Control-flow Discovery from Event Streams

Andrea Burattin
University of Padua

Alessandro Sperduti
University of Padua

Wil M. P. van der Aalst
Eindhoven University of Technology

***Abstract*— Process Mining represents an important research field that connects Business Process Modeling and Data Mining. One of the most prominent task of Process Mining is the discovery of a control-flow starting from event logs. This paper focuses on the important problem of control-flow discovery starting from a stream of event data. We propose to adapt Heuristics Miner, one of the most effective control-flow discovery algorithms, to the treatment of streams of event data. Two adaptations, based on Lossy Counting and Lossy Counting with Budget, as well as a sliding window based version of Heuristics Miner, are proposed and experimentally compared against both artificial and real streams. Experimental results show the effectiveness of control-flow discovery algorithms for streams on artificial and real datasets.**

I. Introduction

Process Mining is an emerging research field that comes from two areas: Data Mining on one hand, and Business Process Modeling (BPM) on the other [1]. In typical scenarios, Process Mining algorithms take an event log as input. An event log is a set of observations that refer to executions of a business process. One of the possible outcome of Process Mining algorithms is a control-flow, which is a formal description of the relationships among activities. Several *control-flow discovery* algorithms have been proposed in the past [2]. The first work in the field of Process Mining is reported in a paper by J. Cook and A. Wolf [3], where three approaches (based on recurrent neural networks, transition systems, and Markov chains) were proposed. These approaches, however, do not generate models that are considered useful for business people. R. Agrawal et al. [4] presented an approach that might be considered as the first Process Mining algorithm in the context of BPM. Specifically, they generate a directed graph where nodes represent activities and arcs represent dependencies between activities. Several other works have been produced in the meanwhile [5], [6], [7], generating models represented in different formalisms (i.e., Stochastic Activity Graph, dependency graph, and a custom "block-based" language), whose semantics, however, is not always well-defined. The first approach capable of mining a Petri Net [8] is reported in [9]. A Petri Net is a bipartite graph that represents a concurrent and distributed system. Nodes can either be "transitions" or "places". Transitions, typically, represent activities; nodes represent the states (intermediate or final) that the process can reach. Petri Nets are also characterized by a well defined "dynamic semantic", which allows them to express "high level" control-flow patterns (such as sequence, parallelism, synchronization, exclusive choice, and simple merge) [10]. Several other control-flow discovery algorithms have been subsequently defined and experimentally evaluated. All these control-flow discovery algorithms have been defined for "batch processing", i.e., on a complete event log. Nowadays, however, information systems supporting business processes are able to produce a huge amount of events thus creating new opportunities and challenges from a computational point of view. In fact, in case of streaming data it is impossible to store all events. In addition to that, a business process may evolve over time. Manyika et al. [11] report possible ways for exploiting large amount of data to improve the company business. In their paper, *stream processing* is defined as "*technologies designed to process large real-time streams of event data*" and one of the example applications is *process monitoring*. The challenge to deal with streaming event data is also discussed in the Process Mining Manifesto[1].

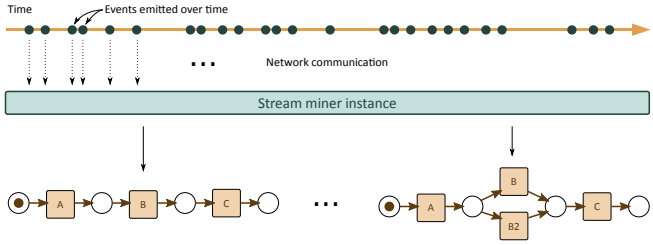

Fig. 1: General idea of SPD: the stream miner continuously receives events and, using the latest observations, updates the process model (in this case, a Petri Net).

This paper proposes and compares some algorithms for the discovery of control-flow models, starting from streaming event data. In the remainder of this paper we refer to this problem as *Streaming Process Discovery* (SPD). The fundamental characteristics of our approach consist in the ability of generating a model with a well-defined semantics[2]. The general representation of the SPD problem that we adopt in this paper is shown in Figure 1: one or more sources emit events (represented as solid dots) which are observed by the stream miner that keeps the representation of the process model up-to-date.

According to [12], [13], [14], a data stream consists of an unbounded sequence of data items with a very high

A. Burattin (*burattin@math.unipd.it*) and A. Sperduti (*sperduti@math.unipd.it*) are with the Department of Mathematics, University of Padua, Italy. Wil M. P. van der Aalst (*w.m.p.v.d.aalst@tue.nl*) is with the Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands

[1]The Process Mining Manifesto is authored by the IEEE CIS Task Force on Process Mining (`http://www.win.tue.nl/ieeetfpm/`).

[2]We also require that the model can easily be converted, preserving its semantics, to a Petri Net.

throughput. In addition to that, the following assumptions are typically made: *i)* data is assumed to have a small and fixed number of attributes; *ii)* mining algorithms should be able to process an infinite amount of data, without exceeding memory limits; *iii)* the amount of memory available to a learning/mining algorithm is considered finite, and typically much smaller than the data observed in a reasonable span of time; *iv)* there is a small upper bound on the time allowed to process an item, e.g. algorithms have to scale linearly with the number of processed items: typically the algorithms work with one pass of the data; *v)* stream "concepts" (i.e. models generating data) are assumed to be stationary or evolving [15]. The task of mining data streams is typically focused on specific types of algorithms [16], [15], [13], [17]. In particular, techniques have been developed for clustering, classification, frequency counting, time series analysis, and changes diagnosis (concept drift detection).

The remainder of this paper is organized as follows: Section II reports on related work; Section III introduces the basic concepts required to analyze the SPD problem; Sections IV to VII describe and discuss the three approaches we propose; Section VIII reports experimental results; Section IX concludes the paper.

