Mining Frequent Infix Patterns from Concurrency-Aware Process Execution Variants

Michael Martini
RWTH Aachen University
Aachen, Germany
michael.martini1@rwth-aachen.de

Daniel Schuster
Fraunhofer FIT
Sankt Augustin, Germany
RWTH Aachen University
Aachen, Germany
daniel.schuster@fit.fraunhofer.de

Wil M.P. van der Aalst
Fraunhofer FIT
Sankt Augustin, Germany
RWTH Aachen University
Aachen, Germany
wvdaalst@pads.rwth-aachen.de

ABSTRACT

Event logs, as considered in process mining, document a large number of individual process executions. Moreover, each process execution consists of various executed activities. To cope with the vast amount of process executions in event logs, the concept of variants exists that group process executions with identical ordering relations among their executed activities. Variants are an integral concept of process mining and help process analysts explore, filter, and manage large amounts of event data. In this paper, we consider concurrency-aware variants that allow activities within a process execution to be partially ordered—the execution of individual activities can overlap in time. However, the number of variants is often vast, making it challenging for process analysts to explore event data. Therefore, we present a novel approach to frequent pattern mining from concurrency-aware variants. We show that mining frequent patterns from concurrency-aware variants can be reduced to the frequent subtree mining problem. Further, we compare our proposed algorithm to a state-of-the-art frequent subtree mining algorithm exhibiting improved performance on real-life event logs.

PVLDB Reference Format:
Michael Martini, Daniel Schuster, and Wil M.P. van der Aalst. Mining Frequent Infix Patterns from Concurrency-Aware Process Execution Variants. PVLDB, 16(10): 2666 - 2678, 2023.
doi:10.14778/3603581.3603603

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fit-daniel-schuster/Mining-Frequent-Infix-Patterns-from-Concurrency-Aware-Process-Execution-Variants.

1 INTRODUCTION

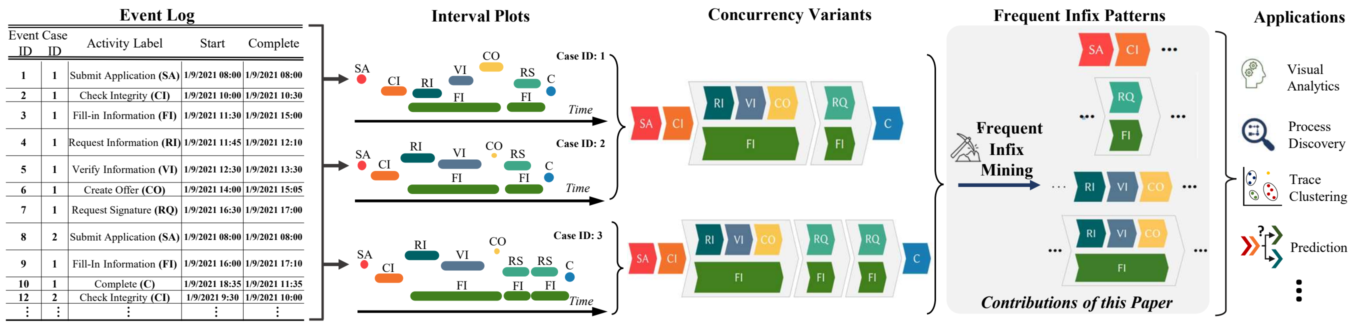
Process mining [31] is an established research field offering a wide range of data-driven techniques that have changed how organizations analyze, manage and improve their processes, from administrative to production processes [22]. A vital artifact for these techniques is *event data*, generated during the execution of processes and stored in information systems that support those process executions. In short, process mining aims to analyze and improve existing processes by extracting insights from event data.

Event data contain *activity instances* representing executions of activities; for instance, the packing of a customer's order within an order-to-cash process. The activity instances associated with an individual process execution form a *trace*. Real-world event logs usually contain many traces, and as the logging increases and becomes more fine-grained, the number increases even further [1, 13]. To cope with large event logs and a high number of traces, *variants* are used. Variants group traces of an event log that have identical ordering relationships between the activities they contain. Thus, variants are an integral concept in process mining to handle a large number of traces. However, the number of variants can still be large due to the distribution of traces per variant often following power-law distributions [32]. Consequently, as exploring all variants is infeasible, one can instead rely on the frequent subpatterns shared by variants that emerge from consistent subprocesses making up larger business processes. The frequent subpatterns of variants this paper focuses on are called frequent *infixes*. The goal is mining all frequent infixes of a set of concurrency-aware variants, henceforth shortened to concurrency variants.

Most existing process mining techniques consider traces and variants as strict totally-ordered sequences of activities [31]. However, in real-world processes, activities overlap in time, i.e., occur concurrent [19]. Consequently, concurrency variants [24] have been introduced to more adequately represent the concurrency of activities. Consider Figure 1 that shows an event log from a loan application process; each row represents an activity instance. Activity instances are grouped into traces according to their case id. Looking at a trace's activity instances represented in an interval plot, one can observe sequentiality, e.g., *submit application* (SA) is followed by *check integrity* (CI) in case 1, and concurrency, e.g., *create offer* (CO) temporarily overlaps *fill-in information* (FI) in case 1. Based on the sequentiality and concurrency of activity instances, concurrency variants are derived [24]. Traces having the same sequential and concurrent relations between instances of the same activities are then grouped into a concurrency variant, e.g., case 1 and 2 in Figure 1. Concurrency variants consist of colored chevrons representing activities. Horizontally aligned chevrons indicate sequential execution of activities, while vertically aligned chevrons indicate concurrent execution.

A benefit of traditional variants is that the strict total ordering of activities allows for established techniques, i.e., frequent sequence mining [2], to mine frequent infixes. However, as concurrency variants model concurrency between activities and sequences, e.g.,

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603603



Event ID	Case ID	Activity Label	Start	Complete
1	1	Submit Application **(SA)**	1/9/2021 08:00	1/9/2021 08:00
2	1	Check Integrity **(CI)**	1/9/2021 10:00	1/9/2021 10:30
3	1	Fill-in Information **(FI)**	1/9/2021 11:30	1/9/2021 15:00
4	1	Request Information **(RI)**	1/9/2021 11:45	1/9/2021 12:10
5	1	Verify Information **(VI)**	1/9/2021 12:30	1/9/2021 13:30
6	1	Create Offer **(CO)**	1/9/2021 14:00	1/9/2021 15:05
7	1	Request Signature **(RQ)**	1/9/2021 16:30	1/9/2021 17:00
8	2	Submit Application **(SA)**	1/9/2021 08:00	1/9/2021 08:00
9	1	Fill-In Information **(FI)**	1/9/2021 16:00	1/9/2021 17:10
10	1	Complete **(C)**	1/9/2021 18:35	1/9/2021 11:35
12	2	Check Integrity **(CI)**	1/9/2021 9:30	1/9/2021 10:00
⋮	⋮	⋮	⋮	⋮

Figure 1: Overview of mining frequent infix patterns from concurrency variants and related concepts from process mining.

fill-in information (FI) being concurrent to *request* (RI) and *verify information* (VI) in Figure 1, frequent sequence mining cannot be used on concurrency variants. Consequently, common techniques in process mining that build on top of frequent infixes, such as trace clustering [7, 39]; next-event prediction [6, 9, 16], and (local) process discovery [20, 30] can no longer be used. Thus, to allow such application for concurrency variants, this paper addresses the following research questions to allow for the mining of frequent infixes.

