Spatio-Temporal Patterns for Problem Determination in IT Services

Shubhadip Mitra, Partha Dutta, Shivkumar Kalyanaraman, Prashant Pradhan
IBM India Research Laboratory, Bangalore, India

Abstract

Problem determination in a large and dynamic IT service is a challenging task. In this paper we propose a framework for problem determination based on monitoring the event streams generated by the different components of an IT service. We give a generic representation of a problem through spatial-temporal patterns*, which is a graph where the vertices capture the location and the time of the matching events, and the edges represent the spatio-temporal conditions between two matching events. The spatial conditions are based on the underlying system topology graph, and the temporal conditions are based on event timestamps.*

A practical implementation of the above framework will require fast algorithms for detecting patterns. We present efficient algorithms when the pattern graph is a range and a tree, which are then used as building blocks for a hierarchical heuristic for detecting general patterns. Finally, we show that our algorithms perform well in practice through extensive numerical simulations.

1 Introduction

1.1 Motivation

As enterprise IT systems become larger and more dynamic, IT service management is becoming an increasingly challenging task. One of the central concerns in IT service management is how to reduce the time required to detect problems in an IT system. This is an important concern because a fault or performance degradation in the underlying IT system may directly impact the end-user's experience of the service. In most large IT systems, due to the complex interdependencies among various components (both software and hardware), the symptoms or root cause of a problem may not be restricted to a single component, and the root cause may not even be a hard fault at a component. In addition, the management of IT systems are frequently outsourced, and the system administrators may occasionally change. In such a setting, reusing the knowledge of the previously resolved problems is essential for quickly detecting new occurrences of problems.

In this context, we present a framework for problem determination based on reusing information from already resolved problems. Suppose we are given some streams of primitive events, where each stream consists of a sequence of timestamped events generated by a component of an IT service (e.g., events indicating that the memory-usage at a server has crossed a threshold, or the time taken to complete an activity in a workflow has crossed a threshold). We propose a method to automatically detect new occurrences of a problem based on the information stored about previously resolved problems. Our framework (1) represents information about a problem as a graph pattern, and stores these patterns in a repository, (2) as new events are generated, searches the event streams for a set of events that match some pattern in the repository, and (3) raises an alarm when a pattern match is found. Building such a tool presents the following three challenges. First we need to log events at different components of the service, and aggregate them in a central location. Second, we need to represent problems in a way that is reusable across system topology changes, and possibly, in different instantiation of the same service. Finally, we need algorithms that can efficiently search for patterns over a collection of event streams.

Today, there are multiple established commercial IT service management products that log events at various components of an IT service and aggregates them in a central location, e.g., IBM Tivoli/Netcool Suite and HP Network Node Manager. In this paper we focus on the remaining two challenges—finding a generic representation of the problems, and designing algorithms to detect occurrences of patterns in event streams.

1.2 Contributions

We make the following contributions in this paper.

1. We present a representation of problems using spatio-temporal patterns in Section 2. We assume that we are given a system topology that captures the dependencies between the various components (or nodes) of the service. Every event has an associated *node type* based on the type of node where the event occurred (e.g., a billing server, a database), and an associated *event type* based on the type of event (e.g., number of requests/queries per second has crossed a thresh-

old). The temporal relationship between any two events is based on their time of occurrences, and their spatial relationship is based on the nodes in the topology where the events occur. A *spatio-temporal pattern* is a graph where each vertex has a node type and an event type, and an edge specifies the spatial and temporal conditions between end-vertices. Roughly speaking, a set of events S in a collection of event streams matches a spatio-temporal pattern if every vertex in the pattern has a matching event in S with the same node and event types, and all temporal and spatial conditions given by the edges in the pattern are satisfied by the matching events.

2. Next, we present algorithms for detecting if a pattern match occurs in an event stream. We assume that type of conditions captured by the edges are of a special kind: upper bounds on the (spatial or temporal) distance between the two events, e.g., the events are not more than 2 seconds apart in time, or at most at distance two in the topology graph. Such conditions are commonly considered in problem determination in IT systems [2], because the symptom events of a problem typically exhibit some proximity in time or space. For such distance-based conditions, we propose two efficient algorithms, first one for the patterns which requires that all matching events occur within a fixed interval of each other, and the second one for the patterns where the pattern graph is a tree. We use these two algorithms presented in Section 3 as building blocks for a hierarchical heuristic for general pattern graphs. Section 4 presents our proposed framework that detects spatio- temporal patterns on event streams. We demonstrate that our algorithms are efficient in practice through extensive numerical simulation in Section 5.

2 Spatio-Temporal Pattern Matching Problem

Topology. We model the underlying system topology using a *topology graph* $G^T = (V^T, E^T)$ where the components of the IT service are modelled as nodes and dependencies between two components are modelled as edges. There is a fixed set NTYPE of possible types of node. Each node $v \in V^T$ has an attribute $v.ntype \in$ NTYPE . (In this paper, we do not consider the case where dependencies have attribute.) The topology graph is dynamic, i.e., node can join or leave the system and dependencies can be deleted or new dependencies formed. The spatial distance between two nodes is the length of the shortest path between the two nodes in G^T.