II. Related Work

Over the last decade, dozens of process discovery techniques have been proposed [2], e.g., Heuristics Miner [18]. Few works in Process Mining literature touch issues related to mining event data streams. In [19], the authors focus on incremental workflow mining and *task mining* (i.e. the identification of the activities starting from the documents accessed by users). The basic idea of this work is to mine process instances as soon as they are observed; each new model is then merged with the previous one so to refine the global process representation. The approach described is thought to deal with the incremental process refinement based on logs generated from version management systems. However, as authors state, only the initial idea is sketched. Another approach for mining legacy systems is described in [20]. In particular, after the introduction of monitoring statements into the legacy code, an incremental process mining approach is presented. The idea is to apply the same heuristics of the Heuristics Miner into the process instances and add these data into AVL trees (Adelson-Velskii and Landis' trees, a kind of self-balancing binary search trees), which are then used to find the best holding relations. Actually, this technique operates on "log fragments", and not on single events, so it is not really suitable for an online setting. An interesting contribution to the analysis of evolving processes is given in [21]. The proposed approach, based on statistical hypothesis testing, aims at detecting *concept drift*, i.e. changes in event logs, and to identify regions of change in a process. Maggi et al. [22] propose an online technique to extract a declarative process model (using the Declare language). Declare models, however, are sets of constraints over activities (the language is based on Linear Temporal Logic) which cannot be used to model full control-flows.

TABLE I: Example of event log.

#	Case Id	Activity	Timestamp
1	$C1$	A	01-10-2013 00:01
2	$C1$	B	01-10-2013 01:12
3	$C2$	A	01-10-2013 15:23
4	$C2$	B	01-10-2013 15:52
5	$C2$	C	01-10-2013 18:34
6	$C1$	D	01-10-2013 20:45
7	$C2$	D	01-10-2013 22:56

None of the above works, however, is able to address the stream processing requirements and to generate a process model using an imperative language.

III. Basic concepts

In this section we introduce the basic concepts required to comprehend the rest of the paper.

In order to better understand how an event log is composed, it is necessary to clarify that each execution of a business process constitutes a *case*. Each case, therefore, is a process instance that produces a *trace*. Each trace consists of the list of *events* that refer to specific executed activities. The fundamental attributes of an event are the name of the executed activity and the timestamp (that reports the execution time of the given activity). Table I presents a brief example of an event log, which consists of 7 events, divided into two cases ($C1$ and $C2$), and referring to 4 activities (A, B, C, and D). Please note that the data fields that we assume in this log structure, should be considered typically available in real-world event logs (all process-aware information systems need to encode the described entities somehow). More formally, given the set of all possible activity names $\mathcal{A}$, the set of all possible case identifiers $\mathcal{C}$ and the set of timestamps $\mathcal{T}$, it is possible to define:

Definition 1: An *event* e is a triplet $e = (c, a, t) \in \mathcal{C} \times \mathcal{A} \times \mathcal{T}$, and it describes the occurrence of activity a for the case c at time t. The set of all possible events is called *event universe* $\mathcal{E} = \mathcal{C} \times \mathcal{A} \times \mathcal{T}$.

An event log is then defined as a set of events. Starting from an event log, it is useful to isolate the set of observed activities, and the set of observed case ids: given an event $e = (c, a, t)$, we get each field using $\#_{\text{case}}(e) = c$; $\#_{\text{activity}}(e) = a$; and $\#_{\text{time}}(e) = t$.

Definition 2: Given a finite set $\mathbb{N}_n^+ = \{1, 2, \ldots, n\}$ and a "target" set A, we define a sequence σ as a function $\sigma : \mathbb{N}_n^+ \to A$. We say that σ maps indexes to the corresponding elements in A. For simplicity, we refer to a sequence using its "string" interpretation: $\sigma = \langle s_1, \ldots, s_n \rangle$, where $s_i = \sigma(i)$ and $s_i \in A$.

In our context, we use timestamps to sort the events. Therefore, it is safe to consider a trace just as a sequence of activity recordings. For example, assuming again the event log of Table I, we observe these two traces: $C1 = \langle A, B, D \rangle$ and $C2 = \langle A, B, C, D \rangle$.

Definition 3: Given the event universe $\mathcal{E}$, an *event stream* S is a sequence of events $S : \mathbb{N}^+ \to \mathcal{E}$.

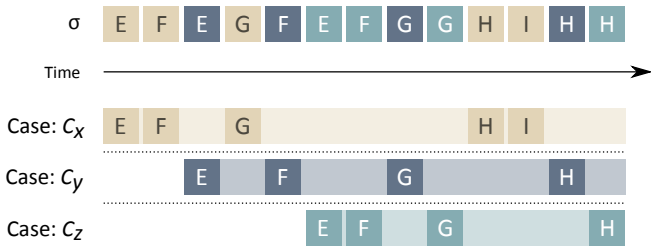

Fig. 2: Graphical representation of an event stream. Boxes represent events: their background colors represent the case id, and the letters inside are the activity names. First line reports the stream, following lines are the single cases.

An event stream is an unbounded sequence of events. We make the assumption that sequence indexes comply with the time order of events (i.e. given the sequence S, for any index $i \in \mathbb{N}^+$, $\#_{\text{time}}(S(i)) \leq \#_{\text{time}}(S(i+1))$). Figure 2 reports a simple representation of an event stream. Please note that two subsequent events of an event stream may belong to different process instances. Our on-line algorithms assume to analyze an event stream one item per time. Specifically, given an event stream S, we will use the notation $e \leftarrow observe(S)$ to indicate that we are going to assign to e the first element not yet taken into account. Therefore, two successive calls of the $observe(S)$ function will return two successive elements of the sequence S.

One of the most used control-flow discovery algorithm is Heuristics Miner [23], [18]. It is composed of a set of heuristics specifically designed to handle real-world data (e.g., to be noise tolerant). The algorithm discovers, via statistical measures and user defined acceptance thresholds, sets of dependencies from the event log and then builds a control-flow model on the basis of these observed causal relations.

