PASS: A Parallel Activity Search System

Andrea Pugliese, V.S. Subrahmanian, Christopher Thomas, and Cristian Molinaro

Abstract—Given a set \( \mathcal{A} \) of activities expressed via temporal stochastic automata, and a set \( \mathcal{O} \) of observations (detections of low level events), we study the problem of identifying instances of activities from \( \mathcal{A} \) in \( \mathcal{O} \). While past work has developed algorithms to solve this problem, in this paper, we develop methods to significantly scale these algorithms. Our architecture consists of three parts: (i) leveraging past work to represent all activities in \( \mathcal{A} \) via a single “merged” graph, (ii) partitioning the graph into a set of \( C \) subgraphs, where \( (C + 1) \) is the number of compute nodes in a cluster, and (iii) developing a parallel activity detection algorithm that uses a different compute node in the cluster to intensively process each subgraph. We propose three possible partitioning methods and a parallel activity search detection (PASS Detect) algorithm that coordinates computations across nodes in the cluster. We report on experiments showing that our algorithms enable us to handle both large numbers of observations per second as well as large merged graphs. In particular, on a cluster with 9 compute nodes, PASS can reliably handle between 400K and 569K observations per second and merged graphs with as many as 50K vertices.

Index Terms—Activity Detection, Temporal Stochastic Automata, Parallel Computation

1 INTRODUCTION

There are many applications where there is a critical need to identify the activities that are going on in a stream of observation data. For instance, we are aware of a large mobile phone company that looks for specific patterns of fraud in call data records. Likewise, a bank or online market (e.g., Amazon) looks for fraudulent transactions in Web transaction logs. In [1] patterns of fraudulent stock market activity are examined—we can imagine future situations where brokerage houses wish, in real-time, to identify instances of the fraud patterns identified in [1]. Other applications include video surveillance where authorities are interested in looking for known activity patterns (e.g., leaving an unattended bag at an airport) in a high throughput observation stream generated somehow (e.g., through transaction logs or video cameras).

In this paper, we address the problem of scalably identifying instances of known activities (i.e., where activity models are known to the application developers such as in the cases listed above) in a high throughput stream of observations. We assume that activities are expressed as temporal stochastic (TS) automata, following the framework of [2] and its predecessor [3]. In particular, [2] took a set of known activity models expressed as temporal stochastic automata and merged them into a single graph and then proposed an algorithm to track activities in observation streams consisting of up to 28.5K observations per second on merged graphs consisting of under 1000 vertices. In this paper, we build upon the work in [2] and scale it up in two directions. First, we are able to look for far more activities than [2] could—our merged automata go to up to 50K vertices. Second, we are able to reliably increase the throughput of observations to 400K observations per second (and in some cases, we can process up to 569K observations per second).

In order to achieve this, we adopt a three-pronged approach. Given a set \( \mathcal{A} \) of activities expressed as temporal stochastic automata, we:

1) First use the results of [2] to generate a merged graph of all the activities in \( \mathcal{A} \).

2) Second, if we have \( (C + 1) \) processors available in a compute cluster supporting an activity detection application, we partition the merged graph into \( C \) subgraphs. One processor is used as a submit node. Splitting the merged graph allows us to scale the number of activities we can process as well as improves our processing time by using a compute cluster. Each component of the merged graph resulting from the split can be processed on the individual compute processors in the cluster. In this paper, we present three ways to partition a merged TS-automaton. The Minimal Overlap Partitioning (MOP) algorithm splits the merged TS automaton by associating a temporal “extent” to each vertex (capturing the time points when a vertex is likely to be active) and then trying to assign sub-automata to different compute nodes that have vertices with similar temporal extents. The Temporal Incidence Partitioning (TIP) method associates an “incidence” measure with any time interval. This incidence measure intuitively measures the number of vertices that are likely to be active.

• A. Pugliese and C. Molinaro are with the DIMES Department, Università della Calabria, Italy. V.S. Subrahmanian and Christopher Thomas are with the Computer Science Department, University of Maryland, USA.
  E-mail: apugliese@dimes.unical.it, topher@umiacs.umd.edu, vs@cs.umd.edu, cmolinaro@dimes.unical.it
within a time interval, and then tries to split the TS automaton into sub-automata by minimizing the standard deviation of these incidence measures. The Occurrence Probability Partitioning (OPP) algorithm transforms the merged TS automaton into a weighted graph where the weights are learned by looking at actual observation (automaton state) streams to understand the true probability that one observation is seen after another observation. OPP then partitions using edge cuts after pruning away edges with very low weights.

3) In the third step, we provide the PASS_Detect algorithm which manages the data structures and the handoffs that occur from one compute node to another so that a high throughput data stream can be searched for activity instances.

The paper is organized as follows. Section 2 describes the TS automaton framework and recapitulates the result from [2] that multiple TS automata can be merged into one. This section does not introduce new results—but recapitulates the part of the [2] framework needed in this paper. Section 3 provides a detailed description of our three partitioning algorithms. Section 4 presents the PASS_Detect algorithm. Section 5 describes our prototype implementation and experimental results. Our results reveal a significant improvement in scalability over the results in [2] showing that PASS can reliably handle over 400K observations per second and merged graphs of up to 50K vertices—in addition, PASS can frequently handle up to 569K observations per second. Finally, Section 6 describes related work and Section 7 outlines conclusions.

2 Temporal Stochastic Automata

Hidden Markov Models and Dynamic Bayesian Networks have been used extensively for representing activities [4], [5], [6], [7]. A slight variant of these methods, stochastic automata, was used to represent activities in [3] and subsequently, a slight extension called Temporal Stochastic Automata was introduced [2] showing that multiple stochastic automata can be merged together to recognize activities. This section does not contain new material—instead it recapitulates definitions first provided in [2]. We use a simple toy example from video surveillance throughout this paper in order to illustrate the definitions.1

Example 2.1: Fig. 1 provides two toy TS automata involving a car picking up/dropping off a person near, say, a building or airport. In these figures, there are several states that can be observed. A motion detection algorithm might detect a person, either followed by a car moving with 60% probability after 1 time unit or a car stopped with 30% probability after 1 time unit and 10% probability after 2 time units. A timespan distribution specifies such transition probabilities, which may vary over time. In the following definition, a time interval is a closed interval of the set $T$ of time points, which in turn can be assumed to be non-negative integers.

Definition 2.1 (Timespan Distribution): A timespan distribution $\omega$ is a pair $(I, \tau)$ where:

- $I$ is a set of disjoint time intervals;
- $\tau: I \rightarrow [0, 1]$ is a function that associates a value $\tau(x, y) \in [0, 1]$ with each time interval $[x, y] \in I$.

We use $\Omega$ to denote the set of all possible timespan distributions. Given a timespan distribution $\omega = (I, \tau)$, we use $S(\omega)$ to denote $\sum_{[x, y] \in I} \tau(x, y)$—we require that $S(\omega) \leq 1$.

A timespan distribution assigns a probability to each member of a set of disjoint time intervals—however, these probabilities can add up to less than 1. The sum $[\ell_1, u_1] \oplus [\ell_2, u_2]$ of two time intervals is defined as $[\ell_1 + \ell_2, u_1 + u_2]$.

We now report the definition of a temporal stochastic (TS) activity, which uses timespan distributions.

Definition 2.2 (Temporal Stochastic (TS) Activity): A temporal stochastic activity (or just activity) is a labeled graph $(V, E, \delta)$ where:

- $V$ is a finite set of observations;
- $E$ is a subset of $(V \times V)$;
- $\exists v \in V$ s.t. $\exists v' \in V$ s.t. $(v', v) \in E$, i.e., there exists at least one start node in $V$—we use start$(A)$ to denote the set of all start nodes of an activity $A$;
- $\exists v \in V$ s.t. $\exists v' \in V$ s.t. $(v, v') \in E$, i.e., there exists at least one end node in $V$—we use end$(A)$ to denote the set of all end nodes of activity $A$;
- $\exists v \in V$ s.t. $\exists (v, v) \in E$, i.e., no self-loops are allowed;
- $\delta: E \rightarrow \Omega$ is a function that associates a timespan distribution with each edge in the graph, such that $\forall v \in V, \sum_{(v' \in V | (v, v') \in E)} S(\delta(v, v')) = 1$.