Events. We model time using tick of a global clock that takes integer values starting from 0. An event e is recorded (or logged) with the following attributes: (1) the time of its occurrence $e.ts$ (also called the timestamp of the event), (2) node in the topology where the event occurred $e.node \in V^T$ (and $e.ntype$ is defined to be $(e.node).ntype$), and (3) the type of the event $e.etype \in$ ETYPE. Here, ETYPE denotes the fixed set of all possible event types. The temporal distance between two events e_i and e_j, denoted by $\Delta_t(e_i, e_j)$, is given by $|e_i.ts - e_j.ts|$. The spatial distance, denoted by $\Delta_s(e_i, e_j)$, is given by the spatial distance between $e_i.node$ and $e_j.node$ in G^T. For each node $v \in V^T$, the event stream $v.es$ is a sequence of events at v, ordered by their timestamps. The event stream is dynamic: new events are appended to $v.es$ as they occur at node v.

Problem Patterns. We represent a problem through a labelled *pattern graph* $P = (V_P, E_P)$ where $V_P = \{V_1, \ldots, V_k\}$. Each vertex[1] $V_i \in V_P$ has two attributes $V_i.ntype \in$ NTYPE and $V_i.etype \in$ ETYPE. No two vertices in the pattern have the same node type, or have the same event type. Each edge of the pattern $(V_i, V_j) \in E_P$ has two attributes [2]: the upper bound $\delta(i, j)$ on spatial distance and upper bound $\tau(i, j)$ on temporal distance.

Spatio-Temporal Match: We say that an ordered k-tuple of nodes in the topology graph $W =< w_1, \ldots, w_k >$ *spatially matches a pattern* P if the following two conditions hold: (1) For every $w_i \in W$, $w_i.ntype = V_i.ntype$, and (2) for every edge (V_i, V_j) in P, the spatial distance between w_i and w_j in G_T is at most $\delta(i, j)$. We say an ordered k-tuple of events $Y =< e_1, \ldots, e_k >$ *matches a pattern* P, if the following two conditions hold: (1) $< e_1.node, \ldots, e_k.node >$ spatially matches P, and (2) for every edge (V_i, V_j) in P, the temporal distance between e_i and e_j is at most $\tau(i, j)$. (In short, (1) for every node V_i of P, $V_i.ntype = e_i.ntype$ and $V_i.etype = e_i.etype$, and (2) for every edge (V_i, V_j) of P, $\Delta_s(e_i, e_j) \leq \delta(i, j)$ and $\Delta_t(e_i, e_j) \leq \tau(i, j)$.)

A set of k events matches a pattern P if some ordering of those events matches P. We say that a set of event streams matches a pattern P if the there is a set of k events in the event streams that matches P. *In this paper, given a set of event streams where events are appended over time, we investigate how to efficiently detect the occurrence of pattern matches.*

3 Algorithms

Our algorithm for pattern search is based on the simple idea of hierarchical *pruning* of events: incrementally delete the events from the event streams using one or more *relaxation* of the original pattern until a match is found, or all events are pruned. If we view a pattern P as a set S of conditions (such that a matching set of events needs to satisfy all the conditions in S), then a relaxation of the pattern is a subset S_1 of S. Obviously, if we prune events from the event streams that do not satisfy conditions S_1, we cannot

[1] To distinguish nodes in the topology graph from nodes in the pattern graph, we call the latter, vertices.

[2] Our results hold even if a vertex has one or both of the attributes. However, for simplicity of presentation, we assume both attributes are present for each edge.

miss any match of P. Example of relaxation of a pattern are any subgraph of the corresponding pattern graph (removing an edge or vertex from the pattern graph, removes the associated conditions), or a set containing only the spatial conditions from the pattern (i.e., does not contain the temporal upper bound on the edges of the pattern).

We first describe how to find temporal pattern matches assuming that we have already selected the topology nodes (and their event streams) that satisfies the spatial conditions of the pattern (i.e., the conditions on $V_i.ntype$ of pattern vertices V_i, and the spatial upper bound $\delta(i,j)$ on the pattern edges). We describe algorithms to find events that satisfies the temporal conditions of the pattern. We first describe with an algorithm for a simple pattern graph, and then show how to extend the algorithm to arbitrary pattern graphs.