A core concept of the algorithm is the *directly-follows* measure. This measure counts the number of times that two activities, a and b, are directly followed each other on an event log (within the same trace). Its value is indicated as $|a > b|$. If we consider again the event log reported in Table I, we have $|A > B| = 2$ (it appears in $C1$ and $C2$), $|B > C| = 1$ (it appears only in $C2$), $|A > D| = 0$ (there are no traces with A directly followed by D), and so on.

Using this frequency measure, Heuristics Miner can calculate the *dependency measure*, a measure of the strength of the causal relation between two activities a and b:

$$a \Rightarrow b = \frac{|a > b| - |b > a|}{|a > b| + |b > a| + 1} \in [-1, 1]. \qquad (1)$$

When the value is close to 1, there is a strong dependency between a and b (b requires a). If the value is close to -1, then there is a strong dependency between b and a (a requires b). Using the dependency measure and a set of user defined thresholds, Heuristics Miner builds a control-flow model which is represented as a *Heuristics Net*, i.e. a directed graph in which nodes represent activities and arcs represent causal dependencies. Moreover, outgoing (ingoing) arcs of a node are annotated as XOR or AND splits (joins) in case they constitutes exclusive choices or parallel alternatives, respectively.

Given a log, the Heuristics Net is constructed as follows. First of all, the dependency measure is computed for all couples of activities occurring in the log. These dependency values are collected into a dependency matrix DM with as many rows and columns as the number of occurring activities. After that, a dependency graph is constructed with a node for each activity. The node corresponding to the activity for which none of the column (row) entries in DM are positive is marked as start (resp. end) node. If the *Use All-Activities-Connected* parameter is set to true (i.e., the default value), the graph is connected by identifying, for each activity a, successors (the activities with the largest value in the row of DM indexed by a), and predecessors (the activities with the largest value in the column of DM indexed by a). Moreover, additional arcs are added between activities a and b for which all the following conditions are satisfied: *i)* $|a > b| \geq \tau_{PO}$, where τ_{PO} is a user defined *positive observations* threshold; *ii)* $a \Rightarrow b \geq \tau_{dep}$, where τ_{dep} is a user defined *dependency* threshold; *iii)* $[(a \Rightarrow a_{max}) - (a \Rightarrow b)] < \tau_{best}$ or $[(b_{max} \Rightarrow b) - (a \Rightarrow b)] < \tau_{best}$, where a_{max} (b_{max}) is any of the strongest (i.e. with highest dependency value) followers (predecessors) of a (b), i.e., $a \Rightarrow a_{max}$ is a maximum of the row of DM indexed by a ($b_{max} \Rightarrow b$ is a maximum of the column of DM indexed by b), and τ_{best} is a user defined *relative-to-best* threshold.

Finally, splits and joins types are identified using the AND-measure. Given a dependency graph with an activity a and two successor nodes b and c, their AND-measure is:

$$a \Rightarrow (b \wedge c) = \frac{|b > c| + |c > b|}{|a > b| + |a > c| + 1} \in [0, 1]. \qquad (2)$$

When the value of the AND-measure is high (i.e. close to 1), it is likely that a is an AND-split (parallelism) and, therefore, that b and c can be executed in parallel. If the value of this measure is close to 0, then b and c are in mutual exclusion and then a is a XOR-split (exclusive choice). Using the AND-measure and the *AND-threshold*, which is a parameter of the algorithm, Heuristics Miner "decorates" all the splits of the dependency graph (either as *AND* or *XOR* split).

Heuristics Miner considers few other heuristics, to handle special cases (such as self loops or long distance dependencies) that we are not going in details here. The dependency graph that Heuristics Miner generates, decorated with the AND/XOR information, can be easily converted into a Petri Net [18].

IV. Baseline Algorithm for Stream Mining (SW)

The simplest way to exploit the Heuristics Miner algorithm with data coming from an event stream is to feed it with the more recently observed events. We call this approach "Heuristics Miner with Sliding Window" (SW).

The basic idea is to: *i)* collect events for a given time span; *ii)* generate a finite event log; *iii)* apply the "standard version" of the algorithm. This idea is described in Alg. 1.

Algorithm 1: Heuristics Miner with SW

Input: S: event stream; M: memory; max_M: maximum memory size

```
1 forever do
2     e ← observe(S)        /* Observe a new event,
      where e = (c_i, a_i, t_i) */
      /* Memory update                                */
3     if size(M) = max_M then
4         shift(M)
5     insert(M, e)
      /* Mining update                                */
6     if perform mining then
7         L ← convert(M) /* Conversion of the
              memory into an event log                */
8         HeuristicsMiner(L)
```

Specifically, an event $e = (c_i, a_i, t_i)$ from the stream S is observed ($e \leftarrow observe(S)$). After that, it is checked whether there is room in memory to accommodate the event. If the memory is full ($size(M) = max_M$) then a *sliding window* policy is adopted [13, Ch. 8], and the oldest event is deleted (*shift*). Subsequently, e is inserted in memory. Periodically (according to user's requirement, line 6) the memory is converted into an event log ($L \leftarrow convert(M)$), and the model is updated by executing the Heuristics Miner algorithm ($HeuristicsMiner(L)$).

The above-mentioned approach has several advantages. The most important one consists in the possibility of mining the log with any process mining algorithm already available in the literature. For example, it is possible to extract information about the social network of people working for the business process. However, the notion of "history" is not very accurate: only the more recent events are considered, and equal importance is assigned to all of them. Moreover, the model is not constantly updated since each new received event triggers only the update of the memory, not necessarily an update of the model: performing a model update for each new event would result in a significant computational burden, well outside the computational limitations assumed for a true online approach. In addition to that, the time required by this approach is completely unbalanced: when a new event arrives, inexpensive operations are performed (*shift* and *insert*); instead, when the model needs to be updated, the log retained in memory is mined from scratch. So, every event is handled at least twice: the first time to store it into a log and subsequently any time the mining phase takes place on it. In online settings, it is more desirable to process each event only once ("one pass algorithm" [24]).

