to 30 times, without loss of solution quality. (2) While MTS can mine critical alerts up to 80 times faster than baseline algorithms on real-life data, the resulting loss ratio is around 0.1. (3) ST is up to 5000 times faster than baseline algorithms, with loss ratio around 0.2.

7. RELATED WORK

Causality models and analysis. Causal relations among time series data have been modeled with Granger causality [32], lagged correlation [21, 22] or Bayesian networks [28, 25], among others. Granger causality measures a cause in terms of whether it passes Granger Test, *i.e.,* whether it helps in predicating the future events, beyond what can be predicted by using only the historical events. Lagged correlation characterizes causal relations with the correlation between two time series shifted in time relative to one another. Causal Bayesian networks interprets causal relations with graphical models, in which the predecessors of a node are interpreted as directly causing the variable associated with that node.

A variety of causality mining techniques have been studied [2, 30, 31], varied with causality models. Silverstein et al. [31] proposed algorithms to mine causal relations in large databases by estimating the conditional probability of rules of interest. For Granger causality, Arnold et al. [2] applied Lasso Granger method to find a set of events that are conditionally dependent with regression, without exhaustively performing pairwise Granger Test. A toolbox for detecting Granger causality is developed [30]. These methods stop at identifying causal relations. Our work, on the contrary, efficiently identifies the most critical alerts rather than suggesting all possible causal relationships. On the other hand, efficient causality mining techniques, as well as existing knowledge bases on event causality scenarios [8] serve as preprocessing in our critical causal mining framework.

Root cause analysis. We are aware of a range of domain-specific studies that aims to find the "root causes". Given a set of observed symptom events, the problem is to identify the set of root causes that can best explain the symptom. In intrusion detection, Julisch [14] leveraged alert clustering techniques to indicate root causes for system alarms. A hierarchical clustering process is iteratively performed over groups of similar alarms, until the top causes are identified. In network performance diagnosis, Mahimkar et al. [21, 22] proposed methods to identify potential root causes as the events that have statistically significant (lagged) correlations with a set of known symptom events. In contrast, we propose a general computational framework for efficient root cause analysis over large-scale alert sequences in networks. While we do not have the luxury to assume the access of rich domain-specific semantics that benefit event filtering, any such knowledge serves as preprocessing to reduce the input size of our problem.

Influence maximization. Node influence evaluation aims to select a group of nodes with maximized influence, under various information diffusion models, such as independent cascade model [19], linear threshold model [17], competing model [3, 13], continuous-time model [9, 29], and credit distribution model [11]. The problem is, however, highly intractable (#P-hard). Sampling methods such as Monte Carlo simulations are usually applied to estimate node influence. Nonetheless, these approaches typically take massive amount of computation time and are hard to scale over large graphs [20]. To improve the scalability, various prun-

ing algorithms have been proposed to reduce the number of Monte Carlo simulations [7, 9, 12, 35], and heuristic algorithms have been studied to estimate node influence [4, 5, 6, 27]. In contrast to these works, we identify efficient algorithms for critical alert mining, with desirable performance guarantees on alert quality and efficiency. Striking a balance between mining quality and efficiency, these algorithms suggest scalable mining for large scale alert analysis.

8. CONCLUSION

We have studied the critical alert mining problem. Despite of the intractability, we developed approximation algorithms with quality guarantees, and proposed fast heuristics. We have verified, analytically and experimentally, that our algorithms are able to find top critical alerts with good quality, and scale well over large alert graphs.

This work is a first step towards mining critical alerts large-scale networks. We are conducting experiments over various large real-life datasets and causality models. One topic is to extend our techniques for distributed network monitoring systems and data centers. In addition, to further improve the alert quality, one wants to combine the alert mining framework with external semantics and knowledge bases, and to automatically interpret the alert graphs and critical alerts, for various application domains.

9. REFERENCES

[1] S. O. Al-Mamory and H. Zhang. Intrusion detection alarms reduction using root cause analysis and clustering. *Computer Communications*, 32(2):419–430, 2009.

[2] A. Arnold, Y. Liu, and N. Abe. Temporal causal modeling with graphical granger methods. In *SIGKDD*, 2007.

[3] C. Budak, D. Agrawal, and A. El Abbadi. Limiting the spread of misinformation in social networks. In *WWW*, 2011.

[4] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *SIGKDD*, 2010.

[5] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *SIGKDD*, 2009.

[6] W. Chen, Y. Yuan, and L. Zhang. Scalable influence maximization in social networks under the linear threshold model. In *ICDM*, 2010.

[7] S. Cheng, H. Shen, J. Huang, G. Zhang, and X. Cheng. Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. In *CIKM*, 2013.

[8] O. Dain and R. K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *ACM workshop on Data Mining for Security Applications*, volume 13, 2001.

[9] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *NIPS*, 2013.

[10] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, 2011.

[11] A. Goyal, F. Bonchi, and L. V. Lakshmanan. A data-based approach to social influence maximization. *VLDB*, 2011.

[12] A. Goyal, W. Lu, and L. V. Lakshmanan. Celf++: optimizing the greedy algorithm for influence maximization in social networks. In *WWW*, 2011.