3.1 Temporal Matching Algorithms

In the rest of this section (except Section 3.5), we assume that there are k event streams $es_1, \ldots, es_k$, and pattern $P = (V_P, E_P)$ with k vertices $V_1, \ldots, V_k$, and each edge (V_i, V_j) has a finite temporal upper bound $\tau(i,j)$ but its spatial upper bound is ∞. New events are appended to the event streams over time, and we need to raise an alarm whenever, there are k events $e_1, \ldots, e_k$ such that (1) for all i, $e_i \in es_i$, and $e_i.etype = V_i.etype$ and (2) for every edge (V_i, V_j) in P, $\Delta_t(e_i, e_j) \leq \tau(i,j)$. We now describe algorithms for different kinds of patterns.

3.2 Range Patterns

In this section, we consider a basic pattern P where all events in a pattern match are required to be within a constant (temporal) distance Δ of each other. In other words, the pattern graph is a clique, and for every edge (V_i, V_j), $\tau(i,j) = \Delta$. We call this pattern, a range pattern, since all events in a match are required to be within a time range. In Figure 1, we give an algorithm for detecting this pattern.

The algorithm is based on a straightforward idea. It maintains a time window from the current event to Δ time units in the past, and counts the number of events of each event types in that window. At any point in the execution, if the window has at least one event of each type, then there is a pattern match. We now describe the algorithm in detail.

Upon occurrence of a new event in any of the k event streams, if the event matches the $etype$ of the pattern vertex corresponding to that stream, the event is put in a new merged stream STR. For every event, the algorithm also maintain a boolean attribute $matched$ (initially set to false), and an attribute $str \in \{1, \ldots, k\}$, initialized to the stream in which e occurred. It also maintains three event pointers in STR: $first$, $last$ and $scan$. The $last$ and the $first$ pointers mark the start and the end of the current window in STR. When there is an event of each of the k types in the current window, there is a pattern match, and the $scan$ pointer is

```
1:  upon arrival of event e at event stream j do
2:  e.matched := false; e.str := j
3:  if e.etype ≠ V_j.etype then return
4:  counter[j] := counter[j] + 1      /*e.etype = V_j.etype*/
5:  put e in STR; first := e
6:  while last.ts < first.ts − Δ do
7:      counter[last.str] := counter[last.str] − 1
8:      point last to the next event in STR
9:  if scan.ts < last.ts then scan := last
10: if ∀i ∈ {1, ..., k}, counter[i] > 0 then
11:     raise a pattern match alarm
12:     while scan.ts ≤ first.ts do
13:         scan.matched := true
14:         if scan ≠ first then point scan to the next event in
            STR
15: return
```

Figure 1. Range Matching Algorithm

used to mark the events in the window as part of of a match (by setting the $matched$ attribute of those events to true). Note that, even though an event can be a part of multiple matches, the scan pointer is incremented such that it visits each event at most once.

Note that, it is straightforward to reduce the amortized time taken to check the condition in line 10 from $\tilde{O}(k)$ to $\tilde{O}(1)$. Hence the following theorem.

Theorem 1. *The algorithm Alg_{range} raises a pattern match alarm if and only if there is a set of events matching the pattern in the current window. Also, the amortized update time of the algorithm is $\tilde{O}(1)$ per event.*

Due to lack of space, the proof is omitted.

3.3 Tree Patterns

We now present an algorithm for detecting pattern where the pattern graph is a tree. Like the range pattern, we only look at the temporal matching problem in this section. The pattern graph has k vertices $V_1, \ldots, V_k$, and there are k event stream $1, \ldots, k$ corresponding to each vertex of the pattern graph. Unlike the range algorithm, we consider the offline version of the problem, i.e., we assume that we are given the complete event streams as input to the problem. We will later describe how to obtain a heuristic for detecting general pattern graphs when events are appended to the event streams over time.

First, we note that an edge of a pattern graph can be viewed as a range with only two vertices. Our algorithm for tree pattern uses the algorithm Alg_{range} as a pruning subroutine. For an edge (V_i, V_j), subroutine $prune(i,j)$ executes Alg_{range} with the event streams i and j as inputs, and it removes the events from event stream i whose $matched$

attribute remain false when Alg_{range} has completely processed the two streams. (Note that, $prune(i, j)$ leaves the event stream j unchanged.)

Our tree pattern matching algorithm, that we call Alg_{tree} and which is given in Figure 2, is composed of H_P rounds of pruning, where H_P is the height[3] of the pattern graph P. In each round, each pattern vertex is visited in a Breadth First Search (BFS) order, and its corresponding event stream is pruned based on the event streams of its parent and children. If at any point in the execution, one of the pruned event streams becomes empty, then the algorithm immediately returns without a pattern match. Otherwise, if all event streams are non-empty at the end of round H_P, then the algorithm ensures that each of the remaining events in every event stream is part of some match of the pattern.

We note that, for our algorithm any vertex of the tree can be used as the root in the BFS that is used to visit the vertices in a round. However, since the running time of the algorithm increases with H_P, we should choose a root rt_P that minimizes the height of the tree. Therefore, before running our algorithm, we find such a root vertex by running BFS from each vertex of the pattern and selecting a vertex whose BFS tree has the smallest height. (This pre-computation takes $O(k^2)$ time.) In the rest of this section, we assume that the pattern graph is a tree rooted at rt_P with height H_P.