V. Heuristics Miner with Lossy Counting (LC)

The approach presented in this section is an adaptation of an existing technique, used for approximate frequency count. In particular, we adapted the "Lossy Counting" (LC) algorithm described in [25]. The same paper proposes another approach, named Sticky Sampling. However, we preferred Lossy counting since, as authors stated, in practice, it reaches better performances.

The basic idea of Lossy Counting is to conceptually divide the stream into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$, where $\epsilon \in (0, 1)$ represents the maximum approximation error allowed. The *current* bucket (i.e., the bucket of the latest seen element) is identified with $b_{curr} = \lceil \frac{N}{w} \rceil$, where N is the progressive events counter. The basic data structure used by Lossy Counting is a set of entries of the form (e, f, Δ) where: e is an element of the stream; f is the estimated frequency of the item e; and Δ is the maximum number of times e has already occurred. Every time a new element e is observed, the algorithm looks whether the data structure contains an entry for the corresponding element. If such entry exists then its frequency value f is incremented by one, otherwise a new tuple is added: $(e, 1, b_{curr} - 1)$. Every time $N \equiv 0 \mod w$, the algorithm cleans the data structure by removing the entries that satisfy the following inequality: $f + \Delta \leq b_{curr}$. Such inequality ensures that, every time the cleanup procedure is executed, $b_{curr} \leq \epsilon N$.

Our proposal consists in adapting the Lossy Counting algorithm to the SPD problem, by creating an online version of Heuristics Miner. Specifically, since the two fundamental measures of Heuristics Miner (i.e., the dependency measure and the AND-measure) are based on the directly-follows measure (e.g. $|a > b|$), our idea is to "replace" the batch version of this frequency measure, with statistics computed over an event stream. To achieve this goal, we need a couple of instances of the basic Lossy Counting data structure. In particular, we need a first instance to count the frequencies of the activities (we will call it $\mathcal{D}_A$), and another to count the frequencies of the direct succession relations ($\mathcal{D}_R$). However, since the event stream may contain several overlapped traces, a third instance of the same data structure is used to keep track of different cases running at the same time ($\mathcal{D}_C$). In $\mathcal{D}_A$, each item is of the type (a, f, Δ), where $a \in \mathcal{A}$ (set of activity names); f and Δ correspond to the frequency and to the maximum possible error, respectively. In $\mathcal{D}_R$, each item if of the type (r, f, Δ), where $r \in \mathcal{A} \times \mathcal{A}$ (set of possible direct succession relations). Finally, in $\mathcal{D}_C$, each item is of the type (c, a, f, Δ) where $c \in \mathcal{C}$ represents the case identifier and $a \in A$ is the latest activity observed for case c. This last instance of the data structure has to behave slightly differently from the typical Lossy Counting data structures: it does not count the frequencies of the two stored items (c and a), but just the frequency of c (a is just updated with the latest observed value).

The entire procedure is presented in Alg. 2. Specifically, every time a new event $e = (c_i, a_i, t_i)$ is observed (i indicates that we observed the i-th item), $\mathcal{D}_A$ is updated with the new observation of the activity a_i. After that, the procedure checks if there is an entry in $\mathcal{D}_C$ associated to the case id of the current event (c_i). If this is not the case a new entry is added to $\mathcal{D}_C$ (by adding the current case id and the activity observed). Otherwise, if the data structure $\mathcal{D}_C$ already contains an entry for c_i, it means that the new event

Algorithm 2: Heuristics Miner with LC

Input: S event stream; ϵ: approximation error
Initialize the data structure $\mathcal{D}_A$, $\mathcal{D}_C$, $\mathcal{D}_R$
$N \leftarrow 1$
$w \leftarrow \lceil \frac{1}{\epsilon} \rceil$ /* Bucket size */
forever do
  $e \leftarrow observe(S)$ /* Event $e = (c_i, a_i, t_i)$ */
  $b_{curr} = \left\lceil \frac{N}{w} \right\rceil$ /* current bucket id */
  /* Update the $\mathcal{D}_A$ data structure */
  if $\exists (a, f, \Delta) \in \mathcal{D}_A$ *such that* $a = a_i$ **then**
    Remove the entry (a, f, Δ) from $\mathcal{D}_A$
    $\mathcal{D}_A \leftarrow \mathcal{D}_A \cup \{(a, f+1, \Delta)\}$
  else
    $\mathcal{D}_A \leftarrow \mathcal{D}_A \cup \{(a_i, 1, b_{curr} - 1)\}$
  /* Update the $\mathcal{D}_C$ data structure */
  if $\exists (c, a_{last}, f, \Delta) \in \mathcal{D}_C$ *such that* $c = c_i$ **then**
    Remove the entry $(c, a_{\text{last}}, f, \Delta)$ from $\mathcal{D}_C$
    $\mathcal{D}_C \leftarrow \mathcal{D}_C \cup \{(c, a_i, f+1, \Delta)\}$
    /* Update the $\mathcal{D}_R$ data structure */
    Build relation r_i as $a_{\text{last}} \rightarrow a_i$
    if $\exists (r, f, \Delta) \in \mathcal{D}_R$ *such that* $r = r_i$ **then**
      Remove the entry (r, f, Δ) from $\mathcal{D}_R$
      $\mathcal{D}_R \leftarrow \mathcal{D}_R \cup \{(r, f+1, \Delta)\}$
    else
      $\mathcal{D}_R \leftarrow \mathcal{D}_R \cup \{(r_i, 1, b_{curr} - 1)\}$
  else
    $\mathcal{D}_C \leftarrow \mathcal{D}_C \cup \{(c_i, a_i, 1, b_{curr} - 1)\}$
  /* Periodic cleanup */
  if $N = 0 \mod w$ **then**
    foreach $(a, f, \Delta) \in \mathcal{D}_A$ *s.t.* $f + \Delta \leq b_{curr}$ **do**
      Remove (a, f, Δ) from $\mathcal{D}_A$
    foreach $(c, a, f, \Delta) \in \mathcal{D}_C$ *s.t.* $f + \Delta \leq b_{curr}$ **do**
      Remove (c, a, f, Δ) from $\mathcal{D}_C$
    foreach $(r, f, \Delta) \in \mathcal{D}_R$ *s.t.* $f + \Delta \leq b_{curr}$ **do**
      Remove (r, f, Δ) from $\mathcal{D}_R$
  $N \leftarrow N + 1$
  Update the model as described in Section III. For the directly follows relations, use the frequencies in $\mathcal{D}_R$.