A TS activity is a graph with at least one start node and one end node. Each edge in the graph is labeled with a timespan distribution. For instance, if an edge $(v, v')$ is labeled with $(I, \tau)$ and $[x, y] \in I$, this means that $v'$ occurs after $v$ with a temporal delay $d \in [x, y]$ with probability $\tau(x, y)$. The last condition requires that if there are multiple edges of the form $(v, -)$, the sum of the probabilities assigned to intervals in these edges must add up to 1.

Definition 2.3 (Activity Instance): Given a temporal stochastic activity $A = (V, E, \delta)$, an instance of $A$ is a sequence $(v_1, \ldots, v_n)$ with $v_i \in V$ such that:

- $v_1$ is a start node in $A$;
- $v_n$ is an end node in $A$;
- $\forall i \in [1, n-1], (v_i, v_{i+1}) \in E$. 

1. To build a complete video surveillance system, the work in this paper would need to be complemented by sophisticated image processing algorithms that are capable of detecting events. As our goal is not to build a video surveillance system, we do not report on image processing algorithms in this paper.
Thus, a temporal activity instance is just a path from any start node to any end node in the TS activity graph. We assume the availability of an observation sequence \( O \) obtained from a set of sources. The observation sequence contains tuples of the form \( f = (ts, obs) \) where \( ts \) is a timestamp and \( obs \) is an observation made at time \( ts \). We use \( f.ts \) and \( f.obs \) to refer to the timestamp and the observation associated with tuple \( f \), respectively. When clear from the context, we interchangeably use \( O = \{ f_1, \ldots, f_n \} \) where:

\[ f_1.ts < f_2.ts < \ldots < f_n.ts; \]
\[ (f_1.obs, \ldots, f_n.obs) \text{ is an instance of } A; \]
\[ \text{prob}(\{f_1, \ldots, f_n\}, A) \geq p. \]

Moreover, we define the span of an occurrence as the time interval \( \text{span}(\{f_1, \ldots, f_n\}) = [f_1.ts, f_n.ts] \).

**Example 2.3:** Consider the observation sequence \( O \) of Example 2.2 with occurrence times 0, 1, 2, 3 respectively. \( O \) is an occurrence of \( A_1 \) with probability 0.1, since \( \text{prob}(O, A_1) = 0.12 > 0.1 \).

Our problem is to find all the occurrences of a set of temporal activities with a probability above a threshold in an observation sequence \( O \). In order to reduce the number of possible occurrences, we apply reasonable restrictions on what constitutes a valid occurrence [2]:

- An occurrence \( O_1 \) of an activity \( A \) is said to satisfy the maximal probability (MP) restriction if and only if there is no other occurrence \( O_2 \) of \( A \) such that \( \text{span}(O_1) = \text{span}(O_2) \) and \( \text{prob}(O_1, A) < \text{prob}(O_2, A) \).
- An occurrence \( \{f_1, \ldots, f_n\} \) of an activity \( A \) is said to satisfy the earliest action (EA) restriction if and only if \( \forall i \in [2, n], \#f \in O \text{ such that } f.ts < f_i.ts \text{ and } \{f_1, \ldots, f_{i-1}, f, f_{i+1}, \ldots, f_n\} \text{ is an occurrence of } A \). That is, EA requires that when looking for the next observation in an activity occurrence, we always choose the first possible successor in the sequence.

Moreover, since short activity occurrences generally tend to have higher probabilities, we normalize occurrence probabilities as defined in [3], by introducing the relative probability \( \text{prob}^* \) of an occurrence \( O \) of activity \( A \) as

\[ \text{prob}^*(O, A) = \frac{\text{prob}(O, A) - p_{\text{min}}}{p_{\text{max}} - p_{\text{min}}} \]

where \( p_{\text{min}}, p_{\text{max}} \) are the lowest and highest possible probabilities of any occurrence of \( A \).

Multi-activity graphs to merge multiple TS stochastic automata together were described in [2] as shown below.

**Definition 2.6 (Temporal Multi-Activity Graph):** Let \( A = \{A_1, \ldots, A_k\} \) be a set of temporal stochastic activities, where \( A_i = (V_i, E_i, \delta_i) \). Let \( ID_A \) be the set of ids of activities in \( A \). The Temporal Multi-Activity Graph (TMAG) for \( A \) is a triple \( G = (V_G, ID_A, \delta_G) \) where:
• $V_G = \cup_{i=1,\ldots,k}V_i$ is a set of observations;
• $\delta_G: V_G \times V_G \times ID_A \rightarrow \Omega$ is a function that maps
  triples $(v, v', id(A_i))$ to the timespan distribution
  $\delta(v, v')$ if $(v, v') \in E_i$, and $\emptyset$ otherwise.

Basically, the TMAG associated with $A$ has the same
vertices as in the graphs being combined. The “global”
probability function $\delta_G$ just returns the probability of
an edge in this graph w.r.t. a specific activity. In particular,
it labels each triple of the form $(v, v', id(A_i))$ with the timespan distribution associated with the
degree $(v, v')$ in activity $A_i$. If no such edge exists in $A_i$,
it associates the empty time span distribution. Fig. 2
shows the merged TS activity when the car dropoff
and car pickup activities are merged. Throughout the
rest of this paper, we will assume that all activities in $A$
have been merged into the TMAG for $A$. Moreover,
we will use $E_G$ to denote $\bigcup_{(v, E, s) \in A} E$, and we will
slightly abuse notation and write $v \in G$ to denote
$v \in V_G$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{Temporal multi-activity graph for the activities of Fig. 1}
\end{figure}

\section{Partitioning TMAGs}

In real world applications, two phenomena occur: (i)
there may be thousands of known normal activities and as a consequence, TMAGs can be quite large,
consisting of tens of thousands of vertices and hun-
dreds of thousands of edges, and (ii) the number of observations made per second is very high, consisting of
hundreds of thousands of observations per second.

In this paper we propose techniques that exploit a
canister of $(C+1)$ compute nodes by partitioning the
set of vertices of a given TMAG $G$ into $C$ components
so that each component can be separately processed by
a different compute node. The additional compute
node is used as a submit node. After building a
partition $P = \{P_1, \ldots, P_C\}$ of $G$, node $N(P_i)$ will thus
handle all tuples $f$ such that $f.obs \in P_i$.

We assume that each compute node includes an
implementation of a sequential activity detection algo-

\subsection{Minimal Overlap Partitioning}

The idea behind minimal overlap partitioning is to try
to partition a TMAG $G$ in such a way that the result-
ing components overlap as little in time as possible.
In order to do this, we first define the temporal extent of
each vertex $v$ in $G$. Intuitively, the temporal extent of
a vertex looks at the time points the observation
corresponding to the vertex can occur once at least
one of the activities that the vertex is involved in has

\begin{definition} (Temporal Extent): Suppose
$G = \langle V_G, ID_A, \delta_G \rangle$ is a TMAG and $v \in V_G$. The
temporal extent of $v$, denoted $te(v)$, is the set of time
points $t \in T$ such that there exist an activity $A \in A$,
an observation $v_0 \in \text{start}(A)$, and a cycle-free path
$v_0, v_1, \ldots, v_n = v$ in $A$ such that for all edges $(v_j, v_{j+1})$ ($0 \leq j < n$) in this path, there is a time interval $I_j$
such that $\delta(v_j, v_{j+1})(I_j) > 0$ and $t \in I_0 \cup I_1 \cup \cdots \cup I_{n-1}$. Recall here that $[t_1, t_2] \cup [t_2, t_3] = [t_1 + t_2, t_1 + t_3]$.

Note that when computing temporal extents, cycles
are not considered.

\begin{example}
Table 1 shows the temporal extents of the vertices of the TMAG in Fig. 2. Consider the observation “Detect Car Stopped”. For $A_1$, the observation can be active anytime within 2 time units after “Detect Car Moving”, which in turn can be active 1 time unit after “Detect Person”. Thus, “Detect Car Stopped” can be active at time points 2 and 3. For $A_2$, it can be active anytime within 5 time units after “Detect Car Moving”, which is a start node itself. Thus, it can be active at time points 1–5. The temporal extent of “Detect Car Stopped” is therefore $[1, 5]$.