[13] X. He, G. Song, W. Chen, and Q. Jiang. Influence blocking maximization in social networks under the competitive linear threshold model. In *SDM*, 2012.

[14] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *TISSEC*, 2003.

[15] K. Julisch and M. Dacier. Mining intrusion detection alarms for actionable knowledge. In *SIGKDD*, 2002.

[16] B. Karrer and M. E. J. Newman. Random graph models for directed acyclic networks. *Phys. Rev. E*, 2009.

[17] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *SIGKDD*, 2003.

[18] S. Kim and E. N. Brown. A general statistical framework for assessing granger causality. In *ICASSP*, 2010.

[19] T. Lappas, E. Terzi, D. Gunopulos, and H. Mannila. Finding effectors in social networks. In *SIGKDD*, 2010.

[20] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *SIGKDD*, 2007.

[21] A. Mahimkar, J. Yates, Y. Zhang, A. Shaikh, J. Wang, Z. Ge, and C. T. Ee. Troubleshooting chronic conditions in large ip networks. In *CoNEXT*, 2008.

[22] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large iptv network. In *ACM SIGCOMM Computer Communication Review*, 2009.

[23] D. S. Mitrinović, J. E. Pečarić, and A. Fink. Bernoulli's inequality. In *Classical and New Inequalities in Analysis*, 1993.

[24] J. Moore, J. Chase, K. Farkas, and P. Ranganathan. Data center workload monitoring, analysis, and emulation. In *Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2005.

[25] K. P. Murphy. *Dynamic bayesian networks: representation, inference and learning*. PhD thesis, University of California, 2002.

[26] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions-I. *Mathematical Programming*, 14(1):265–294, 1978.

[27] H. Nguyen and R. Zheng. Influence spread in large-scale social networks–a belief propagation approach. In *PKDD*, 2012.

[28] U. H. Nielsen, J.-P. Pellet, and A. Elisseeff. Explanation trees for causal bayesian networks. In *UAI*, pages 427–434, 2008.

[29] M. G. Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. In *ICML*, 2012.

[30] A. K. Seth. A matlab toolbox for granger causal connectivity analysis. *Journal of neuroscience methods*, 2010.

[31] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. *Data Mining and Knowledge Discovery*, 2000.

[32] J. Tian and J. Pearl. Probabilities of causation: Bounds and identification. *Annals of Mathematics and Artificial Intelligence*, 2000.

[33] A. W. Van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 2000.

[34] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[35] C. Zhou, P. Zhang, J. Guo, X. Zhu, and L. Guo. Ublf: An upper bound based approach to discover influential nodes in social networks. In *ICDM*, 2013.

Appendix: Proofs

Proof of Theorem 2. We focus on showing the function $\mathsf{Gain}(\cdot)$ has diminishing return as follows.

(1) One could verify when $u \in S_1$, $u \in S_2$, or $u = v$, $P_f(S_1 \cup \{v\}, u) - P_f(S_1, u) \geq P_f(S_2 \cup \{v\}, u) - P_f(S_2, u)$;

(2) For $u \notin S_2 \cup \{v\}$, we prove the diminishing return by mathematical reduction.

(a) Assume that all u's parents u' satisfy $P_f(S_1 \cup \{v\}, u') - P_f(S_1, u') \geq P_f(S_2 \cup \{v\}, u') - P_f(S_2, u')$.

(b) When u has only one parent, $P_f(S \cup \{v\}, u) - P_f(S, u) = f_e(u', u)(P_f(S \cup \{v\}, u) - P_f(S, u))$, and it is easy to see the diminish return for u.

(c) Assume that when u has m parents, we have $a - b \geq c - d$, where $a = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_1, u'))$, $b = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_1 \cup \{v\}, u'))$, $c = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_2, u'))$, and $d = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_2 \cup \{v\}, u'))$. Note that $b \geq d$. Consider the case when u has $m + 1$ parents, and *w.l.o.g.*, u'' is the $m + 1$-th parent satisfying $x_3 - x_1 \geq x_4 - x_2$, where $x_3 = P_f(S_1\{v\}, u'')$, $x_1 = P_f(S_1, u'')$, $x_4 = P_f(S_2 \cup \{v\}, u'')$, and $x_2 = P_f(S_2, u'')$, where $x_1 \leq x_2$. Therefore, we have $a(1 - x_1) - b(1 - x_3) = a - b - ax_1 + bx_3 = (1 - x_1)(a - b) + (x_3 - x_1)b$, and $c(1 - x_2) - d(1 - x_4) = c - d - cx_2 + dx_4 = (1 - x_2)(c - d) + (x_4 - x_2)d$. Thus, we obtain $a(1 - x_1) - b(1 - x_3) \geq c(1 - x_2) - d(1 - x_4)$, which is $P_f(S \cup \{v\}, u) - P_f(S, u) = f_e(u', u)(P_f(S \cup \{v\}, u) - P_f(S, u))$.

In all the cases, when $u \notin S_2 \cup \{v\}$, its diminishing return holds. Hence, $\mathsf{Gain}(\cdot)$ is a submodular function. It is known that for maximizing a submodular function, a greedy strategy achieves $1 - \frac{1}{e}$ approximation ratio [26]. Theorem 2 hence follows.