RQ1 How can concurrency variants and their infixes be modeled to allow for mining frequent infix patterns?

RQ2 How can the mining of infix patterns be done efficiently?

The contributions of this paper are as follows. We answer **RQ1** by showing that frequent infix mining from concurrency variants can be reduced to the task of frequent subtree mining. We model concurrency variants as labeled, rooted, ordered trees and infixes as a specific kind of subtrees called *infix subtrees* that preserve the sequential closure of activities. To answer **RQ2**, we propose the *Valid Tree Miner* algorithm, capable of efficiently mining frequent infix subtrees. Further, we have implemented the proposed algorithm in the open-source process mining tool Cortado [26], allowing the usage of the proposed algorithm in an end-user-oriented tool.

To support our solution to **RQ2**, we evaluate the algorithm using real-world event logs and compare it to a state-of-the-art frequent subtree mining algorithm [4]. Our findings indicate that the Valid Tree Miner outperforms the state-of-the-art algorithm. Additionally, we explore the number of frequent infix patterns for each event log, observing consistency in the result between the Valid Tree Miner and the state-of-the-art algorithm, as well as an exponential growth of the number with decreasing minimum support thresholds.

The remainder of this paper is structured as follows. Section 2 presents related work on frequent pattern mining. Following, Section 3 introduces background and preliminaries. We present the Valid Tree Miner in Section 4. Section 5 presents its implementation and an application scenario. Section 6 presents the evaluation of the Valid Tree Miner. Finally, Section 7 concludes this paper.

2 RELATED WORK & BACKGROUND

Variants summarize unique process executions of a process that have identical ordering relations among their executed activities. Variants find application in multitudes of tasks in process mining [31], including process discovery [5], process model conformance checking [8], and visual analytics [36]. This paper considers concurrency variants, recently introduced in [24]. Concurrency variants address two major short-coming of traditional variants [19]. Traditional variants 1) consider activities to be atomic, thus occurring on a singular time point, and 2) enforce a strict ordering of activities. Both create a representational bias as activities in real-world processes are performed in time intervals and concurrently. Concurrency variants overcome these limitations by modeling activities overlapping in time. Here to note is that the chosen abstraction level of concurrency variants does not differentiate the overlap of two activities, e.g., an activity execution is wholly contained in another activity's execution is modeled the same as them partially overlapping. However, detailed relations are often too fine-granular for many process mining analysis goals and result in a larger number of variants. A consequence of the chosen abstraction level is that, in rare cases, the relation of activities can not be modeled as solely sequential or concurrent. In such cases, an explicit fall-through represents the non-specific order of activities. We refer to [24] for an extensive introduction to concurrency variants.

In this paper, we show that the task of frequent infix mining can be reduced to frequent subtree mining by modeling concurrency variants as trees. We use trees as established techniques for traditional variants, i.e., frequent sequence mining [2], are incapable of representing the concurrency between activities and sequences possible in concurrency variants. Furthermore, techniques such as frequent episode mining [21] cannot reuse the existing tree structure and order of the concurrency variant to speed up computation.

We adopt the idea of frequent subtree mining to mine frequent subtrees corresponding to infixes. General overviews on frequent subtree mining are presented in [10, 18]. This paper focuses on *exact* methods that guarantee mining all frequent subtrees, compared

Table 1: Overview of frequent subtree mining algorithms. The checkmark ✓ indicates that the algorithm can mine the corresponding type of subtrees.

Algorithm	Infix	Induced	Closed	Maximal
FREQT [4]		✓		
CMTreeMiner [11]		✓	✓	✓
AMIOT [17]		✓		
IMB-3 [28]		✓		
TRIPS/TIDES [29]		✓		
PathJoin [37]		✓		✓
Valid Tree Miner	✓			

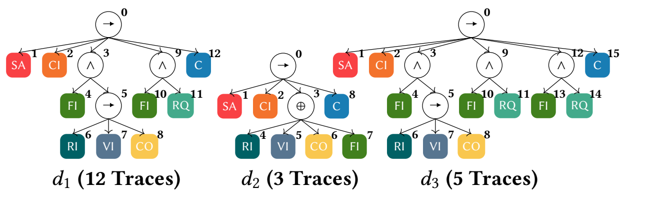


Figure 2: A (variant) tree bank consisting of three trees, d_1, d_2, and d_3. d_1 and d_3 represent the concurrency variants shown in Figure 1.

to *approximate* approaches. The trees we mine are labeled, rooted, ordered trees, with the subtrees we want to extract extending upon the definition of *induced subtrees*. Induced subtrees are subtrees of a tree that preserve the labeling, the child-parent relationship between nodes, and the order of siblings. Table 1 presents selected algorithms for frequent induced subtree mining. The proposed Valid Tree Miner mines a specific type of subtree called infix subtree. In addition to the induced subtree relation, infix subtrees preserve sequential completeness, meaning no activity in a sequence is skipped. No algorithms for mining frequent infix subtrees exist currently.

The presented algorithms [4, 11, 17, 28, 29] in Table 1 follow a generate-and-test approach, similar to the proposed Valid Tree Miner. These algorithms incrementally create new candidate subtrees from known frequent subtrees and prune the candidate patterns by applying the *apriori-principle* [3]. Candidate subtrees are tested if they are frequent in the tree bank, commonly using occurrence lists to speed up the computation. In the next iteration, frequent subtrees are then grown into new candidate trees. Furthermore, PathJoin and CMTreeMiner [11, 37] are capable of mining maximal and closed subtrees from the tree bank, not requiring their a-posteriori computation from the set of all frequent subtrees. Closedness means no supertree exists with the same support, while maximality means no frequent supertree exists.

In short, mining frequent infixes of concurrency variants is not possible via established techniques in process mining, such as frequent sequence mining. However, frequent infix mining can be mapped to the task of frequent subtree mining. Frequent subtree mining is well-established, especially for induced subtrees. However, induced subtrees are yet too lenient in their definition, not enforcing the sequential completeness we are interested in with infixes of concurrency variants. Consequently, we introduce infix subtrees enforcing sequential completeness and a mining algorithm.

3 PRELIMINARIES

First, we introduce definitions for labeled, rooted, ordered trees and their subtrees. Subsequently, we introduce event data and variants.

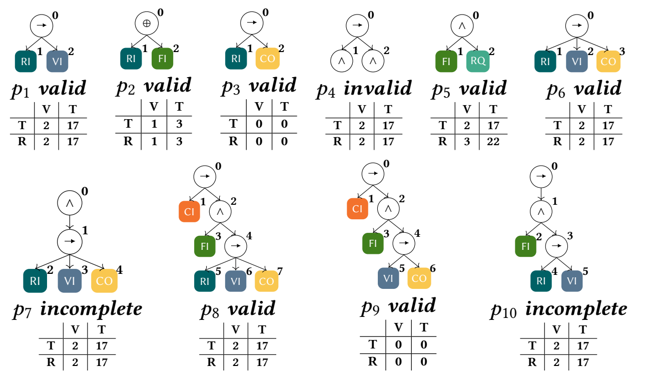
3.1 Trees

To model concurrency variants and their infixes, we use trees.