The temporal lower (resp. upper) bound of $v$, denoted $tlb(v)$ (resp. $tub(v)$), is the minimum (resp. maximum) value in $te(v)$. The temporal lower and upper bounds of a vertex $v$ in a TMAG represent the earliest and latest time that observation $v$ can occur after commencement of any activity in $A$. For instance, in the
TMAG of Fig. 2, we have $tlb(Detect Car Stopped) = 1$ and $tub(Detect Car Stopped) = 5$.

\end{example}

\begin{observation}
Suppose $G$ is a TMAG. Computing $tlb(v), tub(v)$ for all vertices $v \in V_G$ can be done in time $O(|S_G| \cdot (|E_G| + \log |V_G|))$ where $S_G = \cup_{v \in [1,|A|]} \text{start}(A_i)$.

This result follows immediately from the single-source
shortest path computation algorithm using Fibonacci
heaps [8].

In order to partition a TMAG into $C$ components,
we employ temporal lower bounds and upper bounds
of vertices.

\begin{definition} (Temporal Overlap): Suppose $G$
is a TMAG and $P = \{P_1, \ldots, P_C\}$ is a partition of $G$. Let
$int_i = [\min\{tlb(v) \mid v \in P_i\}, \max\{tub(v) \mid v \in P_i\}]$.

2. Note that if a sequence $S$ of observations is accepted by
the automaton with probability $p$, then there exists a cycle free
subsequence of $S$ that is also accepted by the automaton with
probability $p' \geq p$. This is why we focus on the cycle free case.
The temporal overlap between two distinct intervals \( \text{int}_i, \text{int}_j \), denoted \( \text{overlap}(\text{int}_i, \text{int}_j) \), is defined as:

- 0 if \( \text{int}_i \cap \text{int}_j = \emptyset \);
- \( u - \ell + 1 \) if \( [\ell, u] = \text{int}_i \cap \text{int}_j \).

The temporal overlap of \( P \) is defined as:

\[
\text{overlap}(P) = \frac{1}{2} \sum_{i \neq j} \text{overlap}(\text{int}_i, \text{int}_j).
\]

Intuitively, the temporal overlap between the components in partition \( P = \{P_1, \ldots, P_C\} \) refers to how much time is "shared" between the components.

**Example 3.2:** Consider the TMAG of Fig. 2 and suppose \( P = \{P_1, P_2\} \) with \( P_1 = \{\text{Detect Person}, \text{Detect Car Moving}, \text{Detect Car Stopped}, \text{Detect Door Opening}\} \) and \( P_2 = \{\text{Detect Person Moving}, \text{Detect Door Closing}\} \). Since \( \text{int}_1 = [0, 6] \) and \( \text{int}_2 = [2, 8] \), we have \( \text{overlap}(P) = 5 \).

We would like to find a partition that minimizes temporal overlap so that each compute node \( N(P_i) \) processing component \( P_i \) deals with a different "temporal" part of the TMAG. The notion of a minimal overlap partition below requires that component sizes differ by at most 1.

**Definition 3.3 (Minimal Overlap Partition):** A partition \( P = \{P_1, \ldots, P_C\} \) of a TMAG \( G \) is a minimal overlap partition if:

- for all \( i, j \in [1, C] \), \( (|P_i| - |P_j|)^2 \leq 1 \);
- there exists no other partition \( P' = \{P'_1, \ldots, P'_C\} \) of \( G \) such that for all \( i, j \in [1, C] \), \( (|P'_i| - |P'_j|)^2 \leq 1 \) and \( \text{overlap}(P) > \text{overlap}(P') \).

The problem of computing a Minimal Overlap Partition is basically identical to the optimal construction of 1-dimensional R-tree [9] index structures for spatial databases in the static case, i.e., when the set of objects and their associated intervals are known in advance. The first paper that introduced R-trees [9] only considered the dynamic case, and a simple heuristic was suggested to avoid trying all possible partitions. Many heuristics for the static case have been proposed from then on (see [10] for a survey). The greedy algorithm in Fig. 3 (inexactly) solves MOP by using the interval sorting strategy proposed in [11].

**Observation 3.2:** Algorithm MinimalOverlapPartition terminates and computes a partition in time \( O(|V_G| \cdot \log |V_G|) \).

The above observation follows from the fact that, after sorting \( L_G \), the algorithm scans the list in a single pass in order to assign the vertices in \( V_G \) to the different components of the computed partition.

**Algorithm MinimalOverlapPartition**

**Input:** TMAG \( G \), functions \( \text{tlb} \) and \( \text{tub} \)

**Output:** Partition \( P = \{P_1, \ldots, P_C\} \) of \( G \)

1. for each \( v \in V_G \) do
   2. add \((v, \text{tlb}(v), \text{tub}(v))\) to list \( L_G \)
3. end for
4. sort \( L_G \) by \([\text{tlb}(v) - \text{tub}(v)]/2\) then by \( \text{tub}(v) - \text{tlb}(v) \)
5. size ← \(|V_G|/C|
6. \( L G \leftarrow |V_G| - C \cdot |V_G|/C| \)
7. for each \( i \in [1, C] \) do
   8. \( s \leftarrow \text{size} \)
   9. if \( i \leq L G \) then \( s \leftarrow s + 1 \)
   10. \( L \leftarrow \text{first } s \text{ elements in } L_G \)
   11. \( P \leftarrow \{v \text{ s.t. } (v, \text{tlb}(v), \text{tub}(v)) \in L\} \)
   12. remove \( L \) from \( L_G \)
   13. end for

Fig. 3: Computing a minimal overlap partition

**Example 3.3:** When applied to the TMAG of Fig. 2 with \( C = 2 \), algorithm MinimalOverlapPartition returns the partition \( P = \{P_1, P_2\} \) with \( P_1 = \{\text{Detect Person, Detect Car Moving, Detect Car Stopped}\} \) and \( P_2 = \{\text{Detect Door Opening, Detect Person Moving, Detect Door Closing}\} \).

### 3.2 Temporal Incidence Partitioning

In this section, we try to find a partition \( P = \{P_1, \ldots, P_C\} \) that associates a time interval with each \( P_i \) so that the temporal extent of each vertex in \( P_i \) falls within this interval. Moreover, we would like the time intervals associated with each \( P_i \) to be associated with approximately the same number of vertices (we explain what this means more precisely below) so \( P \) splits the load more or less evenly, and in accordance with time, across the different compute nodes.

**Definition 3.4 (Temporal Incidence):** Suppose \( G = (V_G, ID_A, \delta_G) \) is a TMAG, \( v \in V_G \), and \( t \) is a time point. The incidence of \( v \) on \( t \) is defined as

- \( \text{vi}(v, t) = \frac{1}{\text{te}(v)} \) if \( t \in \text{te}(v) \);
- 0 otherwise.

The incidence of \( v \) on interval \([t', t'']\) is defined as

\[ \text{vi}([t', t''], v) = \sum_{t \in [t', t'']} \text{vi}(t, v). \]

The total incidence on interval \([t', t'']\) is defined as

\[ \text{vi}([t', t'']) = \sum_{v \in V_G} \text{vi}([t', t''], v). \]

We also use \( \text{vi}(t) \) to denote \( \text{vi}([t, t]) \).

Intuitively, given a time point \( t \), the incidence of \( v \) on \( t \) is the percentage of time during the temporal
extent of $v$ that $t$ is “on”. The larger the temporal extent of $v$, the smaller the role $t$ plays. The incidence of $v$ on an interval is the sum of the incidences on the time points in the interval. The incidence of an interval overall is the sum of the incidences of all vertices on that interval—the larger the incidence of an interval is, the more vertices are “involved” during that interval (as their temporal extents intersect the interval) and the greater the load.

Example 3.4: Table 1 shows the temporal incidences of the vertices of the TMAG in Fig. 2. For instance, consider the vertex “Detect Car Stopped” discussed earlier whose temporal extent is $[1, 5]$. The incidence on each of these time points is 1/5. In the same vein, consider the “Detect Person Moving” vertex. Its temporal extent is $[2, 7]$, so its incidence is 1/6 on each of the time points between 2 and 7.