To prove the correctness of the algorithm, we need to introduce some definitions. Consider the event streams at any time t in the execution. In the rooted tree P, if vertex V_j is the parent of vertex V_i and if there exists some event e' in stream j and some event e in stream i such that $\Delta_t(e', e) \leq \tau(i, j)$, then e is said to have a *matching parent* e' at time t. Now, suppose pattern vertex V_j has c children vertices $V_{j_1}, \ldots, V_{j_c}$. An event e in stream j is said to have a *matching set of children* at time t if $\forall i \in \{1, \ldots, c\}$, there is an event e_{j_i} in stream j_i such that $\Delta_t(e, e_{j_i}) \leq \tau(j, j_i)$. We say that $\{e_{j_i} \mid i = 1, \ldots, c\}$ is a matching set of children of e. Let $T(V_i)$ denote the subtree rooted at V_i (i.e., the induced subgraph of P consisting of V_i and all its descendants). We say that an event e in stream i is *c-valid* at time t, if there is a set S of unpruned events, one from each of the event streams of the descendants of V_i, such that $S \cup \{e\}$ is a pattern match of $T(V_i)$. (By definition, if V_i is a leaf vertex, then all the events in its event stream are *c-valid*.)

Lemma 1. If an event becomes *c-valid* at some point of time in an execution, then until the event is (possibly) pruned, the event remains *c-valid*.

Proof. Suppose otherwise. Suppose event e in event stream i becomes *c-valid* at time t. Consider the first event $e' \neq e$ that is pruned such that, e is *c-valid* before e' is pruned, and e becomes not *c-valid* after e' is pruned. Let e' be pruned at time $t' > t$. Then from the definition of e', at time $t' - 1$ there is a set S of unpruned events in the event streams such that, S is a pattern match of $T(V_i)$, and $e, e' \in S$. Note that, e' is the first event to be pruned in S. Now, as S is a pattern match of $T(V_i)$, so there is a matching parent of e' and a set of matching children of e' in S. Since an event can be pruned at t' only if there is no matching parent or no matching set of children in the event streams at t', e' cannot be pruned at time t'; a contradiction. □

[3]The *depth* of a vertex is the number of edges in the path from the root to that vertex. The *height* of a tree, is maximum depth over all vertices of the tree.

```
1:  for round r = 1, ..., H_P do
2:      for V_i ∈ {V_1, ..., V_k} in the BFS order of vertices do
3:          if V_i is not the root of the pattern graph then
4:              prune(i, j), where V_j is the parent of V_i
5:          if V_i is not a leaf of the pattern graph then
6:              prune(i, j), for every child V_j of the V_i
7:          if event stream i is ∅ then return
8:  if all event streams are not-empty then
9:      raise a pattern match alarm
10: return
```

Figure 2. Tree Pattern Matching Algorithm

Lemma 2. If an execution does not return before the end of round r ($1 \leq r \leq H_P$), then at the end of round r, for every event stream i such that $depth(V_i) \geq H_P - r$, every unpruned event e in the event stream i is *c-valid*.

Proof. We prove the lemma by induction on the round number.

Base case: Suppose an execution does not return before the end of round 1. This implies that no event stream is empty at the end of round 1. At the end of round 1, consider any event e in a stream i such that $depth(V_i)$ is $H - 1$ or H. Note that, either V_i is a leaf vertex or all children of V_i are leaf vertices. From the definition of *c-valid*, any unpruned event e' in a stream corresponding to a leaf vertex is *c-valid*. Suppose V_i is not a leaf vertex. Consider the time t when line 6 is completed for V_i in round 1. Then e must have a matching set S of children at time t (otherwise, it would have been pruned in line 6 of round 1). Thus e is *c-valid* at time t. Since e is not pruned in round 1, using Lemma 1, e remains *c-valid* at the end of round 1.

Induction Hypothesis: If an execution does not return before the end of round r, then at the end of round r, for every event stream i such that $depth(V_i) \geq H_P - r$, every unpruned event e in the event stream i is *c-valid*.

Induction Step: Suppose an execution does not return before the end of round $r + 1$. This implies that no event stream is empty at the end of round $r + 1$. Let S be the set of all unpruned events, at the end of round $r + 1$, which occur at event streams at depth $H_P - r$ or higher. From the induction hypothesis, events in S were *c-valid* at the end of round r. Then, using Lemma 1, events in S remain *c-valid* at the end of round $r + 1$.