Definition 3.1 (Labeled, Ordered, Rooted Tree). A labeled, ordered, rooted tree t is a 6-tuple $(V_t, E_t, L_t, \lambda_t, r_t, <_t)$ where V_t is a set of nodes, $E_t \subseteq V_t \times V_t$ is a set of edges, L_t is a set of labels, $\lambda_t : V_t \rightarrow L_t$ is a surjective labeling function that assigns labels to nodes, and $r_t \in V_t$ is the root node. For $(v, v') \in E_t$, we call v the *parent* of v' and v' the *child* of v. If two nodes $v, v' \in V_t$ have the same *parent*, we call them *siblings*. Last, $<_t \subseteq V_t \times V_t$ is a strict partial order defined for every sibling pair; for siblings $v, v' \in V_t$ and $v \neq v'$, either $v <_t v'$ or $v' <_t v$ hold. Further, for two siblings $v, v' \in V_t$ with $v <_t v'$, we call them *immediate* siblings, denoted by $v \lessdot_t v'$, if $\nexists v^* \in V_t (v <_t v^* \wedge v^* <_t v')$. We denote the universe of labeled, ordered, rooted trees as $\mathcal{O}$.

We refer to a set $D \subset \mathcal{O}$ of labeled, rooted ordered trees as *tree bank*. The set of tree bank labels for a tree bank $D \subset \mathcal{O}$ is defined as $L_D = \bigcup_{d=(V_d, E_d, L_d, \lambda_d, r_d, <_d) \in D} L_d$. The Figure 2 shows an example tree bank. For now, one can ignore the difference in the shape of nodes. For a tree $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t) \in \mathcal{O}$. We define the *size* of a tree t as $|V_t|$. We consider trees in *normal form*, meaning all nodes are integers that are assigned in the order of the preorder traversal of the tree. In Figure 2, we write the assigned integer right above each node. The node $|V_t| - 1$ is called the right-most leaf of t, and we denote it as rml_t. Hereinafter, we refer in examples to a node $v \in V_t$ as $\lambda_t(v)^v$ to emphasize its label; for instance, consider $d_1 \in D \subset \mathcal{O}$ in Figure 2, we then write $r_{d_1} = \rightarrow^0$ and $rml_{d_1} = C^{12}$.

3.1.1 Functions on labeled, rooted, ordered trees. For a node $v \in V_t$ and $p \in \mathbb{N}_0$, we define $prt_t^p : V_t \nrightarrow V_t$ that returns the p-th parent of v with $prt_t^0(v) = v$. Consider d_1 (Figure 2) as an example, $prt_{d_1}^0(FI^4) = FI^4$, $prt_{d_1}^1(FI^4) = \wedge^3$, and $prt_{d_1}^2(FI^4) = \rightarrow^0$. For an arbitrary set X, we denote its power set as $\mathbb{P}(X) = \{S | S \subseteq X\}$. We define the function $chd_t : V_t \rightarrow \mathbb{P}(V_t)$ returning the children of a node $v \in V_t$. Further, $lmc_t : V_t \nrightarrow V_t$ and $rmc_t : V_t \nrightarrow V_t$ return the left- and right-most child of v. For example, for d_2 it holds that $lmc_{d_2}(\oplus^3) = RI^4$, $rmc_{d_2}(\oplus^3) = FI^7$ and $chd_{d_2}(\oplus^3) = \{RI^4, VI^5, CO^6, FI^7\}$. The function $dec_t : V_t \rightarrow \mathbb{P}(V_t)$ with $dec_t(v) = \{v' \in V_t | \exists n \in \mathbb{N} [v = prt_t^n(v')]\}$ returns the descendants of v. As an example, $dec_{d_1}(\wedge^3) = \{FI^4, \rightarrow^5, RI^6, VI^7, CO^8\}$. Last, we denote *paths* in t as sequences of unique nodes $\langle v_0, \ldots, v_n \rangle \in V_t^*$, such that an edge in E_t connects every pair of adjacent nodes in the sequence. A path of particular interest is the *right-most path*, i.e., the path $\langle rml_t, \ldots, r_t \rangle$ leading from the right-most leaf to the root node. Consider d_1 as an example, $\langle RI^6, \rightarrow^5, \wedge^3 \rangle$ is a path and $\langle C^{12}, \rightarrow^0 \rangle$ is the right-most path.



p_1 ***valid***

	V	T
T	2	17
R	2	17

p_2 ***valid***

	V	T
T	1	3
R	1	3

p_3 ***valid***

	V	T
T	0	0
R	0	0

p_4 ***invalid***

	V	T
T	2	17
R	2	17

p_5 ***valid***

	V	T
T	2	17
R	3	22

p_6 ***valid***

	V	T
T	2	17
R	2	17

p_7 ***incomplete***

	V	T
T	2	17
R	2	17

p_8 ***valid***

	V	T
T	2	17
R	2	17

p_9 ***valid***

	V	T
T	0	0
R	0	0

p_{10} ***incomplete***

	V	T
T	2	17
R	2	17

Figure 3: Induced (infix) subtrees for the (variant) tree bank in Figure 2. Beneath the subtrees is indicated if they are valid/invalid/incomplete. The table below each subtree shows the support of the *infix* subtree in the variant tree bank. The rows represent Transactions and Root-occurrence support, and the columns indicate Variant and Trace weighting.

We define *induced subtrees* via *one-to-one* mappings from the nodes of a subtree to a tree. The mapping must preserve the *labeling*, the *sibling order*, and the *child-parent* relationship of both the subtree and the tree. As an example for the tree bank in Figure 2 and the induced subtrees in Figure 3, p_1 is an induced subtree of d_1 and d_3. Further, p_2 is an induced subtree of d_2; observe that the sibling order is not restricted to immediate siblings in induced subtrees. For instance, an occurrence maps RI^1 of p_2 to RI^4 in d_2, while FI^2 of p_2 is mapped to FI^7 in d_2; RI^1 and FI^2 are immediate siblings in p_2, but not RI^4 and FI^7 in d_2. Last p_3 is not an induced subtree of any trees shown in Figure 2, as a mapping into d_1 or d_3 violates the sibling order, and a mapping into d_2 the labeling.

Definition 3.2 (Induced Subtree). Let $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t), t' = (V_{t'}, E_{t'}, L_{t'}, \lambda_{t'}, r_{t'}, <_{t'}) \in \mathcal{O}$. Tree t is an *induced subtree* of t', if an injective mapping $\delta_{t\to t'}: V_t \to V_{t'}$ for all $v, v' \in V_t$ preserves:

(Child-Parent) $(v, v') \in E_t \Leftrightarrow (\delta_{t\to t'}(v), \delta_{t\to t'}(v')) \in E_{t'}$

(Sibling order) $v <_t v' \Leftrightarrow \delta_{t\to t'}(v) <_{t'} \delta_{t\to t'}(v')$

(Labeling) $\lambda_t(v) = \lambda_{t'}(\delta_{t\to t'}(v))$

We write $t \sqsubseteq t'$ to denote t being an induced subtree of t', if $|V_t| < |V_{t'}|$ we write $t \sqsubset t'$. We then call t' a *supertree* of t. We define the set of all mappings between t and t' that preserve the above relations as $\Delta^{Ind}_{t\to t'} \subseteq \{\delta_{t\to t'}: V_t \to V_{t'}\}$.