Given a TMAG $G = (V_G, ID_A, \delta_G)$, we define its associated set of “possibly active” time points as $T_G = \bigcup_{v \in V_G} t_e(v)$. For the TMAG in Fig. 2, we simply have $T_G = [0, 7]$.

Observation 3.3: Given a TMAG $G$ and the temporal extent $t_e(v)$ for all $v \in V_G$, computing $vi(v, t)$ for all $(v, t) \in V_G \times T_G$ can be done in time $O(|T_G| \cdot |V_G|)$. The above observation easily follows from the fact that for each $(v, t)$ it suffices to just check whether $t \in t_e(v)$.

The temporal slicing method tries to find a valid partition of the time interval $[\min(T_G), \max(T_G)]$ that is balanced in accordance with the above intuition.

Definition 3.5 (Temporal Slicing): Suppose $G = (V_G, ID_A, \delta_G)$ is a TMAG. A temporal slicing of $T_G$ is a partition $P_T = \{[t_0, t_1], [t_1, t_2], \ldots, [t_{C-1}, t_C]\}$ of $T_G$ such that the standard deviation of $\{vi([t_0, t_1]), vi([t_1, t_2]), \ldots, vi([t_{C-1}, t_C])\}$ is minimized.

By finding a temporal slicing $P_T$ that minimizes standard deviation above, we are trying to split $T_G$ into $C$ intervals, each of which is assigned to a different compute node. We achieve the “balance” intuition mentioned above by minimizing the standard deviation of the incidences on intervals. The dynamic programming algorithm in Fig. 4 finds a temporal slicing.

Theorem 3.1: Algorithm TemporalSlicing correctly computes a temporal slicing of $T_G$ in time $O(C |T_G|^2)$.

Proof: A temporal slicing can be found by solving the optimal partitioning problem, which is defined as follows. An instance of the optimal partitioning problem is a pair $(X, C)$ where $X$ is a finite sequence of $n$ real numbers and $C \leq n$. A $C$-partition of $X$ is a sequence $P = \langle Bin_1, \ldots, Bin_C \rangle$ of $C$ sequences such that $X = Bin_1 \oplus \cdots \oplus Bin_C$, where $\oplus$ is the concatenation operator. The cost of $P$ is $cost(P) = \text{devstd}(|\text{sum}(Bin_1), \ldots, \text{sum}(Bin_C)|)$, where $\text{sum}(Bin_i) = \sum_{x \in Bin_i} x$. The optimal partitioning problem consists in finding a $C$-partition of $X$ with minimum cost—we call it an optimal $C$-partition, or optimal solution.

Given a $C$-partition $P = \langle Bin_1, \ldots, Bin_C \rangle$, we use the following notation:

- $\text{mean}(P) = \text{mean}\{\text{sum}(Bin_1), \ldots, \text{sum}(Bin_C)\}$
- $\text{variance}(P) = \text{variance}\{\text{sum}(Bin_1), \ldots, \text{sum}(Bin_C)\}$
- $\text{devstd}(P) = \text{devstd}\{\text{sum}(Bin_1), \ldots, \text{sum}(Bin_C)\}$

Furthermore, we use $X[i]$ to denote the $i$-th element of $X$, and $X[i]$ to denote the $i$-th prefix of $X$, namely the sequence $\langle X[1], \ldots, X[i] \rangle$, $1 \leq i \leq n$.

To see why solving the optimal partitioning problem allows us to find a temporal slicing, it suffices to observe that $vi([t', t'']) = \sum_{t \in [t', t'']} vi([t, t], v) = \sum_{i \in [t', t'']} vi(t)$.

Before showing the correctness of Algorithm TemporalSlicing, we recall two incremental formulae for computing mean and variance—Algorithm TemporalSlicing is based on them. Let $Y = (y_1, \ldots, y_m)$ be a finite sequence of $m$ real numbers. We use $\mu_i$ (resp. $\sigma_i^2$) to denote the mean (resp. variance) of the first $i$ elements of $Y$, $1 \leq i \leq m$. For each $2 \leq k \leq m$,

$$\mu_k = \mu_{k-1} + \frac{1}{k} (y_k - \mu_{k-1})$$

$$\sigma_k^2 = \left(\frac{k-1}{k}\right) \sigma_{k-1}^2 + \frac{1}{k} (y_k - \mu_{k-1}) (y_k - \mu_{k-1})$$

Fig. 4: Computing a temporal slicing

<table>
<thead>
<tr>
<th>Algorithm TemporalSlicing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Set $T_G$, function $vi$</td>
</tr>
<tr>
<td><strong>Output:</strong> A temporal slicing of $T_G$</td>
</tr>
<tr>
<td>1. $X \leftarrow vi(\min(T_G)), vi(\min(T_G) + 1), \ldots, vi(\max(T_G))$</td>
</tr>
<tr>
<td>2. $n \leftarrow</td>
</tr>
<tr>
<td>3. $Sum \leftarrow n \times n$ matrix</td>
</tr>
<tr>
<td>4. with $Sum[i, j] = X[i]$ for each $i \in [1, n]$</td>
</tr>
<tr>
<td>5. and $Sum[i, j] = Sum[i, j - 1] + X[j]$</td>
</tr>
<tr>
<td>6. for each $i \in [1, n], j \in [i + 1, n]$</td>
</tr>
<tr>
<td>7. $M, V, Sol \leftarrow C \times n$ matrices</td>
</tr>
<tr>
<td>8. with $M[1, j] = Sum[1, j], V[1, j] = 0$,</td>
</tr>
<tr>
<td>9. $Sol[1, j] = 1$ for each $j \in [1, n]$</td>
</tr>
<tr>
<td>10. for each $i \in [2, C], j \in [1, n]$</td>
</tr>
<tr>
<td>11. mine + oldi minm ≔ 0; minsol ≔ 0</td>
</tr>
<tr>
<td>12. for each $k \in [1, j]$</td>
</tr>
<tr>
<td>13. $oldm \leftarrow M[i - 1, k - 1]; newel \leftarrow Sum[k, j]$</td>
</tr>
<tr>
<td>14. $newm \leftarrow oldm + ((newel - oldm)/i)$</td>
</tr>
<tr>
<td>15. $oldv \leftarrow V[i - 1, k - 1]$</td>
</tr>
<tr>
<td>16. $newv \leftarrow (1/k) oldev + (newm - newel)(newel - oldm)$</td>
</tr>
<tr>
<td>17. if $newv &lt; minv$</td>
</tr>
<tr>
<td>18. $minv \leftarrow newv; minm \leftarrow newm; minsol \leftarrow k$</td>
</tr>
<tr>
<td>19. end if</td>
</tr>
<tr>
<td>20. end for</td>
</tr>
<tr>
<td>21. $M[i, j] \leftarrow minm; V[i, j] \leftarrow minv; Sol[i, j] \leftarrow minsol$</td>
</tr>
<tr>
<td>22. end for</td>
</tr>
<tr>
<td>23. $S \leftarrow 0; row \leftarrow C; column \leftarrow n; last \leftarrow n$</td>
</tr>
<tr>
<td>24. while $row &gt; 1$</td>
</tr>
<tr>
<td>25. $S \leftarrow S \cup {Sol[row, column], last}$</td>
</tr>
<tr>
<td>26. $last \leftarrow Sol[row, column] - 1$</td>
</tr>
<tr>
<td>27. $column \leftarrow Sol[row, column] - 1; row \leftarrow row - 1$</td>
</tr>
<tr>
<td>28. end while</td>
</tr>
<tr>
<td>29. $S = S \cup {1, last}$</td>
</tr>
<tr>
<td>30. $P_T = {\ell + \min(T_G - 1, u + \min(T_G) - 1) \mid \ell, u \in S}$</td>
</tr>
<tr>
<td>31. return $P_T$</td>
</tr>
</tbody>
</table>
We start by showing that the optimal partitioning problem exhibits the optimal substructure property, which makes it suitable to be solved with a dynamic programming algorithm. Consider an instance \((X, C)\) of the optimal partitioning problem. Let \(n\) be the number of elements of \(X\). In any \(C\)-partition of \(X\), the last bin must start with an \(X[k]\) where \(k\) is subject to \(C \leq k \leq n\). Thus, we can partition the solution space—i.e., the set of all (optimal and non-optimal) \(C\)-partitions of \(X\)—into \(n\) sets \(S_1, \ldots, S_n, S_{n-C+1}\) depending on where the last bin starts: solutions in \(S_1\) have the last bin starting at \(X[C]\), solutions in \(S_2\) have the last bin starting at \(X[C+1]\), \ldots, solutions in \(S_{n-C+1}\) have the last bin starting at \(X[n]\). Let \(O_i\) be the set of optimal solutions among those in \(S_i\). Obviously, an optimal solution to \((X, C)\) is a solution that is optimal among those in \(O_1, \ldots, O_{n-C+1}\). Consider the solutions in \(S_1\) (a similar reasoning applies to any other \(S_i\)). These are the solutions where the last bin is \(\{X[C], \ldots, X[n]\}\). Let \(P = \{Bin_1, \ldots, Bin_{C-1}, Bin_C\}\) be an optimal solution among those in \(S_1\) (once again, here \(Bin_C = \{X[C], \ldots, X[n]\}\)). We now show that \(P = \{Bin_1, \ldots, Bin_{C-1}\}\) must be an optimal solution to \((X_{C-1}, C-1)\). Reasoning by contradiction, assume that \(P\) is not an optimal solution to \((X_{C-1}, C-1)\). Then, there exists a solution \(P' = \{Bin_1', \ldots, Bin_{C-1}'\}\) to \((X_{C-1}, C-1)\) such that \(cost(P') < cost(P)\), i.e., \(devstd(P') < devstd(P)\). Hence, \(variance(P') < variance(P)\).