is not the first of the given process instances and, therefore, we are observing a directly-follows relation. The algorithm builds the relation r_i and updates its corresponding frequency in $\mathcal{D}_R$. After the possible update of $\mathcal{D}_R$, the periodic cleanup operation might be performed. Please note that when an entry is removed from $\mathcal{D}_A$, the corresponding activity cannot appear in any relations in $\mathcal{D}_R$. This happens because, according to our approach, relations always have lower frequency than activities, and therefore they are removed sooner. The model generation follows exactly the same procedure of Heuristics Miner: the dependency graph is built using the procedure described in Section III. The only difference with Heuristics Miner, in this case, is that given two activities a and b, the directly-follows frequency $|a > b|$ must be retrieved from the $\mathcal{D}_R$ data structure. In Section VII we discuss how to dynamically update the model.

VI. Heuristics Miner with Lossy Counting with Budget (LCB)

The main drawback of the approach based on Lossy Counting, presented in the previous section, is that the only parameter of Lossy Counting is the maximum allowed error ϵ in the frequency approximation and no information on space usage can be provided. In a recent work by Da San Martino et al. [26], a "Lossy Counting with Budget" (LCB) is proposed.

The underlying rationale of Lossy Counting with Budget is to "adapt" the approximation error according to the stream and the available *budget* (i.e. the maximum memory that the algorithm can use). As stated in the previous section, Lossy Counting divides the stream in virtual buckets, each of them with size $w = \lceil \frac{1}{\epsilon} \rceil$. Lossy Counting with Budget, instead, divides the stream in buckets of variable sizes. In particular, each bucket B_i will contain all occurrences that can be accommodated into the available budget. The approximation error, for a given bucket B_i, depends on the bucket size and is defined as $\epsilon_i = \frac{1}{|B_i|}$. Buckets of variable sizes imply that the cleanup operations do not occur always at the same frequency, but only when the current bucket cannot store the new observed item. The cleanup operation iterates the data structure looking for items to be removed. The remove condition is the same as in Lossy Counting: given an item (a, f, Δ) the deletion condition is $f + \Delta \leq b_{curr}$. The current bucket b_{curr}, however, is updated only when no more items can be stored. Moreover, the deletion condition may not be satisfied by any item, thus not creating space for the new observation. In this case, the algorithm has to increase the b_{curr} arbitrarily, until at least one item is removed.

The basic structure of the procedure is exactly the same of Lossy Counting, reported in Alg. 2. There are, however, two main differences: one related to the cleanup operations, and another related to the budget. Concerning the first point, the cleanup operations must be performed before inserting an item (not after, as in Lossy Counting) and the cleanup has to update the current bucket accordingly (several updates might be required, if no elements are removed during the first attempt). Concerning the budget, since we have three different data structures, we need to let them share the memory. Just splitting the available budget, one third per data structure, may lead to a too raw partitioning. Therefore we let the three data structures to grow independently of each other. Once the sum of their sizes has reached the budget size, the cleanup on the shared memory is performed. This implies that, for example, a new activity may kick out an old relation (in this case, the size of $\mathcal{D}_A$ increases, the size of $\mathcal{D}_R$ decreases), thus allowing a dynamic adaptation of the three partitions.

VII. Dynamic Model Update

In this section, we discuss how to update the dependency graph dynamically, as soon as a new event is processed. The synopsis (i.e. the data structures) maintained by the proposed LC-based algorithms (LC and LCB) contains counts. After any update of these synopses (an update occurs when a new event is observed and when the cleanup operations are performed) we would like to update the model with the minimum computational effort, just by looking at the update data structures. In order to do that, we observe that the conditions over dependency measures can actually be rewritten as conditions on the available counts: given two

activities a and b, the condition involving the dependency threshold τ_{dep} can be restated as

$$|a > b| \geq \frac{(1+\tau_{dep})|b > a| + \tau_{dep}}{1-\tau_{dep}} \quad \text{or}$$
$$|b > a| \leq \frac{(1-\tau_{dep})|a > b| - \tau_{dep}}{1+\tau_{dep}}$$

according to whether the last updated count has been $|a > b|$ or $|b > a|$.

A bit more involved is the treatment of the *relative-to-best* threshold, since a modification of the maximum row (column) value may lead to a modification of the dependency graph involving several arcs. First of all, let $best_1(a) = \max_{x \in A}(a \Rightarrow x)$ and $best_2(b) = \max_{x \in A}(x \Rightarrow b)$. The condition $[(a \Rightarrow a_{max}) - (a \Rightarrow b)] < \tau_{best}$ can be rewritten as:

$$|a > b| > \frac{(\tau_{best} - best_1(a) - 1)|b > a| - best_1(a) + \tau_{best}}{best_1(a) - 1 - \tau_{best}},$$

while condition $[(b_{max} \Rightarrow b) - (a \Rightarrow b)] < \tau_{best}$ can be rewritten as:

$$|a > b| > \frac{(\tau_{best} - best_2(b) - 1)|b > a| - best_2(b) + \tau_{best}}{best_2(b) - 1 - \tau_{best}}.$$