Having defined induced subtrees, we are interested in the *support* of an induced subtree to quantify its number of occurrences in a tree bank. We introduce two different notions of support, *transaction* and *root-occurrences* support, as well as a weighting function that assigns each tree of a tree bank a weight. *Transaction* support counts the number of trees in the tree bank with at least one occurrence of a subtree, while *root-occurrence* support counts the number of occurrences with unique roots in the trees of the tree bank. Root-occurrence support addresses the limitation of transaction support to not count multiple occurrences per tree in a way that still allows for mining using the apriori-principle, cf. Subsection 3.1.3. Weighting functions consider that a tree, i.e., a concurrency variant, of the tree bank represents multiple traces. *Variant weighting* assigns every tree in the tree bank a weight of 1. In comparison, *trace weighting* assigns every tree in the tree bank a weight according to the number of traces its concurrency variant represents. As an example for the subtrees in Figure 3 and the tree bank in Figure 2 using variant weighting, p_1 has a transaction support of 2 given occurrences in d_1 (cf. nodes $\to^5$, RI^6, and VI^7) and d_3 (cf. nodes $\to^5$, RI^6, and VI^7); both trees having a weight of 1. The subtree p_5 has a transaction support of 2, but a root-occurrence support of 3, as p_5's root can be mapped to $\{\wedge^9\}$ in d_1 and to $\{\wedge^9, \wedge^{12}\}$ in d_3.

Definition 3.3 (Weighted Support). Let $D \subseteq \mathcal{O}$ be a tree bank, let $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t) \in \mathcal{O}$. For $d = (V_d, E_d, L_d, \lambda_d, r_d, <_d) \in D$, let $roots(t, d): \mathcal{O} \times \mathcal{O} \nrightarrow \mathbb{P}(V_d)$ with $roots(t, d) = \{\delta_{t\to d}(r_t) \in V_d \mid \delta_{t\to d} \in \Delta^{Ind}_{t\to d}\}$ return the nodes of d to which r_t is mapped. Let $w: D \to \mathbb{N}_0$ be a function assigning weights to trees of D. It follows

$$sup^w_{trans}(t, D) = \sum_{\{d \in D \mid t \sqsubseteq d\}} w(d) \tag{1}$$

$$sup^w_{root}(t, D) = \sum_{d \in D} w(d) \cdot |roots(t, d)| \tag{2}$$

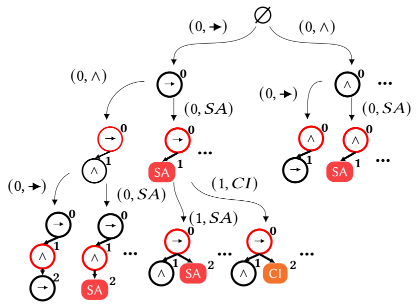


Figure 4: Excerpt of the right-most extension enumeration tree for subtrees of the tree bank in Figure 2. We annotate the edges with the respective (p, l)-extension and highlight the extended node in the trees in red.

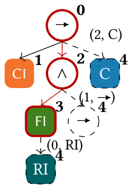


Figure 5: Candidate (p, l)-extensions on a subtree. The right-most path is highlighted in red, and examples of different candidate extensions are shown via dashed arcs and borders.

Next, we define the term *frequent* for induced subtrees. An induced subtree t is frequent in a tree bank D if its support is above a user-supplied threshold $m \in \mathbb{N}_0$, i.e., $sup(t, D) > m$. Consider the previous example, for root-occurrence support and minimum support of $m = 2$, p_5 from Figure 3 is frequent in the tree bank from Figure 2. At the same time, all other subtrees are *infrequent*.

3.1.2 *Right-Most Extension.*

To mine frequent subtrees, we utilize the idea of the right-most extensions. Observe that for any tree of size k by removing its right-most leaf, we can derive a smaller subtree of size $k-1$. Repeating the removal of the right-most leaf, one derives a chain of $k-1, k-2, \ldots, 1, 0$ sized trees. By reverting the incremental removal, i.e., incrementally appending a new right-most leaf from a given label set, we can uniquely enumerate any tree of size k. Incrementally appending a new right-most leaf forms an *enumeration tree*, shown on an example in Figure 4. To formalize the right-most enumeration, we introduce (p, l)-extensions that encode the position at which the new right-most leaf is appended in a tree and the label of the newly added leaf. We show an example of a subtree and potential (p, l)-extensions in Figure 5.

Definition 3.4 ((p, l)-Extension). Let $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t) \in \mathcal{O}$. A (p, l)-extension $(p, l) \in (\mathbb{N}_0 \times L_t)$ consists of an offset p along the right-most path of t representing $v_p = prt^p_t(rml_t) \in V_t$ and a label $l \in L_t$. We denote the set of all (p, l)-extensions of t as $P_t \subseteq (\mathbb{N}_0 \times L_t)$

3.1.3 *Apriori-principle.*

Using (p, l)-extensions, all frequent subtrees of a tree bank can be uniquely enumerated. However, enumerating all possible candidate subtrees is infeasible due to the combinatorial blow-up. Therefore, the apriori-principle is used. For a frequent subtree t in a tree bank D, it holds that if it is frequent, all its subtrees also are frequent. In the counterfactual, thus, only frequent subtrees need to be extended, as an infrequent subtree cannot be extend into a frequent subtree. Applying the apriori-principle depends on the *admissibility* of the support definitions, cf. Definition 3.3. A support definition is admissible if for a tree bank $D \subseteq \mathcal{O}$ $\forall t, t' \in \mathcal{O}\ t \subseteq t' \implies sup(t, D) \geq sup(t', D)$ holds. Thus, the support of a subtree is always greater or equal to that of its supertrees.

3.1.4 Right-Most Occurrence Set. We use tree embeddings to make the computation of the frequency of a subtree more tractable, foregoing explicit sub-isomorphism checking. The idea of a tree embedding is storing occurrences of a subtree in a tree bank and incrementally updating the embedding when growing the subtree. The occurrence information is then used for support computation and to verify frequency. For the right-most extension-based enumeration, Asai et al. [4] proposed the **R**ight-**M**ost **O**ccurrence list [4] to store the embeddings of induced subtrees in a tree bank. An RMO stores occurrence information as a tuple, based on the tree of the tree bank, and where the root and the right-most leaf of an induced subtree are mapped to in the tree of a tree bank. As an example, for p_5 from Figure 3 and d_3 from Figure 2, one occurrence of p_5 maps $rml_{p_5}{=}RQ^2$ to RQ^{11} and $r_{p_5}{=}\wedge^0$ to $\wedge^9$ in d_3. The occurrence is then represented by the tuple $(d_3, RQ^{11}, \wedge^9)$ in an RMO.

Definition 3.5 (RMO Entry). Let $t{=}(V_t, E_t, L_t, \lambda_t, r_t, <_t) \in \mathcal{O}$ and $D \subseteq \mathcal{O}$. An RMO entry is a triple $(d, v_{rml}, r) \in (D \times V_d \times \mathbb{N}_0)$ that represents $\delta_{t \to d} \in \Delta^{Ind}_{t \to d}$ with $v_{rml} = \delta_{t \to d}(rml_t)$ and $r = \delta_{t \to d}(r_t)$.

Having defined the RMO entries, we define the RMO of a tree $t{=}(V_t, E_t, L_t, \lambda_t, r_t, <_t) \in \mathcal{O}$ in a tree bank $D \subseteq \mathcal{O}$ as a set. For a $d \in D$ we define $RMO^t_d{=}\{(d, \delta_{t \to d}(v_{rml}), \delta_{t \to d}(r_t)) \mid \delta_{t \to d} \in \Delta^{Ind}_{t \to d}\}$ and for the tree bank D we define $RMO^t_D{=}\bigcup_{d \in D} RMO^t_d$. As an example, for p_5 from Figure 3, p_5's RMO in Figure 2 is given by $RMO^{p_5}_D{=}\{(d_1, RQ^{11}, \wedge^9), (d_3, RQ^{11}, \wedge^9), (d_3, RQ^{14}, \wedge^{12})\}$.