Let \(P' = \{Bin_1', \ldots, Bin_{C-1}', Bin_C\}\). Clearly \(P'\) is a solution to \((X, C)\).

Notice that \(cost(P')^2 = variance(\sum(Bin_1), \ldots, \sum(Bin_{C-1}), \sum(Bin_C)) = \frac{\sum(Bin_{C-1})variance(p)}{C} + \frac{\sum(Bin_C)variance(p)}{C}\) and \(cost(P)^2 = variance(\sum(Bin_1), \ldots, \sum(Bin_{C-1}), \sum(Bin_C)) = \frac{\sum(Bin_{C-1})variance(p)}{C} + \frac{\sum(Bin_C)variance(p)}{C}\).

In the equations above we used the definition of \(cost\) and Equation (1).

It is easy to check that \(mean(P) = mean(p')\) and \(mean(P) = mean(P')\). The reason is that it does not matter how a \(C\)-partition divides \(X\), \(mean\) remains the same as long as \(X\) is partitioned into \(C\) bins. In other words, all \(C\)-partitions of a sequence have the same value of mean. Since \(variance(p') < variance(p)\), then \(cost(P')^2 < cost(P)^2\), whence \(cost(P') < cost(P)\), which contradicts the optimality of \(P\). Hence, \(P\) is an optimal solution to \((X_{C-1}, C-1)\).

The dynamic programming algorithm uses three \(C \times n\) dynamic programming tables \(Sol, M, V\). It can be readily verified that Algorithm Temporal Slicing computes the following recursive formulae:

- **Base case** \((i = 1 \land 1 \leq j \leq n)\).
  
  
  \[
  Sol[i, j] = \begin{cases} 
  1 & \text{if } i = 1 \land j = 1 \\
  \sum_{k=1}^{j} X[k] & \text{otherwise}
  \end{cases}
  \]

  \[
  M[i, j] = \sum_{k=1}^{i} X[k]
  \]

  \[
  V[i, j] = 0
  \]

- **Recursive formula** \((1 < i \leq C \land i \leq j \leq n)\). For \(i \leq k \leq j\) we define:
  
  \[
  x(k) = \sum_{l=k}^{i} X[l]
  \]

  \[
  \mu_{old}(k) = M[i-1, k-1]
  \]

  \[
  \mu_{new}(k) = \mu_{old}(k) + \frac{1}{2}(x(k) - \mu_{old}(k))
  \]

  \[
  \sigma_{old}^2(k) = V[i-1, k-1]
  \]

  \[
  \sigma_{new}^2(k) = (i-1)\sigma_{old}^2(k) + \frac{1}{2}(x(k) - \mu_{new}(k))(x(k) - \mu_{old}(k))
  \]

  Let \(k_{opt} = \arg\min_{i \leq k \leq j} \sigma_{new}^2(k)\). Then
  
  \[
  Sol[i, j] = k_{opt}
  \]

  \[
  M[i, j] = \mu_{new}(k_{opt})
  \]

  \[
  V[i, j] = \sigma_{new}^2(k_{opt})
  \]

Thus, to show the correctness of Algorithm Temporal Slicing, it suffices to notice that it leverages the optimal substructure property as follows: it computes the optimal solution to \((X, i)\) by computing the optimal solution under the assumption that the last bin starts at \(X[k]\) for every \(i \leq k \leq j\), and eventually taking the one with minimum cost. The \(O(C|T_G|^2)\) worst-case time complexity of the algorithm easily follows.

**Definition 3.6 (Temporal Incidence Partition):** A temporal incidence partition \(P = \{P_1, \ldots, P_C\}\) of a TAMG \(G\) corresponds to the temporal slicing \(P_T = \{[t_0, t_1], [t_1, t_2], \ldots, [t_{C-1}, t_C]\}\) of \(G\) is a partition where, if \(v \in P_i\), then

- \([t_{i-1}, t_i] \cap te(v) \neq \emptyset\);
- \(\#j \in [1, C] \text{ such that } vi([t_{i-1}, t_i], v) < vi([t_{j-1}, t_j], v)\).

**Observation 3.4:** Given a TAMG \(G\), a temporal slicing \(P_T\) of \(G\), the temporal extent \(te(v)\) for all \(v \in V_G\), and the incidences \(vi([t_i, t_{i+1}], v)\) for all \([t_i, t_{i+1}] \in P_T\) and \(v \in V_G\), computing a temporal incidence partition of \(G\) corresponding to \(P_T\) can be done in time \(O(|V_G| \cdot C)\). The above observation follows from the fact that, for each vertex in \(V_G\), it suffices to look at its incidence on all of the intervals in \(P_T\) to decide which component in \(P\) the vertex has to be assigned to, based on the conditions dictated by the definition of temporal incidence partition.

**Example 3.5:** The temporal incidence partition of the TAMG of Fig. 2 with \(C = 2\) is \(P = \{P_1, P_2\}\) with \(P_1 = \{Detect\ Person, Detect\ Car\ Moving\}\) and \(P_2 = \{Detect\ Car\ Stopped, Detect\ Door\ Opening, Detect\ Person\ Moving, Detect\ Door\ Closing\}\). Observe that, differently from MOP, 2 vertices are assigned to the first component and 4 to the second. In fact, the temporal slicing that better balances vertex incidence is \([[0, 2], [3, 8]]\), and “Detect Car Stopped” has higher incidence on the second interval. Thus, in order to balance the load w.r.t. the whole \(T_G\), the vertex is assigned to the second component, even though this clearly results in a higher overlap(P).

### 3.3 Occurrence Probability Partitioning

Suppose \(occ(v, v', t)\) is the probability that observation \(v'\) occurs within \(t\) units of time after observation \(v\).
These probabilities may be easily learned from some training data and are different from the probabilities in the TMAG.

Definition 3.7 (t-Probability-Weighted Graph): Suppose $A$ is a finite set of activities, $G = (V_G, ID_A, A_G)$ is the TMAG associated with $A$, and $t$ is a time point. The t-probability-weighted graph $PW^t(G) = (V_G, E_G, w_P)$ is the weighted graph where $w_P(v, v') = \text{occ}(v, v', t)$ and $E_G = \bigcup_{(v, E, \delta) \in A} E$. If $\varphi \in [0, 1]$, the $\varphi$-pruning of $PW^t(G)$ eliminates all edges with weight less than $\varphi$.