Similar disequalities can be derived for $|b > a|$. In addition, we observe that the dependency measure is monotonic with respect to the counts: w.r.t. $|a > b|$, its derivative is $\frac{\partial(a \Rightarrow b)}{\partial |a>b|} = \frac{2|b>a|+1}{(|a>b|+|b>a|+1)^2} > 0$ (monotonically increasing), and w.r.t. $|b>a|$, its derivative is $\frac{\partial(a \Rightarrow b)}{\partial |b>a|} = \frac{-2|a>b|-1}{(|a>b|+|b>a|+1)^2} < 0$ (monotonically decreasing). Thus, given a new event involving activity a, it is not difficult to identify the arcs entering and exiting node a that need to be added or deleted. Therefore, it is sufficient to keep sorted lists of "counts" where a is, in turn, the first or the second element of the relation, together with pointers to the first count not satisfying conditions for inserting the corresponding arc. Because of monotonicity of the dependency measure, a local search starting from these pointed counts allows an efficient update of the model. A similar approach is adopted to distinguish the "type" of splits (either AND or XOR). Given three activities a, b, and c (such that b and c are exiting from a), and the AND threshold τ_{and} it is possible to derive conditions similar to the previous ones:

$$|b > c| \leq \tau_{and}(|a > b| + |a > c| + 1) - |c > b|$$
$$|c > b| \leq \tau_{and}(|a > b| + |a > c| + 1) - |b > c|$$
$$|a > b| \leq \frac{|b > c| + |c > b|}{\tau_{and}} - |a > c| - 1$$
$$|a > c| \leq \frac{|b > c| + |c > b|}{\tau_{and}} - |a > b| - 1$$

In this case, as long as all these inequalities are satisfied, the type of the split is AND, once at least one of these are not satisfied, the split type becomes XOR.

VIII. Experimental Evaluation

We performed evaluations on both artificial and real datasets, with different parameter configurations.

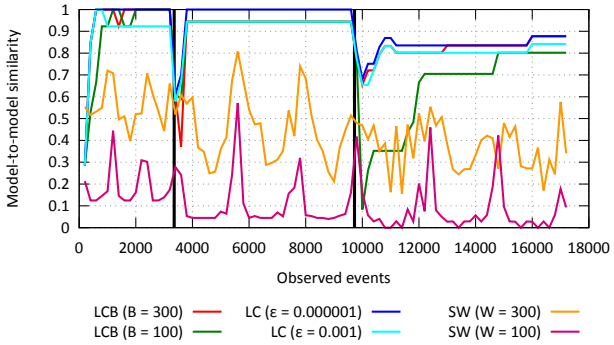

Fig. 3: Model-to-model metric for the approaches presented in this work, with different configurations. Vertical black lines indicate concept drifts.

A. Evaluation on Artificial Models

We evaluated our approach on an artificial dataset, in order to be able to compare the mined model with the one that originated the event stream. The comparison between these two models is performed using the model-to-model metric reported in [27]. This metric performs a "behavioral comparison" between the set of "mandatory behavior" and the set "forbidden behavior" (combined using the Jaccard similarity) of the two process models.

We generated three artificial processes, and corresponding execution logs, using PLG [28]. Then, we combined the three logs in a single stream, in order to simulate two concept drifts. In total, this stream contains 17265 events, referring to 16 activities (the most complex model has 3 splits: 1 AND and 2 XOR). We compared our three approaches using different configurations: for LC we tested $\epsilon = 0.000001$ and $\epsilon = 0.001$; for LCB we tested $B = 300$ and $B = 100$; and for SW we tested $W = 300$ and $W = 100$. Fig. 3 reports the values of the model-to-model metric (computed every 200 events). SW is almost always the worst approach. The other two approaches are basically equivalent and both of them (with all configurations) are able to detect the drifts and correct the model. Fig. 4 reports the space requirements of the approaches (computed every 200 events). As we expected, LC is very eager in terms of space requirements. SW requires constant space. LCB is the most parsimonious. Comparing the results of LC and LCB, both in terms of model-to-model similarity and space requirements, LCB clearly outperforms LC. Fig. 5 reports the average time required to process each event (computed every 200 events). LC and LCB require, basically, half of the time that SW requires. Fig. 6 reports the memory partitioning, for LCB, over its three data-structures $\mathcal{D}_A$, $\mathcal{D}_R$ and $\mathcal{D}_C$. As we expect, the space required by $\mathcal{D}_A$ and $\mathcal{D}_R$ is almost constant (except when drifts occur, in this case, new events and relations are observed), while most of the space is reserved for cases, which are many more in number and covering a relatively small span of time. This is the reason that makes the cleanup procedure prefers the deletion of old cases.

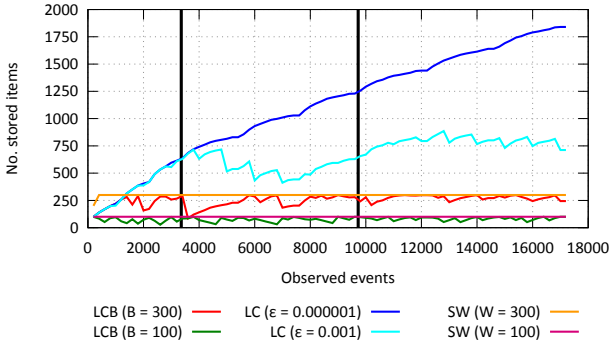

Fig. 4: Space required by the approaches with different configurations. Vertical black lines indicate concept drifts.

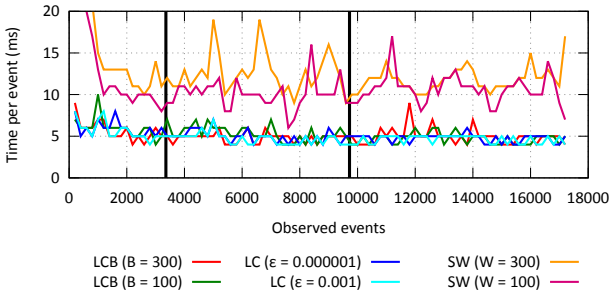

Fig. 5: Processing time, per event, required by the approaches. Vertical black lines indicate concept drifts.