Now consider an unpruned event e at the end of round $r+1$, which occur at event stream i such that $depth(V_i) = H_P - (r+1)$. If V_i is a leaf vertex, then from the definition of *c-valid*, e is *c-valid* at the end of round $r+1$. Suppose that V_i is not a leaf vertex. Consider the time t when line 6 is completed for V_i in round $r+1$. Since e is not pruned in round $r+1$, when line 6 is executed for V_i, e has a matching set of children, say $e'_1, \ldots e'_c$, occurring at event streams of vertices $W_1, \ldots, W_c$, respectively. Note that, all W_is have depth $H_P - r$. Then, from the induction hypothesis, all events e'_i are *c-valid* at the end of round r. Also, since vertices are visited in BFS order in a round, in round $r+1$ the pruning of the event streams of W_is occur after time t. Therefore at time t, for each e'_i, there is the set of unpruned events S'_i (that includes e'_i) such that S'_i is a pattern match of $T(W_i)$. Union of all sets S'_i, and the singleton $\{e\}$, gives a set of unpruned events which is a pattern match of $T(V_i)$ at time t. Thus, e is *c-valid*at time t. From Lemma 1, e remains *c-valid* at the end of round $r+1$. □

We say that an unpruned event e is part of a pattern match of P at some time t, if there is a set S of unpruned events at time t, such that $e \in S$ and S is a pattern match of the pattern graph P. The depth of an event e that occurs at event stream i is defined as $depth(V_i)$.

We are now ready to show the correctness of the algorithm. The next two lemmas show that an event remains unpruned at the end of H_P rounds if and only if the event is part of a pattern match of P.

Lemma 3. At the end of round H_P, every unpruned event is a part of some pattern match of P.

Proof. Suppose by contradiction, at the end of round H_P, there is an unpruned event that is not a part of any pattern match of P. Consider such an event e with the minimum depth. Suppose that event e occurs in event stream i. Suppose V_j is the root of the pattern graph. From Lemma 2, at the end of round H_P, every unpruned event e' in the event stream j is *c-valid*. As V_j is the root vertex, from the definition of *c-valid*, e' is part of a pattern match of P at the end of round H_P. Thus, e does not occur at the root V_j.

Since e is not pruned in round H_P, and the pruning in done in BFS order in a round, e has an unpruned matching parent e'' at the end round H_P. Suppose, e'' occurs in event stream x (i.e., V_x is the parent of V_i in the pattern graph). Since at the end of round H_P, e is an event with minimum depth that is not a part of some pattern match of P, e'' is part of some pattern match of P. Let us denote the set of events in this match of P that contains e'' by $S(e'')$. Since $S(e'')$ is a match of P, there is a subset of $S(e'')$ that matches $T(V_i)$; let us call this subset $S_i(e'')$.

Now from Lemma 2, e is *c-valid* at the end of round H_P. Thus, there is a set of unpruned event $S(e)$ (which includes e) such that $S(e)$ matches $T(V_i)$. The set $(S(e'') \setminus S_i(e'')) \cup S(e)$ is a pattern match of P containing e, a contradiction. □

Lemma 4. Any event that is a part of a pattern match of P at the beginning of an execution, is never pruned during the execution.

Proof. Consider any event e that is part of pattern match of P at the beginning of an execution. Then, there is a set S of events at the beginning of the execution that is a pattern match of P. Suppose during the execution, some event of S is pruned. Consider the first such pruned event e' in S, say at time t. Then at time $t-1$, since no event in S has yet been pruned, e' has unpruned matching parent and an unpruned set of matching children. Thus, e' cannot be pruned at time t. It follows that, no event in S (including e) ever gets pruned during the execution. □

Theorem 2. *(1) An event is unpruned at the end of round H_P if and only if the event is part of some pattern match of P. (2) Suppose there are n events in each of the k event streams. Algorithm 2 requires in $O(k^2 n)$ time and $O(kn)$ space to process all the events.*

Following the above stated results, we omit the straightforward proof.

3.4 A Heuristic for General Pattern Graphs

As we discussed at the beginning of Section 3, to prune for a given pattern P, if we consider only a subset S of all the conditions in P, then none of the events that is part of a match of P will be pruned. However, these unpruned events might contain some events that are not part of any pattern match of P. We call S a relaxation of P. We use the idea of relaxation of a pattern to prune events using various subgraphs of the pattern graph P.

We now describe our heuristic based on hierarchical pruning using subgraphs of P. It consists of three steps. If at any point of the execution any event stream is completely pruned, then the execution terminates without a match.

Step 1: For each edge e in P, we execute separate instances of Alg_{range}, denoted by $Alg_{range}(e)$. The inputs to $Alg_{range}(e)$ are two event streams corresponding to the two end vertices of e in P, and we prune any event with $matched = false$ after $Alg_{range}(e)$ is executed. The instances of Alg_{range} are executed one after the other: the unpruned events from an instances are the input events for the next instance.

Step 2: Next, we execute m instances of Alg_{tree} with distinct trees $T_1, \ldots, T_m$ as the pattern graph. Here, each tree T_i is a subgraph of P. (We discuss the trade-offs involved in choosing T_is and m below.) The m instances of Alg_{tree} are executed one after the other.