Definition 3.8 (t-Probability Partition): A t-probability partition of a TMAG $G$ is any partition $P = \{P_1, \ldots, P_C\}$ of $PW^t(G)$ such that $\sum_{v_i \in P_i, v_j \in P_i, i \neq j} w_P(v_i, v_j)$ is minimized.

The idea of an occurrence probability partition is that events that occur “together” often within an observation sequence with high probability are processed by a single compute node.

Observation 3.5: Suppose $G$ is a TMAG and $t$ is a time point. Computing a t-probability partition of $G$ can be done in time $O(|E_G| \cdot \log^3(|V_G|))$.

The above observation follows from the fact that a t-probability partition can be computed through a minimum cut algorithm such as [12].

Example 3.6: Consider the TMAG $G$ of Fig. 2 and the t-probability-weighted graph of Fig. 5. With $C = 2$, $\varphi = 0.12$, and $t = 0$, the t-probability partition of $G$ is $P = \{P_1, P_2\}$ with $P_1 = \{\text{Detect Person}, \text{Detect Car Moving}, \text{Detect Door Opening}, \text{Detect Car Stopped}\}$ and $P_2 = \{\text{Detect Person Moving}, \text{Detect Door Closing}\}$.

![Fig. 5: Example t-probability-weighted graph](image)

4 Parallel Activity Detection

When we have $(C + 1)$ compute nodes available in a cluster for activity detection, our PASS implementation uses one of those compute nodes as a submit node and the other $C$ compute nodes are each used to store the component $P_i$ of a partition $P = \{P_1, \ldots, P_C\}$ of the TMAG $G$ associated with a given set $A$ of activities. Each compute node $N(P_i)$ stores the restriction of $G$ to the vertices in the component $P_i$, denoted $G(P_i)$.

Moreover, each compute node stores information about the set of frontier vertices w.r.t. $P_i$. A frontier vertex w.r.t. $P_i$ is a vertex $v_j \in P_i$ with $i \neq j$ such that there exists a vertex $v_i \in P_i$ such that either $(v_i, v_j)$ or $(v_j, v_i)$ is an edge in the TMAG. When $v_j$ is a frontier node w.r.t. $P_i$, $N(P_i)$ also stores the location of $N(P_j)$. This way, during activity detection, $v_j$ can be transmitted to $N(P_j)$ as well, then a seamless handoff can be done to $N(P_j)$.

The submit node is the entry point to the system where observations are submitted. It maintains a mapping from each vertex $v$ in the TMAG to components in the partition of the TMAG so that every observation can be associated with the compute nodes that will handle it. Each compute node $N(P_i)$ maintains a slight variant of the tMAGIC [2] index corresponding to $G(P_i)$. In particular, it maintains:

- a function $\text{start}_G: V_{G(P_i)} \rightarrow 2^{ID_A}$ that associates with each node $v \in V_{G(P_i)}$ the set of activity id’s for which $v$ is a start node;
- a function $\text{end}_G: V_{G(P_i)} \rightarrow 2^{ID_A}$ that associates with each node $v \in V_{G(P_i)}$ the set of activity id’s for which $v$ is an end node;
- for each $v \in V_{G(P_i)}$, a set $\text{table}_{G(P_i)}(v)$ of records of the form $(\text{current}, \text{activityID}, t_0, \text{prob}, \text{previous})$, where $\text{current}$ and $\text{previous}$ are references to observation tuples, $\text{activityID} \in 2^{ID_A}$ is an activity id, $t_0$ is a timestamp, and $\text{prob} \in [0, 1]$.

The index is used as an association table $T$ that contains tuples that associate an observation with a valid previous observation. Each compute node only has knowledge about the partial graph that it maintains, as well as the location of its frontier vertices on other compute nodes.

The addition and association of observations is formally described in Fig. 6 and the PASS_Detect algorithm is described in Fig. 7. The distributed observation sequence queue $Q = \{Q_1, \ldots, Q_C\}$ holds all observation sequences that are partial instances of an activity. $Q$ is distributed over the nodes and each $Q_i \in Q$ holds observation sequences in which the earliest observation corresponds to a vertex in component $P_i$. In particular, each tuple in $Q_i$ is of the form $(\text{trace}, \text{id}, \text{p})$ where $\text{trace}$ is a set of references to tuples in the observation table (the first reference in the trace of a tuple $a \in Q_i$ is denoted as $a.\text{first}$), $\text{id}$ is an activity id, and $\text{p}$ is a probability. The set of finished activities $F$ contains all observation sequences that are activity instances. The threshold $p_t$ is used to prune low-probability tuples of observations and low-probability observation sequences.

An incoming observation is sent to its corresponding compute node. As soon as the observation is received, the index associates it with any previous observation that it could potentially be the successor of. If the observation corresponds to a frontier vertex, it is sent to multiple nodes. If the observation corresponds to an incoming frontier vertex, it is not associated with a previous observation, but immediately added to the index in order to minimize network communication—an association would require a query to the compute
node on which the vertex is an outgoing frontier vertex. Once an end vertex is observed, the associations are traced back to an observation corresponding to a start vertex.

To see how the algorithms work, let us assume that \( P = \{P_1, P_2\} \) with \( P_1 = \{\text{Detect Person, Detect Car Moving, Detect Car Stopped}\} \) and \( P_2 = \{\text{Detect Person Moving, Detect Door Opening, Detect Door Closing}\} \), and that the observation sequence \( O \) contains the observations reported in Table 2.

### TABLE 2: Example observation sequence

<table>
<thead>
<tr>
<th>( t )</th>
<th>( t.\text{obs} )</th>
<th>( t.\text{ts} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>Detect Person</td>
<td>0</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>Detect Car Moving</td>
<td>1</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>Detect Car Stopped</td>
<td>2</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>Detect Person Moving</td>
<td>3</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>Detect Door Closing</td>
<td>4</td>
</tr>
</tbody>
</table>

When observation \( t_1 \) with \( t_1.\text{obs} = \text{"Detect Person"} \) is entered, it is handled by the "insert" algorithm. At Line 1, \( P_1 \leftarrow P_1. \) Since "Detect Person" is a start vertex, the loop in Lines 3–5 starts and a start tuple is entered into the association table corresponding to "Detect Person" for each activity that starts with "Detect Person". In the example, this is only \( A_1 \), leading to the addition of record \( r_1 \) (Table 3). Since "Detect Person" does not have any preceding nodes, the algorithms ends. The observation \( t_2 \) can correspond to a start vertex of \( A_2 \), producing \( r_2 \), but it can also continue \( A_1 \), producing \( r_3 \). In this case the algorithm loops over all incoming vertices to "Detect Car Moving" at Line 7, which is only "Detect Person", and pulls all observations that have been made of "Detect Person" within the timeframe allowed by \( \delta_G \) (Line 13). Observations \( t_3, t_4 \) and \( t_5 \) are inserted similarly, leading to records \( r_3 \) through \( r_{10} \).

### TABLE 3: Example observation associations

<table>
<thead>
<tr>
<th>( r )</th>
<th>( {t_1, A_1, 0, 1, \bot} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_2 )</td>
<td>( {t_2, A_2, 1, 1, \bot} )</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>( {t_2, A_1, 0, 0, 0, 1} )</td>
</tr>
<tr>
<td>( r_4 )</td>
<td>( {t_3, A_2, 1, 0, 24, t_2} )</td>
</tr>
<tr>
<td>( r_5 )</td>
<td>( {t_4, A_1, 0, 0, 6, t_2} )</td>
</tr>
<tr>
<td>( r_6 )</td>
<td>( {t_5, A_2, 1, 0, 6, t_2} )</td>
</tr>
<tr>
<td>( r_7 )</td>
<td>( {t_6, A_1, 0, 0, 0, 6, t_2} )</td>
</tr>
<tr>
<td>( r_8 )</td>
<td>( {t_7, A_2, 1, 0, 18, t_3} )</td>
</tr>
<tr>
<td>( r_9 )</td>
<td>( {t_8, A_1, 0, 0, 0, 6, t_4} )</td>
</tr>
<tr>
<td>( r_{10} )</td>
<td>( {t_9, A_2, 1, 0, 18, t_4} )</td>
</tr>
</tbody>
</table>

Assume now that \( t_5 \) with \( t_5.\text{obs} = \text{"Detect Door Closing"} \) is entered. Line 1 assigns \( P_1 \leftarrow P_2. \) The only previous vertex is "Detect Person Moving", so \( t_5 \) is associated with \( t_4 \) in Line 15 for both activities \( A_1 \) and \( A_2 \), producing \( r_9 \) and \( r_{10} \). Line 18 identifies "Detect Door Closing" as an end vertex. Line 19 adds the association to queue \( Q_i \).