3.1.5 Closed & Maximal Subtrees. A common issue of frequent subtree mining is that the number of mined frequent subtrees can be large to the extent that human analysis becomes infeasible. The apriori-principle explains the large number of frequent subtrees; for a frequent subtree, all its subtrees are both frequent and cover the same information redundantly. To address the redundancy, the idea of *closedness* and *maximality* of subtrees emerged [10, 11]. A subtree is *closed* if none of its proper supertrees have the same support. Similarly, a subtree is *maximal* if no frequent supertree exists.

Definition 3.6 (Closed and Maximal Subtrees). Let $D \subseteq \mathcal{V}$ be a variant tree bank, let $m \in \mathbb{N}_0$ be a minimum support, let $F^{D,m}$ be the set of frequent subtrees of D with minimum support above m and let $t \in \mathcal{F}^{D,m}$ be a frequent subtree. Let *sup* be a support function as defined in Definition 3.3.

$$t \text{ is closed} \Leftrightarrow \nexists t' \in F^{D,m} [t \sqsubset t' \wedge sup(t, D) \leq sup(t', D)]$$
$$t \text{ is maximal} \Leftrightarrow \nexists t' \in F^{D,m} [t \sqsubset t']$$

For example, consider the subtrees in Figure 3, the variant tree bank in Figure 2, variant-weighted root-occurrence support, and a minimum support of $m = 1$. Then p_5 is closed, as none of its supertrees has the same support of 3. Further, for minimum support of $m = 2$, p_5 is maximal as no frequent supertree of it exists. As a further example, p_1 is not closed, as its supertree p_6 also has a support of 2, and thus p_1 is also not maximal.

3.2 Event Data & Variants

An event log is a set of activity instances that describe activities performed in a process, cf. Figure 1. We assume each activity instance has two timestamps modeling the *start* and *end* of its execution, as well as a *case identifier* and a *label* representing a specific activity. Using the case identifiers, multiple activity instances are grouped into a single *case*, i.e., an individual process execution (cf. Figure 1). Activity instances represent a time interval in which the activity is performed. Thus multiple activity instances in the same case can overlap, i.e., one activity starts before the other activity is completed. We refer to temporal overlapping activities as *concurrent*. If activities do not overlap, i.e., one activity strictly ends before the other starts, we call them *following*. The concurrency and follows relations between activity instances is a subclass of partial orders called an interval order [15]. The interval order is used to derive a representation of the concurrency and sequentiality of activity instances in a case. We call this representation a concurrency variant [24]. We present a concurrency variant and the interval plot of a possible trace of the variant in Figure 1. The colored chevrons of the variant, cf. Figure 1, represent distinct activities performed in the variant. The left-to-right order of the variant indicates the sequentiality of activities, while stacking activities above each other indicates concurrency. As is shown in Figure 1, a sequence of activities, *RI*, *VI*, *CO* can be concurrent to a single activity, *FI*. An implication of the abstraction level introduced by concurrency variants [24] is that certain interval orders of activity instances cannot be represented, as the relation between groups of activity instances is neither strictly concurrent nor sequential. In these cases, the relation between the activities is abstracted as an *fallthrough*. An example of a fallthrough would be an activity *RI* being followed by an activity *VI* and both happening concurrently to an activity *CO*. Then an activity *FI* concurrent to *VI*, but following *RI* and *CO*, would allow for no partition of the activities based on sequentiality and concurrency between activities.

We represent concurrency variants as rooted, ordered labeled trees, cf. Definition 3.1. We refer to a tree representing a concurrency variant as a *Variant Tree*. As an example, the concurrency variants representing cases 1, 2, and 3 from Figure 1 correspond to the variant trees d_1 and d_3 of Figure 2. On variant trees, some restrictions compared to labeled, rooted ordered trees (cf. Definition 3.1) hold based on their derivation. The variant tree's labels consist of *activity labels* L_{act} and *operator labels* $L_{op} = \{\rightarrow, \wedge, \oplus\}$. The inner nodes are labeled with operator labels only, shown as circles in Figure 2. Further, every inner node has at least two children. The leaf nodes are labeled with activity labels only and are colored squares in Figure 2. The operator labels are either modeling sequentially ($\rightarrow$), concurrency ($\wedge$), or a fallthrough ($\oplus$). Operator nodes with the same label cannot be children of another. Similarly, the fallthrough operator node cannot have operator node children and concurrent operator nodes at most one operator node child. The latter is a consequence of using interval orders to represent the concurrency in the derivation of concurrency variants. Last, to enforce consistency across variant trees, the children's order of the fallthrough and the concurrent operator follows a lexicographical order defined over the labels, with operator labels sorting highest, i.e., as the right-most child. Next, we define variant trees as a subclass of labeled, ordered, rooted trees.

Definition 3.7 (Variant Tree). We denote a variant tree as a 7-tuple $d{=}(V_d, E_d, L_d, \lambda_d, r_d, <_d, n_d^{trc}){\in}\mathcal{V}$, where $(V_d, E_d, L_d, \lambda_d, r_d, <_d){\in}\mathcal{O}$ is a labeled, rooted ordered tree representing a concurrency variant [24] and $n_d^{trc}{\in}\mathbb{N}$ is the number of traces of the variant. Let $\leq_{Lex}$ be a lexicographical order over the set of labels $L_d{=}L_{op}{\cup}L_{act}$, with $\forall\, l_{act}{\in}L_{act},\, l_{op}{\in}L_{op}\,[l_{act}{\leq_{Lex}}l_{op}]$, where $L_{op}{=}\{\rightarrow, \wedge, \oplus\}$. Based on the definition of concurrency variants [24], the following restrictions regarding d hold for arbitrary $v{\in}V_d$:

$\lambda_d(v) \notin L_{op} \Leftrightarrow |chd_d(v)| = 0$
$\lambda_d(v) \in L_{op} \Leftrightarrow |chd_d(v)| \geq 2$
$\lambda_d(v) \in L_{op} \Rightarrow \nexists v' \in chd_d(v)[\lambda_d(v') = \lambda_d(v)]$
$\lambda_d(v) = \wedge \Rightarrow \nexists v', v''{\in}chd_d(v)[v'{\neq}v'' {\wedge} \lambda_d(v'), \lambda_d(v'') \in L_{op}]$
$\lambda_d(v) = \oplus \Rightarrow \nexists v' \in chd_d(v)[\lambda_d(v') \in L_{op}]$
$\forall v', v''{\in}chd_d(v)\big([\lambda_d(v){\neq}{\rightarrow} \wedge v'{<_d}v''] \Rightarrow \lambda_d(v'){\leq_{Lex}}\lambda_d(v'')\big)$

The universe of variant trees is denoted by $\mathcal{V}$. For a variant tree bank $D \subseteq \mathcal{V}$, we denote the set of activity labels as $L_{act}^D = L_D \backslash L_{op}$.