Step 3: Finally, we take all the unpruned event at the end of Step 2 and perform an exhaustive search for pattern P. An exhaustive search takes every possible combination of unpruned events and checks if all the conditions in P are

true for any combination. Here, a combination is a set of events that has one event from each event stream. A pattern match alarm for P is raised if a combination is found that satisfies all conditions of P.

Suppose that there are at most n events in each of the event stream and the number of edges k in the pattern graph is small compared to n. From Theorems 2, we know that an instance Alg_{tree} runs in time linear in n. Thus running time of Step 2 increases linearly with mn, but we expect that increasing m decreases the number of unpruned events at the end of Step 2. On the other hand, if there are u unpruned events when Step 3 is triggered, the exhaustive search may required $O(u^k)$ time. Thus there is a trade-off between the running times of Step 2 and Step 3, which can be tuned by choosing appropriate T_is and m. We empirically investigate this problem in our simulations.

3.5 Spatial Pattern Matching

In this section we discuss algorithms for spatial pattern matching. First note that, the spatial conditions on the edges of the pattern graph P specifies an upper bound on the distance between the two nodes in the topology graph G^T. Therefore we maintain the distance matrix for G^T which gives the distance between any pair of nodes in G^T. Whenever G^T changes (and at initialization), this distance matrix is recomputed in $O(N^3)$ time using the Floyd-Warshall algorithm [5]. Here N is the number of nodes in G^T. Since, changes in G^T are infrequent (compared to the rate at which new events are generated), we do not expect distance matrix re-computations to be frequent.

Now, for each pattern vertex V_i in P, there might be multiple nodes in G^T that has the same $ntype$; we denote the set of these nodes by $W(V_i)$. Let M be the maximum size of $W(V_i)$ over all vertices V_i of P. (Note than, M can be $O(N)$.) We now describe how to prune nodes in G^T for a spatial pattern match.

To prune $W(V_i)$ for spatial match of an edge (V_i, V_j) of P that has spatial upper bound $\delta(i, j)$, we execute the following obvious algorithm. For each node w_i in $W(V_i)$, we check whether any node in $W(V_j)$ has distance less than equal to $\delta(i, j)$ from w_i. If there is no such node, w_i is pruned. This check requires at most M lookups in the distance matrix for each vertex in $W(V_i)$, and therefore, $O(M^2)$ time in total for $W(V_i)$. Let us denote this pruning algorithm by $Alg_{sp}(i, j)$.

We note that, we cannot reuse the linear time algorithm Alg_{range} for the spatial match of an edge because Alg_{range} exploits the ordering of events that is imposed by time. However, for spatial pruning when the pattern graph is a tree, we can reuse the Alg_{tree} algorithm by making the following two substitutions for all i, j: (1) for $prune(i, j)$, use $Alg_{sp}(i, j)$ instead of Alg_{range}, and (2) for input, use nodes in $W(V_i)$ instead of event stream i. The correctness of using Alg_{tree} for spatial pruning follows from the following observation: if we assume $prune(i, j)$ correctly prunes the nodes in $W(V_i)$ (or the events in event stream i) for the condition on edge (V_i, V_j), then the proof of correctness of Alg_{tree} is independent of whether the conditions on the pattern edges are spatial or temporal. From Theorem 2, the time-complexity of Alg_{tree} for spatial match is $O(k^2M^2)$.

Similarly, the heuristic in Section 3.4 can be used for spatial matching if Alg_{range} is replaced by $Alg_{sp}(i, j)$ and $Alg_{sp}(j, i)$. To obtain a complete spatio-temporal match, for each spatial match $< w_1, \ldots, w_k >$ obtained in Step 3 of the heuristic for spatial match, the heuristic for temporal match is triggered with events from node w_i used as event stream i.

4 A Proposed Framework for Problem Determination

In this section we briefly propose a scalable framework for problem determination based on the algorithms proposed in this work. We classify occurrences of a problem into two categories, *resolved problems instances* and *new problems instances*. For a problem type, we assume that there is an existing problem pattern constructed out of resolved problem instances. A system administrator may construct such a pattern, either manually, or with the aid of a learning technique [2]. Techniques from temporal data mining [8] for discovering hidden patterns in temporal data can also be used for constructing problem patterns.

Every problem pattern is stored in central repository along with any other relevant information, e.g., a method to resolve the problem. Now, consider a pattern P in the repository. In P, let S_E and S_N be the set of all event types and node types. Let W be the longest path in the weighted version of pattern graph P where the edges have weights equal to their temporal upper bounds. Now, if a set S' of events is a pattern match of P, then all events of S occurs within W time units of each other. Next, on the set of event streams that are generated by all nodes with node types in S_T, we execute the algorithm Alg_{range} using a range pattern consisting of a set of event types S_E and $\Delta = W$. Whenever this execution raises a pattern match for P', using pattern P, we run the heuristics from Sections 3.4 and 3.5 over all events that occurs within last W time units. This scheme is run separately for each pattern in the repository. Whenever, any of these schemes raises a pattern match, an alarm is sent to the system administrator. For obtaining the spatial information, the system topology can be discovered and maintained using established commercial products, e.g., IBM Tivoli Application Dependency Discovery Manager (TADDM) and HP Discovery and Dependency Mapping Software, and the distance matrix is updated whenever a topology change occurs.