In parallel, PASS_Detect pulls unfinished activity traces \( a \) from \( Q_i \) and checks if \( T_i \) contains an association \( r \) with \( r.\text{current} = a.\text{first} \) and \( r.\text{id} = a.\text{id} \), i.e., where the second observation of \( r \) matches the first observation in \( a \) and the activities match. If the product of \( a.\text{prob} \) and \( r.\text{prob} \) exceeds \( p_t, r.\text{current} \) is the new head of the intermediate activity trace \( a.\text{trace} \) and the probability of \( a' \) becomes \( a.\text{prob} \cdot r.\text{prob} \) if \( r.\text{current} \) corresponds to a start vertex, \( a' \) is added to the set of finished activity traces. Line 11 distributes the activity traces over the different graph components. If \( P_j \) is different from the current component, the activity traces are sent to the compute node that hosts component \( P_j \).

Given the example observations and the resulting associations, the final traces are \( \{r_9, r_7, r_5, r_3, r_1\} \) for \( A_1 \) with probability 0.06 (we denote the corresponding sequence as \( O_{A_1} \)) and \( \{r_{10}, r_8, r_6, r_2\} \) for \( A_2 \) with probability 0.18 (\( O_{A_2} \)). The latter is thus the most probable occurrence. Moreover, we have \( \text{prob}^*(O_{A_1}, A_1) = 0 \) and \( \text{prob}^*(O_{A_2}, A_2) = 0.2 \).

## 5 Implementation and Experiments

### 5.1 Experimental Setup

We implemented a prototype of the PASS system in Java. The implementation required a good balance between fast network communication and simplicity of use. Hadoop did not appear well suited for the amount of inter-node communication required in PASS. On the other hand, we did not want to use low level socket communication for a prototype application in order to have guaranteed transfer of data. As a compromise, the Java Remote Method Invocation framework was chosen, because it provides sufficiently fast communication while independently handling all communication threads.

Our experiments used a Linux cluster of up to 9 nodes out of which one served as the submit node and either 2, 4 or 8 nodes served as compute nodes when we were assessing the performance of the system by varying the number of compute nodes. All compute nodes had an identical hardware configuration with two Intel Xeon Quad Core 2.3 GHz processors and 8 GB of RAM. Only one process was allocated to each compute node.

We considered different sizes of synthetic TMAGs in different densities. The configurations are reported in Table 4. The algorithm to create synthetic TMAGs first randomly creates a user specified number of vertices and then randomly generates outgoing edges based on a Gaussian distribution. Temporal activity graphs assume a temporal progression from a start node to an end node, that is, all paths through the graph have a temporal ordering. To ensure that in the synthetic creation of the TMAGs all activities have a fairly fine-grained temporal progression, an edge between vertices was only allowed to be created when the events that correspond to the vertices were in relative temporal proximity. Individual timespan
distributions $\delta_G(v, v')$ contain a maximum of 3 time intervals, where intervals are chosen at random based on a user specified minimum and maximum duration between events and a random probability distribution over the intervals with the only restriction that $\sum_{(v, v') \in E} S(\delta_G(v, v')) = 1$. The paths within a TMAG that represent activity instances range from 2 to 21 vertices and are on average of length 7 (though the TMAG itself could have between 10K to 50K vertices).

The creation of synthetic observation sequences follows activity paths in the corresponding activity graphs and adds noise to simulate spurious and unrelated observations. Sequences are generated based on the timespan distributions $\delta_G$ in the TMAG $G$. 

---

**Algorithm insert**

Input: New tuple to be inserted $f_{new}$, tMAGIC index $tm$, probability threshold $p_t$, boolean flag $EA$, indicating whether the $EA$ restriction must be applied, association table $T_G = \{T_1, \ldots, T_C\}$, partition $P = \{P_1, \ldots, P_C\}$ of tMAG $G$

Output: Updated $tm$ and $T_G$

```plaintext
// retrieve TMAG component for $f_{new}$
P_i ← $P_i \in P$ s.t. $f_{new}.obs \in P_i$
// Look at start nodes
if startG($f_{new}.obs$) \neq \{\} then updatedActivities \leftarrow \{\}
   for each id \in startG($f_{new}.obs$) updatedActivities do
      add ($t_{new}.id, f_{new}.ts, 1, \bot$) to $T_i(f_{new}.obs)$
   end for
end if
// Look at intermediate nodes
for each node $v \in V_G$s.t.$\exists id \in ID_A, \delta_G(v, f_{new}.obs, id) \neq \{\}$ do
   for each $r \in T_i(v)$
      if $\delta_G(v, f_{new}.obs, id) \neq \{\} \land (\neg EA \lor (EA \land \exists r' \in T_i(v), s.t. r'.prob > r.prob)$ then
         $p \leftarrow \tau(x, y)$ where $[x, y] \in I$ and $x \leq f_{new}.ts - r.current.ts \leq y$
         if $r.prob \cdot p \geq p_t$ then
            $r_n \leftarrow (t_{new}.id, r.t0, r.prob \cdot p, r.current)$
            add $r_n$ to $T_i(f_{new}.obs)$
         end if
      end if
   end for
end for
```

---

**Algorithm PASS_Detect**

Input: TMAG $G$, min activity probability $p_t$, boolean flag $MP$, indicating whether the $MP$ restriction must be applied

Global variable: List of finished activity instances $F$

Local variables: TMAG component $P$, association table $T_i$, observation sequence queue $Q_i$

```plaintext
if $Q_i = \{\}$ then wait
   for each $a \in Q_i$
      $t \leftarrow a.first$
      for each $(t, id, t0, prob, t_p) \in T_i$
         if $a.prob \cdot prob > p_t$ then
            $a' \leftarrow (t_p + a.trace, id, a.prob \cdot prob)$
            if $MP \land (\exists a_p \in Q_i \cup F, s.t. span(a_p) = span(a') \land a_p.prob > a.prob)$ then discard $a'$
            else
               if(startG($t.obs$id) $\neq \{\}$ then $F \leftarrow F \cup a'$
               else
                  for each $P_j \in t.obs.components \land a_p.obs \in P_j$
                     $Q_j \leftarrow Q_j \cup a'$
                  end for
               end if
            end if
         end if
      end for
   end for
end if
```
After choosing an activity $A$ and one of its start nodes at random for the start of an observation sequence, the subsequent observations are chosen based on the probability distributions in $\delta_G$, that is $\text{prob}((t', v') \in O | (t, v) \in O) = \tau(t, t')$ where $(I, \tau) = \delta_G(v, v', A)$. For instance, if for the example TMAG in Fig. 2, “Detect Person” is chosen as a start node, the first observation created is $(t_0, \text{Detect Person})$. The second observation is either $(t_0 + 1, \text{Detect Car Moving})$ with 60% probability, or $(t_0 + \Delta t, \text{Detect Car Stopped})$, where $\Delta t = 1$ with 30% probability and $\Delta t = 2$ with 10% probability.

The results were obtained for various combinations of TMAGs, partitioning schemes, number of nodes, and MP/EA restrictions. Results were averaged over 10 runs, excluding the highest and lowest values. In order to compare different partitioning algorithms, we set $\varphi$ to 0 and $t$ to the maximum allowed time.

### 5.2 Experimental Results

Figs. 8 and 9 show the performance of the proposed partitioning schemes in terms of time to compute the partitions, when varying TMAG sizes and number of compute nodes. OPP performs best in the majority of cases, with a performance gain that increases with larger TMAGs. On the other hand, it appears to suffer more than MOP and TIP\(^3\) from the density of large TMAGs. All of the partitioning schemes scale well with the size of the input TMAGs, and their performance is almost independent of the number of partition size.