3.2.1 *Directly follows relations.* For variant trees and their subtrees, we define the (directly)-follows relations of their activities. To this end, we look at the lowest-common ancestor (LCA) of two activity leaves. The LCA being the node, if it exists, that is the lowest shared parent of two nodes in a tree. As an example, observe that for tree d_1 (cf. Figure 2), the *LCA* of SA^1 and C^{12} is $\rightarrow^0$. As their LCA's label is ($\rightarrow$), we then say that SA^1 is eventually followed by C^{12} in d_1. We denote this as $SA^1{<}_{EF}^{d_1}C^{12}$. As a counter-example $FI^4 {\nless}_{EF}^{d_1} RI^6$, as their *LCA* is a concurrent operator $\wedge^3$. If no activity occurs in between two activities eventually following each other, we denote them as directly-following each other, e.g., $SA^1{<}_{DF}^{d_1}CI^2$. The (directly) follows relation holds across different tree levels, e.g., $CI^2{<}_{DF}^{d_1}RI^6$. Last, we cannot establish direct-follow relations across tree levels in case of a fallthrough, as it indicates no specific relation between activities. Thus, for d_3, $CI^2{\nless}_{DF}^{d_2}RI^4$, but $CI^2{<}_{EF}^{d_2}RI^4$.

4 MINING FREQUENT VALID INFIX SUBTREES

To mine frequent infix patterns from concurrency variants, we exploit their inherent tree structure and adapt frequent subtree mining. This section introduces the *Valid Tree Miner* that follows a *generate and test* approach shown in Figure 6. From the concurrency variants of the event log, a set of initial frequent infix subtrees is computed. Subsequently, the subtrees are grown into new candidate infix subtrees of a larger size. The frequency of the candidates is then verified, and if they are frequent, they are again grown to generate new candidate infix subtrees of a larger size. The growth and verification are repeated until all frequent infix subtrees of the concurrency variants are mined.

First, Subsection 4.1 models infixes of concurrency variants as infix subtrees. In Subsection 4.2, we introduce the Valid Tree Miner algorithm to compute the frequent infix subtrees of a variant tree bank. Finally, Subsection 4.3 and Subsection 4.4 close out on the

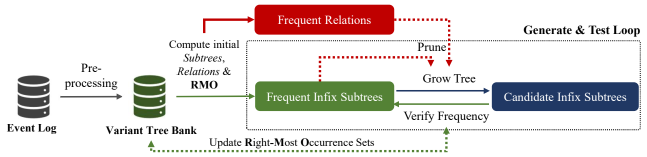


Figure 6: Overview of the proposed Valid Tree Miner frequent infix subtree mining algorithm.

algorithm by presenting a correctness proof of the support function for infix subtrees and a runtime analysis of the Valid Tree Miner.

4.1 Infix Patterns in Process Execution Variants

This paper focuses on infix patterns that are sequentially closed, i.e., no activity is skipped within a sequence. Sequential closedness makes the mined infixes represent closed sequences of activities, not skipping a process activity. We model infix patterns based on induced subtrees, cf. Definition 3.2. Infix subtrees are labeled, rooted ordered trees that share the same label set as the variant tree bank they are a subtree of but are not necessarily variant trees. The difference between induced and infix subtrees is shown in the following example. For the subtrees from Figure 3, we know that p_3 is an induced subtree of both d_1 and d_3 of Figure 2. However, it is neither an infix subtree of d_1 nor d_3 because any potential occurrence of p_3 skips the activity *VI* in d_1 and d_3 in sequential order. The sequential completeness must further hold across tree levels, i.e., p_8 is a subtree of d_1 and d_3, while p_9 is not, as activity *RI*, sequentially between *CI* and *VI*, would be skipped for all potential occurrences of p_9 in the tree bank.

Definition 4.1 (Infix Subtree). Let $D \subseteq \mathcal{V}$ be a variant tree bank and $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t)$, $t' = (V_{t'}, E_{t'}, L_{t'}, \lambda_{t'}, r_{t'}, <_{t'}){\in}\mathcal{O}$ with $L_t, L_{t'}{\subseteq}L_D$. We say that t is an infix subtree of t', if $t{\sqsubseteq}t'$, and there exists $\delta_{t\rightarrow t'}{\in}\Delta_{t\rightarrow t'}^{Ind}$ that preserves the following relation for every $v, v' \in V_t$:

$$v <_{DF}^{t} v' \Leftrightarrow \delta_{t\rightarrow t'}(v) <_{DF}^{t'} \delta_{t\rightarrow t'}(v')$$
$$(v <_t v' \wedge \lambda_t(prt_t^1(v)){=}{\rightarrow}) \Leftrightarrow \delta_{t\rightarrow t'}(v) <_{t'} \delta_{t\rightarrow t'}(v')$$

We define the set of all $\delta_{t\rightarrow t'}$ that preserves the above relations as $\Delta_{t\rightarrow t'}^{Infix}$. For infix subtree t of t', we write $t{\subseteq}t'$ and $t{\subset}t'$, if $|V_t|{<}|V_{t'}|$.

We use the infix subtree relation, $\subseteq$ see Definition 4.1, with the notations previously introduced for induced subtrees, foremost Definition 3.3. Furthermore, we introduce a *trace*-weighting function $w_{trace}{:}\,\mathcal{V}{\longrightarrow}\mathbb{N}_0$, that for a variant tree $d{=}(V_d, E_d, L_d, \lambda_d, r_d, <_d, n_d^{trc}){\in}\mathcal{V}$ weights occurrence according to the number of traces of d, $w_{trace}(d){=}n_d^{trc}$. Figure 3 shows the different infix subtrees and their corresponding support values for the variant tree bank shown in Figure 2. As an example, the trace-weighted root-occurrence support of p_5 is computed given one occurrence of p_5 in d_1 (12 traces) and two occurrences in d_3 (5 traces), as $sup_{root}^{w_{trace}}(p_5, D) = 22 = 1{\cdot}12 + 2{\cdot}5$.

Not every infix subtree corresponds to an interesting infix pattern of the concurrency variants. Since the operator nodes of a variant tree model the relation between activities, infix subtrees containing operator nodes without children show little information. To encompass this, we define *valid* and *invalid* infix subtrees. An infix subtree is *valid* if all operator nodes have at least *two* children, else it is *invalid*. Given the right-most enumeration (cf. Definition 3.4), infix subtrees are incrementally grown into larger trees by appending a new right-most leaf along the right-most path. Thus, invalid subtrees with all operator nodes having less than two children located on the right-most path can still be grown into a valid infix subtree. We call these infix subtrees *incomplete*. We indicate the (in)validness for the subtrees in Figure 3.

Definition 4.2 (Valid & Incomplete Trees). Let $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t) \in O$. We define *valid* and *incomplete*.

$$t \text{ is valid} \Leftrightarrow (\forall\, v \in V_t\ (\lambda_t(v) \in L_{op} \Rightarrow |chd_t(v)| > 1))$$
$$t \text{ is incomplete} \Leftrightarrow (t \text{ is not valid} \wedge \forall\, v \in V_t\, [(\lambda_t(v) \in L_{op} \wedge |chd_t(v)| < 2) \Rightarrow v \in \langle rml_t, \dots, r_t \rangle])$$

If t is neither *valid* nor *incomplete*, we call it *invalid*.

4.2 Valid Tree Miner

The proposed Valid Tree Miner, cf. Algorithm 1, follows a generate-and-test approach. The generate-and-test routine breadth-first traverses the right-most enumeration tree of the infix subtrees of the variant tree bank using the apriori-principle to prune infrequent subtrees. Through exhaustive search, the algorithm is exact as it enumerates and checks all frequent valid infix subtrees.