5 Performance Study

To evaluate the performance of our algorithms, we performed extensive numerical simulations. We generated a set of pattern graphs and investigated how efficiently they can be detected on event streams using our algorithm. For tree patterns we used Alg_{tree} and for general patterns we used the heuristic described in Section 3.4. We did our simulations only for temporal patterns. The temporal condition on each edge of the pattern is an upper bound of 20 millisecond (ms).

The simulations were performed on an Intel dual core processor, each of capacity 2.16 GHz. The machine was running Windows XP with 2GB memory. The algorithms were implemented in C++.

Let n be the number of primitive events in each of the k event streams at the beginning of the experiment. Suppose at the end of the experiment, there are n_i unpruned events in event stream i. Then, the the number of distinct pattern matches should be at least $\max_{1\leq i\leq k}\{n_i\}$ and at most $\prod_{i=1}^{k} n_i$ distinct matches. Suppose after round r, n_i^r events are left unpruned in event stream i. Then, $z_r = \max_{1\leq i\leq k}\left\{\frac{n_i^r}{n}\right\}$ indicates the minimum possible fraction of distinct pattern matches at the end of round r.

Our experiments were performed on randomly generated spanning trees of a complete graph, unless otherwise specified. We considered two kinds of event distribution, namely, events are generated by a Poisson process, and events with uniform distribution of inter-arrival time. In both cases we kept the same mean inter-arrival time. Let λ be the mean inter-arrival duration between the events. Figure 3(a) shows the performance variation with λ for both the distributions. We fixed $n = 20000$ and $k = 65$ and varied λ from 18 ms. to 26 ms. and measured the variation of the running times. Figure 3(b) plots z_r for $1 \leq r \leq 5$ for three different values of λ for Poisson distribution. Observe that for a fixed n and k, as λ increases, the potential number of matches decreases. As a consequence, z_r converges sharply with the number of iterations. Since the number of unpruned events decreases, the time taken for each subsequent round decreases. Therefore, the total running time of the experiment decreases with increase of λ.

Figure 3(c) shows that for a fixed pattern and a fixed λ, the running time varies almost linearly with n, and thus confirming our time-complexity analysis of Alg_{tree} in Theorem 2. Figure 3(d) shows the variation of running time with increase in k. For this experiment we use a pattern that is a binary tree of k vertices. From the proof of Theorem 2, we see that the time complexity of Alg_{tree} can also be given by $O(H_P kn)$ (instead of $O(k^2 n)$) where H_P is the height of the rooted tree with minimum height, for pattern P. For a binary tree, with the increase in the number of vertices k, the height H_P grows logarithmically. Thus, our experimental results show sub-quadratic growth of the running time with k.

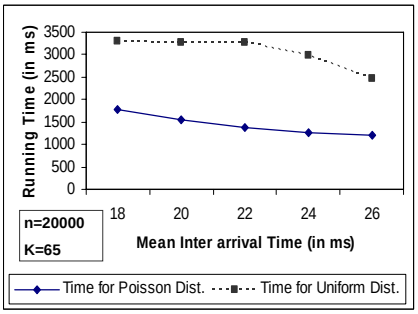
Next we examine the performance of our heuristic. We considered a pattern graph with $k = 80$ vertices. n was fixed to 40000 and λ was fixed to 20 ms. We selected three spanning trees T_1, T_2 and T_3 of the pattern. The unpruned events left after running the tree algorithm using pattern T_1 were given as input to the next execution of the algorithm using T_2. And this was followed up by execution of the algorithm using T_3. By the end of the three executions, the fraction of unpruned events had significantly reduced so that we could perform exhaustive search and detect the matching patterns. We illustrate the decline of z_r for $1 \leq r \leq 5$ over these three passes in Figure 3(e). It clearly demonstrates the effectiveness of our heuristic for general patterns.

6 Related Work

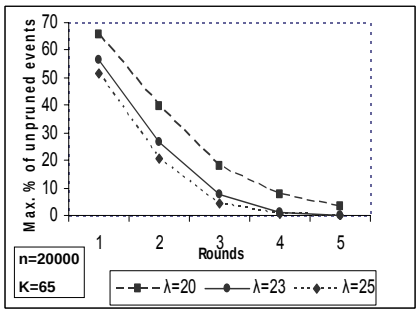
Problem determination and root cause analysis has been extensively studied in the IT service management literature [1–4, 6, 9]. In [2], a repository of problem pattern is maintained, where a problem (or fault) pattern is a set of change-point events occurring within a fixed time interval. Each event in a problem pattern is assigned a score which indicates the correlation between an event and the occurrence of the problem. The authors present a method to learn the problem pattern and the associated scores over time. The method described in their work is complementary to our work, and our Alg_{range} algorithm can be directly used to detect if a set of events in a problem pattern has occurred.