B. Evaluation on Real Dataset

We performed some evaluations also against a real dataset. This dataset is called BPIC-12, and it has been used for the Business Processing Intelligence Challenge 2012[3]. The dataset we used is publicly available[4] and comes from a Dutch Financial Institute. This stream contains 262198 events, distributed among 13087 process instances. The process originating the events consists in the application process for a personal loan or overdraft within the financing organization. Since a formalization of the original process model is not available, it is impossible to compare the mined models with the target one (as we did for the artificial case). In order to compute the quality of the results, we considered the adaptations to Process Mining of *fitness* [29] and *precision* [30] measures. The first considers the ability, for the model, to replay the log. The latter indicates whether the model does not allow for "too much" behavior. We compute these two measures against finite logs generated using the latest n observed events. Since we do not know the original process, and since we are not able to discuss it with the data producer, we are not going to dive into details on the quality of the mined processes.

Fig. 7 reports the average values of fitness and precision of the three approaches, with different configurations and

[3]More information are reported at `http://www.win.tue.nl/bpi2012/doku.php?id=challenge`.

[4]The dataset can be downloaded from `http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f`.

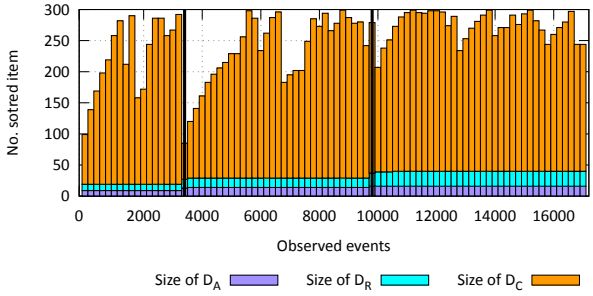

Fig. 6: Partitioning of the three Lossy Counting with Budget data structures: $\mathcal{D}_A$, $\mathcal{D}_R$ and $\mathcal{D}_C$ with $B = 300$.

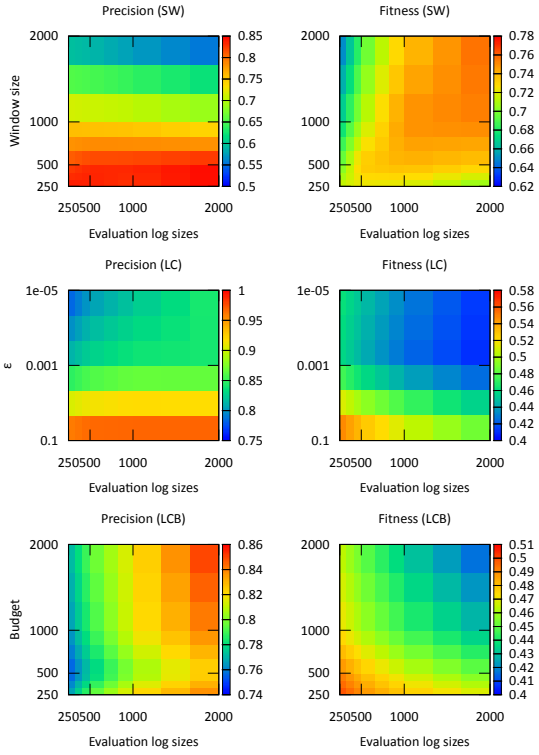

Fig. 7: Average precision and fitness of SW (top), LC (middle), and LCB (bottom), on the BPIC-12 dataset, with different window sizes/budget and evaluation log sizes.

against logs generated with a different number of events. Analyzing the two graphs referring to LCB (bottom), it is interesting to note, as we expected, that the two measures are capturing orthogonal quality perspectives: we can use these graphs to identify the best parameters configuration (to maximize both qualities and avoiding over- and underfitting). Concerning LCB, as the available budget and the evaluation log size increase, the precision increases proportionally, but the fitness decreases. The fitness of LC (middle) behaves similarly to LCB, but the model precision values, when $\epsilon = 0.1$, instead, are so high because of ϵ which allows the approach to store too few elements and so to be extremely small. It is also interesting to note that the fitness measure of SW (top) tends to be higher when the window sizes and the

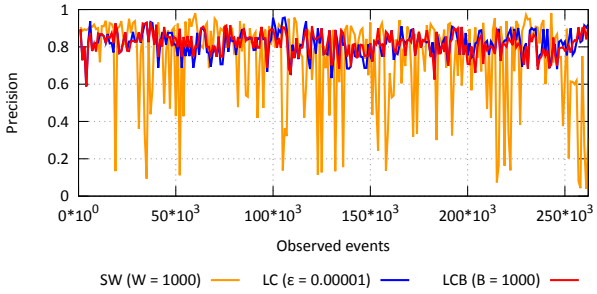

Fig. 8: Comparison of the precision of the models generated by SW, LC and LCB on BPIC-12.

evaluation log sizes are "synchronized". Instead, its precision is extremely unstable, as shown in Fig. 8. This phenomenon indicates that the extracted model, although it is capable of replaying the log (high fitness), it is not very precise (i.e., it is too general).

To analyze the BPIC-12 dataset, the three approaches required (per event): *i)* SW: 24.59 ms; *ii)* LC: 5.68 ms; and *iii)* LCB: 2.56 ms. These time performance demonstrate the applicability of the approaches in real-world business scenarios.