The first step of the Valid Tree Miner is the computation (line 1) of the initial set of frequent infix subtrees, which are later grown into larger candidate infix subtrees. Next, we initialize a processing queue Q (line 2) using the set of initial frequent infix subtrees. While Q contains frequent infix subtrees, these are gradually processed (lines 3-4). For the frequent infix subtree t, we compute feasible extensions using a pruning function (line 5) and then verify if the extended candidate infix subtree is frequent. The verification is done by computing the updated RMO of the extend infix subtree t' based on the RMO of t (line 6). Based on the RMO, we verify if t' is frequent (line 7). If so, t' is added to the result set of frequent infix subtrees $F^{D,m}$ (line 9) and the processing queue Q (line 10). Finally, we filter out the incomplete subtree from the set of frequent infix subtrees (line 14).

Algorithm 1: Valid Tree Miner

```
   Input: $D \subseteq \mathcal{V}, m \in \mathbb{N}_0$
   Output: $F^{D,m}_{valid} \subseteq O$
1  $F^{D,m} \leftarrow GenerateInitialFrequentSubtrees(D, m)$
2  $Q \leftarrow F^{D,m}$                                   //Initialize the processing queue
3  while $Q \neq \emptyset$ do                            //While trees left to process
4      $t \leftarrow Q.dequeue()$                         //Process the next frequent tree
5      for $(p, l) \in GrowTree(t)$ do                    //For every $(p, l)$-extension
6          $t' \leftarrow grow(t, (p, l))$                //Grow $t$ into $t'$ based on $(p, l)$
7          $RMO^{t'}_D \leftarrow UpdateRMO(RMO^t_D, (p, l))$    //Compute RMO of $t'$
8          if $sup(t', D) > m$ then                       //If $t'$ is frequent
9              $F^{D,m} \leftarrow F^{D,m} \cup \{t'\}$   //Add $t'$ to the frequent trees
10             $Q.enqueue(t')$                            //Add $t'$ to the processing queue
11 $F^{D,m}_{valid} \leftarrow FilterIncomplete(F^{D,m})$      //Filter incomplete trees
12 return $F^{D,m}_{valid}$
```

4.2.1 *Generating Initial Pattern Candidates.*

The Valid Tree Miner algorithm starts by computing the sets of frequent infix subtrees of size three, i.e., $F^{D,m}_3$, from the variant tree bank as the initial frequent subtrees. The algorithm starts with size three, as these are the first possible valid infix subtrees. The initial subtrees and their right-most occurrence sets, cf. Definition 3.5, are computed in a single pass over the variant tree bank. The pass over the variant tree bank also computes the sets of frequent (directly)-follows relations. The frequent relations sets are used for pruning and are computed based on counting the number of activity leaf pairs with the respective relation, i.e., $<_{DF}$ and $<_{EF}$. We denote them as $\mathcal{F}^{D,m}_{DF} \subset L^D_{act} \times L^D_{act}$ and $\mathcal{F}^{D,m}_{EF} \subset L^D_{act} \times L^D_{act}$.

4.2.2 *Growing Patterns.*

Frequent infix subtrees are grown into larger candidate subtrees using right-most extension, cf. Algorithm 1 lines 5-6. The growth algorithm is shown in Algorithm 2. GrowTree takes a frequent infix subtree as input, moves up along the tree's right-most path, computes all feasible (p, l)-extensions at each position, and outputs a set of (p, l)-extensions.

An infeasible extension is an extension that results in a structure not observed in a variant tree, an invalid subtree, or an extension that cannot be frequent based on the apriori-principle. To prevent *infeasible* extensions, the algorithm utilizes two pruning strategies: *structure-based pruning*, which prevents extensions leading to invalid subtrees, and *label-based pruning* using information from previous iterations of the algorithm. Structure-based pruning is implemented during the processing along the right-most path in Algorithm 2, stopping early if an infeasible extension would be made. Label-based pruning is implemented as a function (cf. Algorithm 2, line 4), which returns the pruned set of labels feasible at a given extension position along the right-most path based on the application of the apriori-principle.

Algorithm 2: GrowTree

```
  Input: $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t) \in O$
  Output: $C = \{(p_1, l_1), \dots (p_n, l_n)\} \subseteq \mathbb{P}(P_t)$
1 $p \leftarrow 0$                                              //Initialize Offset Counter
2 for $v_{extend} \in \langle rml_t, \dots, r_t \rangle$ do       //For every node along the right-most path
3     if $\lambda_t(v_{extend}) \in L_{op}$ then                  //Only extend on operators
4         $C \leftarrow C \cup LabelPrn(t, p)$
5         if $|chd_t(v_{extend})| < 2 \wedge |V_t| > 2$ then
6             break                                 //Break early if the node only has 2 children
7     $p \leftarrow p + 1$
8 return C
```

4.2.3 *Structure-based pruning.*

Consider Figure 5, for the subtree and (p, l)-extensions depicted, one easily sees that both the (p, l)-extensions $(0, RI)$ and $(2, C)$ would be *infeasible*. The former, $(0, RI)$, results in a pattern where two nodes with an activity label are nested under one another. Variant trees of this form do not exist based on the restrictions defined in Definition 3.7. The latter, $(2, C)$, results in an *invalid* subtree, cf. Definition 4.2. To prune infeasible extensions, we use two checks in GrowTree (Algorithm 2). We only extend on operator nodes (line 3) and terminate the backtracking after processing a node with less than two children (line 5).

4.2.4 *Label-based pruning.*

The function *LabelPrn* (cf. Algorithm 2, line 4) represents label-based pruning and is composed of two different pruning strategies introduced below. The first label-based pruning strategy *tree pruning* utilizes the apriori-principle to prune extensions in case the subtrees of sizes 3 are not frequent. Observe that if any of the subtrees of an extended subtree would be infrequent, the extension is redundant as the extended tree cannot be

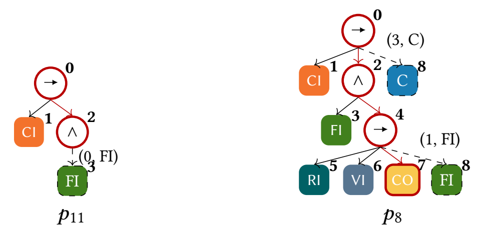


Figure 7: Candidate (p, l)-extensions on two subtrees. Structurally pruned extension positions are not considered. The right-most path is demarcated in red, and the candidate extensions are shown via dashed arcs and borders.

frequent by the apriori principle. Depending on the extension position p of the (p, l)-extension, we consider two forms of subtrees of size 3. For p=0, we consider the subtree of the extended tree, consisting of the new right-most leaf and its parent and grandparent. For example, consider p_{11} in Figure 7. For the extension $(0, FI)$, we first verify if the subtree consisting of $\rightarrow^0$, $\wedge^2$ and FI^3 is frequent, i.e., contained in the set $F^{D,m}$. For the other case, p>0, the subtree is made up of the right-most leaf; its left-sibling, the $p-1$-th node; and its parent, the p-th node. As an example, for this, consider extension $(1, FI)$ on p_8 in Figure 7. Then, we verify that the corresponding tree with root $\rightarrow^4$ and children, CO^7 and FI^8, is frequent and thus the (p, l)-extension feasible.