Searching for a given sequential pattern (or a given subsequence) in very long sequences has been studied in the area of temporal data mining [8]. However, the graph based patterns that we consider are more general than sequential patterns as they allow more flexibility in how the matching events of a pattern are ordered. Detecting whether an event pattern (also called complex events) has occurred in an event stream has been studied in the area of complex event processing. Recent work in this area has studied detection of complex events that have temporal constraints among the basic events [7, 10]. In [10], the authors study complex events with temporal conditions in the setting of RFID streams, and considers many different kinds of temporal constraint, e.g., both upper and lower bounds on the distance between the events, and event which occur over an interval of time (in contrast to instantaneous events). While some of those constraints can be handled by straightforward extension of our algorithms (e.g., events occurring over an interval), some others are more challenging to handle (e.g., simultaneously handling upper and lower bounds between two events). However, there are two major differences from [10] in our work: (1) in addition to temporal constraints, we also consider system topology graph based spatial constraints on events, and (2) we provide provable bounds on the running time of our algorithm for range and tree patterns (whereas, the algorithm presented in [10] provides no such time-complexity bound).

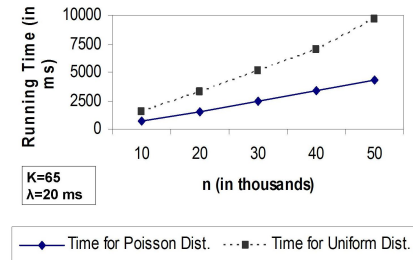
(a)



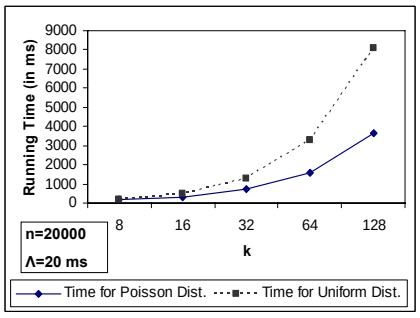
(b)



(c)



(d)



(e)

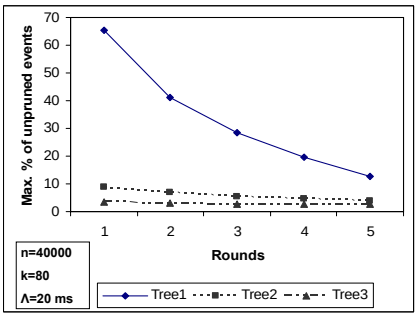


Figure 3. Simulation Results

7 Conclusion

We now briefly discuss three directions for future work. First, an immediate open problem is to design efficient algorithms for general pattern graphs. In this paper, we have given algorithms for range and tree patterns, whose running time is linear in the number of events in the event streams. However, it is left open whether such algorithms exist for general pattern graphs. Second direction for investigation, following [10], is to study efficient algorithms for more general temporal conditions on the events, e.g., events that take place over an interval, and lower bounds on the distance between two events. Finally, to make the framework easier to deploy, we need further investigation extending the work done in [2] to automatically extract problem patterns from historical event logs.

References

[1] M. K. Agarwal, M. Gupta, V. Mann, N. Sachindran, and L. Mummert. Problem determination in enterprise middleware systems using change point correlation of time series data. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006.

[2] M. K. Agarwal, N. Sachindran, M. Gupta, and V. Mann. Fast extraction of adaptive change point based patterns for problem resolution in enterprise systems. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management(DSOM)*, 2006.

[3] P. Bahl, R. Chandra, A. G. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*, 2007.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2003.

[6] X. Gu, S. Papadimitriou, P. S. Yu, and S.-P. Chang. Toward predictive failure management for distributed stream processing systems. In *ICDCS*, 2008.

[7] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. Sase: Complex event processing over streams. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[8] S. Laxman and P. S. Sastry. A survey of temporal data mining. *SADHANA, Academy Proceedings in Engineering Sciences, The Indian Academy of Sciences*, 31, 2006.

[9] I. Miloucheva, U. Hofmann, and P. A. Aranda-Gutiérrez. Spatio-temporal QoS pattern analysis in large scale internet environment. In *Interactive Multimedia on Next Generation Networks, First International Workshop on Multimedia Interactive Protocols and Systems (MIPS)*, 2003.

[10] F. Wang, S. Liu, P. Liu, and Y. Bai. Bridging physical and virtual worlds: Complex event processing for RFID data streams. In *International Conference on Extending Database Technology (EDBT)*, 2006.