Fig. 10 reports the number of observations per second the system is able to process versus partitioning schemes, for different numbers of compute nodes, with both the sparse and dense 50K TMAG. The results show that the performance of the system is very satisfactory: for instance, under TIP with $C = 8$ the system processes up to 577K observations per second—569K when averaged over TMAG sizes.

Fig. 11 reports the performance of the system when varying the size of the TMAGs for different numbers of compute nodes, with sparse and dense TMAGs. The results confirm that the system scales well with respect to the number of compute nodes used.

Moreover, they show that for TMAG sizes upto 50K vertices, performance is impacted more by the actual topology of the TMAGs than TMAG size.

Fig. 12 shows how performance varies with respect to TMAG sizes for different densities. As expected, we always obtain better average performance with sparser TMAGs, and the topology of the TMAGs impacts more than their size. Finally, Fig. 13 reports the performance obtained when varying the number of compute nodes, for different applied restrictions and TMAG densities. Besides scaling well with respect to the number of compute nodes used, the results show that the density of the TMAGs has a higher impact when using more compute nodes, whereas the performance appears rather stable with respect to the applied restrictions.

In summary, the results of the experiments show that PASS significantly scales past work on tMAGIC [2] which represented the highest processing throughput for activity detection observed to date. While tMAGIC only detects activity instances in TMAGs containing hundreds of vertices, PASS can reliably (as seen from the performance graphs shown

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Edges (sparse)</th>
<th>Edges (dense)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>59,911</td>
<td>99,921</td>
</tr>
<tr>
<td>20,000</td>
<td>120,027</td>
<td>199,938</td>
</tr>
<tr>
<td>30,000</td>
<td>180,080</td>
<td>300,368</td>
</tr>
<tr>
<td>40,000</td>
<td>240,613</td>
<td>399,758</td>
</tr>
<tr>
<td>50,000</td>
<td>300,022</td>
<td>499,129</td>
</tr>
</tbody>
</table>

In our implementation of TIP, we reduced the size of $T_G$, via a preprocessing step where a fixed number of consecutive time points were merged into one point, and their vertex incidences adjusted appropriately. As a consequence, the temporal slicings used in the experimental section were faster to compute but also may return sub-optimal results.

Fig. 8: Partitioning times vs. TMAG size for different partitioning schemes and TMAG densities (sparse-S, dense-D), averaged over number of compute nodes.

Fig. 9: Partitioning times vs. numbers of compute nodes for different partitioning schemes, averaged over TMAG size and density.

3. In our implementation of TIP, we reduced the size of $T_G$, via a preprocessing step where a fixed number of consecutive time points were merged into one point, and their vertex incidences adjusted appropriately. As a consequence, the temporal slicings used in the experimental section were faster to compute but also may return sub-optimal results.
in this section) process TMAGs containing up to 50K vertices and almost 500K edges. Moreover, PASS can reliably handle 400K observations per second (and up to 569K observations in some cases), while tMAGIC can only reliably handle 28,500 observations per second on a single compute node (with a max of about 40K observations in isolated cases). Finally, the three proposed partitioning schemes offer various efficiency/effectiveness tradeoffs. In short, the strategy of partitioning large tMAGs taking into account, expected inter-node communication, proves to pay off well.

6 RELATED WORK

Hidden Markov Models (HMMs) and their variants have been used extensively in the past to model activities. For instance, Duong et al. [5] introduce the Switching Hidden Semi-Markov Model, a two-layered extension of the Hidden Semi-Markov Model (HSMM). The bottom layer represents atomic events and their duration using HSMMs, while the top layer represents high-level activities in terms of atomic events. A survey of temporal concepts and data models used in unsupervised pattern mining from symbolic temporal data is presented in [13]. Automatic learning of transition probabilities in activity models is discussed in [14]. Finally, dynamic Bayesian networks [7] and
Petri nets can also be used for tracking multi-agent activities. A probabilistic extension of Petri nets for activity detection is proposed in [15]. Context free grammars have also been used to define activities [16]. HMMs and their variants are the models more closely related to the temporal stochastic automata used in this paper, but our performance results significantly improve upon past work.

Limitations of traditional database management systems in supporting streaming applications and event processing have prompted extensive research in Data Stream Management Systems (DSMS). An early yet comprehensive survey of relevant issues in data stream management was presented in [17]. Amongst the several systems resulting from research efforts in this direction, of particular relevance is TelegraphCQ [18], a streaming query processor that filters, categorizes, and aggregates flow records according to one or more CQL [19] continuous queries, generating periodic reports. Differently from traditional queries on static data collections, results of continuous queries on streaming data need to be periodically and incrementally updated as new data is received. A significant portion of research in this area has been devoted to optimization of continuous queries [20]. Other works target the recognition of events based on streams of possibly uncertain data [21]. Although the system we propose in this paper operates on streams of observation data, the scope of our work is drastically different from the scope of DSMSs. In fact, we are not interested in retrieving a set of data items satisfying (exactly) certain conditions and keeping this set up to date as new data items are received. Instead, we are interested in finding sets of records such that, with a probability above a given threshold, the records in each set together constitute the “evidence” that a given activity occurred in a specific time interval. Additionally, we want to track partially completed activity occurrences. To the best of our knowledge, DSMSs do not provide support for this type of probabilistic inference.

Moreover, there has been limited work on efficient indexing to support probabilistic activity recognition. The aim of past work on indexing of activities was specialized to very specific activities as opposed to the very general temporal stochastic automaton based approach that we build on top of in this paper. For instance, Ben-Arie et al. [22] use multidimensional index structures to store body pose vectors in video frames. Kerkez [23] develops indexes for case-based plan recognition where knowledge about planning situations enables the recognizer to focus on a subset of the plan library containing relevant past plans. A two-level indexing scheme, along with incremental construction of the plan libraries, is proposed to reduce the retrieval efforts of the recognizer. In short, past work does not address the issue of indexing observations to find activity instances, and these indexing approaches do not account for uncertainty in what defines an activity. Most importantly, none of them is targeted to distributed scenarios when multiple compute machines are available over a network.

7 Conclusions

Recognizing activities in time-stamped observation data poses a major challenge for many applications. As shown in [1], a well known pattern that stock exchanges look for are patterns of sell transactions in which a set of stock brokers successively sell stock in order to induce others to do the same, thus artificially driving the stock price down temporarily, and then buy the stock at a low price in order to subsequently sell at a profit. Large mobile phone companies have sets of patterns they look for in call data records that indicate fraudulent activity. Big box retailers like Amazon’s servers record thousands of transactions (or observations) every second—and they need to process these observations for known patterns of fraudulent activity.

All of these applications have some common features: first, the activities these applications are looking for are many in number, and each can be executed in a multiplicity of ways. Second, the observation streams that come in are high volume observation streams comprising hundreds of thousands of observations per second. They need to be processed quickly.

In this paper, we develop the first parallel search system for activities. Using a well known temporal stochastic automaton model [2], [3], [15], we are able to quickly process temporal stochastic automata that are two orders of magnitude larger than those considered previously in the state of the art tMAGIC [2] system—our PASS system has been shown to experimentally handle temporal stochastic activities containing up to 50K vertices whereas tMAGIC was only able to handle graphs with several hundred vertices. Moreover, PASS can scale the number of observations...
processed per second reliably to about 400K observations per second on 9 compute nodes (with some experimental readings going up to 569K observations per second). In contrast, past work on tMAGIC [2] was able to reliably process 28,500 observations per second (with some experimental readings going up to 40K observations per second). The new ideas and results that PASS brings to the table are two-fold: (i) PASS develops multiple ways to split a tMAG graph—these ideas are based on the temporal annotations within the tMAG in the case of the MOP and TIP approaches, and on occurrence probabilities in the case of OPP. (ii) Once a graph has been partitioned, PASS includes a distributed algorithm and index to process incoming observation streams when looking for activity instances. Of course, PASS heavily uses tMAGIC (with small modifications) within individual compute nodes to compute activity instances, and thus builds on past work.

Acknowledgements. Some authors may have been supported by ARO grants W911NF0910206, W911NF110215, W911NF1110344, an ARO/Penn State MURI award, and ONR grant N000140910685.

REFERENCES