IX. Conclusions

In this paper, we addressed the problem of discovering process models from streams of event data. As baseline approach we have considered a sliding window-based approach, where the standard batch-based Heuristics Miner algorithm is applied to logs obtained using a moving window. Two adaptations of Lossy Counting and Lossy Counting with Budget[5] are proposed and experimentally evaluated. These algorithms extend the ideas of Heuristics Miner in the context of stream. Discussions on time/space requirements and online model generation are reported. The approaches we proposed show important improvements, w.r.t. the baseline, in terms of quality of the mined models, execution time, and space requirements as well. As future work, we plan to improve the process analyst "experience" by extending the current approach to mine different perspectives (such as data or time) and to generate animations that point out evolution points of the process.

Acknowledgment

The work by A. Burattin and A. Sperduti is supported by the Eurostars-Eureka project PROMPT (E!6696). Authors are also grateful to Francesca Rossi and Fabio Aiolli for their advice.

References

[1] W. M. P. van der Aalst, "Process Discovery: Capturing the Invisible," *IEEE Computational Intelligence Magazine*, vol. 5, no. 1, pp. 28–41, 2010.

[2] ——, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[3] J. E. Cook, "Process discovery and validation through event-data analysis," PhD Thesis, University of Colorado, 1996.

[5]The alternative use of *space sketching* algorithms would lead to much frequent cleanup operations.

[4] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining Process Models from Workflow Logs," in *Proc. of EDBT*, Jan. 1998, pp. 469–483.

[5] J. Herbst, "A Machine Learning Approach to Workflow Management," in *Proc. of ECML*, vol. 1810, 2000, pp. 183–194.

[6] S.-Y. Hwang and W.-S. Yang, "On the discovery of process models from their instances," *Decision Support Systems*, vol. 34, no. 1, pp. 41–57, 2002.

[7] G. Schimm, "Process Miner – A Tool for Mining Process Schemes from Event-Based Data," in *Proc. of JELIA*. Springer, 2002, pp. 525–528.

[8] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[9] W. M. P. van der Aalst and B. van Dongen, "Discovering Workflow Performance Models from Timed Logs," in *Proc. of EDCIS*. Springer, 2002, pp. 45–63.

[10] N. Russell, A. H. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar, "Workflow Control-flow Patterns: A Revised View," *BPM Center Report BPM-06-22, BPMcenter. org*, 2006.

[11] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Hung Byers, "Big Data: The Next Frontier for Innovation, Competition, and Productivity," McKinsey Global Institute, Tech. Rep. June, 2011.

[12] L. Golab and M. T. Özsu, "Issues in Data Stream Management," *ACM SIGMOD Record*, vol. 32, no. 2, pp. 5–14, Jun. 2003.

[13] C. Aggarwal, *Data Streams: Models and Algorithms*, ser. Advances in Database Systems. Boston, MA: Springer US, 2007, vol. 31.

[14] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive Online Analysis Learning Examples," *The Journal of Machine Learning Research*, vol. 11, pp. 1601–1604, 2010.

[15] G. Widmer and M. Kubat, "Learning in the Presence of Concept Drift and Hidden Contexts," *Machine Learning*, vol. 23, no. 1, pp. 69–101, 1996.

[16] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining Data Streams: a Review," *ACM Sigmod Record*, vol. 34, no. 2, pp. 18–26, Jun. 2005.

[17] J. Gama, *Knowledge Discovery from Data Streams*. Chapman and Hall/CRC, May 2010.

[18] W. M. P. van der Aalst and T. A. J. M. M. Weijters, "Rediscovering Workflow Models from Event-based Data Using Little Thumb," *Integrated Computer-Aided Eng.*, vol. 10, no. 2, pp. 151–162, 2003.

[19] E. Kindler, V. Rubin, and W. Schäfer, "Incremental Workflow Mining for Process Flexibility," in *Proc. of BPMDS*, 2006, pp. 178–187.

[20] A. C. Kalsing, G. S. do Nascimento, C. Iochpe, and L. H. Thom, "An Incremental Process Mining Approach to Extract Knowledge from Legacy Systems," in *Proc. of EDOC*, Oct. 2010, pp. 79–88.

[21] R. P. J. C. Bose, W. M. P. van der Aalst, I. Žliobaitė, and M. Pechenizkiy, "Handling Concept Drift in Process Mining," in *Proc. of CAiSE*. Springer, 2011, pp. 391–405.

[22] F. M. Maggi, A. Burattin, M. Cimitile, and A. Sperduti, "Online Process Discovery to Detect Concept Drifts in LTL-Based Declarative Process Models," in *Proc. of CoopIS*. Springer, 2013, pp. 94–111.

[23] W. M. P. van der Aalst, T. A. J. M. M. Weijters, and A. K. A. de Medeiros, "Process Mining with the Heuristics Miner-algorithm," BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven, 2006.

[24] N. Schweikardt, "Short-Entry on One-Pass Algorithms," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer, 2009, pp. 1948–1949.

[25] G. S. Manku and R. Motwani, "Approximate Frequency Counts over Data Streams," in *Proc. of VLDB*. Hong Kong, China: Morgan Kaufmann, 2002, pp. 346–357.

[26] G. Da San Martino, N. Navarin, and A. Sperduti, "A Lossy Counting Based Approach for Learning on Streams of Graphs on a Budget," in *Proc. of IJCAI*. AAAI Press, 2012, pp. 1294–1301.

[27] F. Aiolli, A. Burattin, and A. Sperduti, "A Business Process Metric Based on the Alpha Algorithm Relations," in *Proc. of BPI*. Springer, 2011.

[28] A. Burattin and A. Sperduti, "PLG: a Framework for the Generation of Business Process Models and their Execution Logs," in *Proc. of BPI*. Springer, 2010, pp. 214–219.

[29] A. Adriansyah, B. van Dongen, and W. M. P. van der Aalst, "Conformance Checking Using Cost-Based Fitness Analysis," in *Proc. of EDOC*. IEEE, Aug. 2011, pp. 55–64.

[30] J. Muñoz Gama and J. Carmona, "A fresh look at Precision in Process Conformance," in *Proc. of BPM*. Springer, 2010, pp. 211–226.