Definition 4.3 (Tree Prune). Let $D \subseteq \mathcal{V}$ be a variant tree bank, minimum support $m \in \mathbb{N}_0$, $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t) \in F^{D,m}$, and $p \in \mathbb{N}_0$. Let $v_p = prt_t^p(rml_t)$; if p>0 let $v_{p\text{-}1} = prt_t^{p\text{-}1}(rml_t)$; let $v_{p+1} = prt_t^{p+1}(rml_t)$. Let $l_{p+1}{=}\lambda_t(v_{p+1})$, $l_p{=}\lambda_t(v_p)$, $l_{p\text{-}1}{=}\lambda_t(v_{p\text{-}1})$. Let $F_3^{D,m}$ be the frequent subtrees of size 3 of D and let t=(V_t, $E_t, L_t, \lambda_t, r_t, <_t) \in F_3^{D,m}$. We define $f_{3,Sib}^{D,m} : (L_D \times L_D) \to \mathbb{P}(L_D)$ with $f_{3,Sib}^{D,m}(l_p, l_l) {=} \{\lambda_t(v_r) \mid v_l, v_r {\in} chd_t(r_t) \wedge v_l {<_t} v_r \wedge \lambda_t(r_t){=}l_p \wedge \lambda_t(v_l){=}l_l\}$ and $f_{3,Nest}^{D,m} : (L_D \times L_D) \to \mathbb{P}(L_D)$ with $f_{3,Nest}^{D,m}(l_g, l_p) = \{\lambda_t(v_c) \mid \lambda_t(r_t){=}l_g \wedge v_p {\in} chd_t(r_t) \wedge v_c {\in} chd_t(v_p) \wedge \lambda_t(v_p){=}l_p\}$. The functions $f_{3,Nest}^{D,m}$ and $f_{3,Sib}^{D,m}$ return the labels of the frequent subtree of size 3 with parent label l_p and respective left sibling l_l or grandparent label l_g. We then define the function, $TreePrn(t, p)$: $O \times \mathbb{N}_0 \nrightarrow \mathbb{P}(P_t)$

$$TreePrn(t, p) = \begin{cases} \{(p, l) \mid l \in L_D \wedge l \in f_{3,Nest}^{D,m}(l_{p+1}, l_p)\} & \textit{if } p = 0 \\ \{(p, l) \mid l \in L_D \wedge l \in f_{3,Sib}^{D,m}(l_p, l_{p\text{-}1})\} & \textit{if } p > 0 \end{cases}$$

The second label-based pruning strategy *relation prune* utilizes the frequent relation sets, i.e., $\mathcal{F}_{DF}^{D,m}$ and $\mathcal{F}_{EF}^{D,m}$, computed during the initial candidate generation (cf. Subsection 4.2.1). For an extension next to an operator node, one can show that for all pairs of the descendant activity leaves and the newly extended activity leaf sibling, the (directly)-follows relation is frequent. For instance, for the extensions $(3, C)$ on p_8 in Figure 7 as $FI^3 {<_{DF}} C^8$ and $CO^7 {<_{DF}} C^8$ hold in the extended tree, we verify that $(CO, C), (FI, C) {\in} \mathcal{F}_{DF}^{D,m}$ holds. Similarly, in the extended tree $RI^5 {<_{EF}} C^8$ and $VI^6 {<_{EF}} C^8$ hold, thus we verify if $(RI, C), (VI, C) {\in} \mathcal{F}_{EF}^{D,m}$. Only if all such relations are frequent the corresponding extended subtree can be frequent

Definition 4.4 (Relations Prune). Let $D \subseteq \mathcal{V}$ be a variant tree bank, minimum support $m {\in} \mathbb{N}_0$, $t{=}(V_t, E_t, L_t, \lambda_t, r_t, <_t) {\in} F^{D,m}$, and $p {\in} \mathbb{N}$. Let $v_p{=}prt_t^p(rml_t)$ and let $v_{p\text{-}1} = prt_t^{p\text{-}1}(rml_t)$. Let $l_p = \lambda_t(v_p)$, $l_{p\text{-}1}{=}\lambda_t(v_{p\text{-}1})$. Let $L_{EF} = \{\lambda_t(v_d) \mid v_d {\in} dec_t(v_{p\text{-}1}) \wedge \lambda_t(v_d) {\in} L_{act}^D\}$ and $L_{DF}{=}\{\lambda_t(v_d) \mid v_d {\in} dec_t(v_{p\text{-}1}) \wedge \lambda_t(v_d) {\in} L_{act}^D \wedge \lambda_t(prt_t^1(v_d)) {\neq} \oplus \wedge \nexists v' {\in} V_t [v_d <_{EF}^t v']\}$. Let $rhdl_\square^{D,m} : L_{act}^D \to \mathbb{P}(L_{act}^D)$ for $\square {\in} \{EF, DF\}$ with $rhdl_\square^{D,m}(l) = \{l' {\in} L_{act}^D \mid (l, l') {\in} \mathcal{F}_\square^{D,m}\}$. Using $rhdl_\square^{D,m}$, we define $L_{prn}{=} \bigcap_{l_d \in L_{EF}} rhdl_{EF}^{D,m}(l_d) \cap \bigcap_{l_d \in L_{DF}} rhdl_{DF}^{D,m}(l_d)$. Then, we define $RelPrn : O \times \mathbb{N}_0 \nrightarrow \mathbb{P}(P_t)$, that computes the set of (p, l)-extensions after applying the relation-based pruning.

$$RelPrn(t, p) = \{(p, l) \mid l \in (L_{prn} \cup \{\wedge, \oplus\})\}$$

Combining the two pruning strategies Definition 4.3 and 4.4, we define the label-pruning function (cf. Algorithm 2, line 4) as follows.

Definition 4.5 (Label Prune). Let $D \subseteq \mathcal{V}$ be a variant tree bank, $m {\in} \mathbb{N}_0$ a minimum support, and $t = (V_t, E_t, L_t, \lambda_t, r_t, <_t) {\in} F^{D,m}$ and $p {\in} \mathbb{N}_0$. Let $l_p = \lambda_t(prt_t^p(rml_t))$; let $l_{p\text{-}1} = \lambda_t(prt_t^{p\text{-}1}(rml_t))$, if $p > 0$. We define the pruning function $LabelPrn : O \times \mathbb{N}_0 {\nrightarrow} \mathbb{P}(P_t)$, that for a given p offset along the right-most path of t computes the pruned set of (p, l)-extensions.

$$LabelPrn(t, p){=}\begin{cases} TreePrn(t, p) {\cap} RelPrn(t, p) & p{>}0 {\wedge} l_p{=}{\rightarrow} {\wedge} l_{p\text{-}1} {\in} L_{op} \\ TreePrn(t, p) & \textit{otherwise} \end{cases}$$

4.2.5 Updating the RMO Set. After computing the feasible extensions of the frequent infix subtrees of size k into candidate infix subtrees of size $k + 1$ (cf. Algorithm 1, line 6), we compute the RMO of the candidate by updating the RMO of its frequent infix subtree parent. Figure 8 shows the update routine. Using the right-most occurrences of an infix subtree stored in the RMO and a (p, l)-extension, one first computes the extension position in the variant tree for every occurrence by backtracking to the p-th parent of the node. From this extension position, a sibling of the $p-1$-th parent matching the label l of the (p, l)-extension is searched. If the $p-1$-th parent does not exist, all children of the p-th parent are searched instead. If a node with a label matching l is found, a new entry for the RMO of the extended infix subtree is created.

Further restrictions are made to ensure the sequential completeness of the infix subtree. If the p-th parent is a sequential operator, only the immediate right sibling of the $p-1$-th parent is considered instead of all siblings. Similarly, for sequential operators, if p=0, only the left-most child is checked instead of all children if an activity exists that happened sequentially before the explored children. Last, backtracking is stopped if it passes a node, e.g., CO^8, that would be skipped in sequential order, i.e., its to the right of the backtracking path under a sequential operator. We introduce the function *Entries* that given a right-most occurrence (cf. Definition 3.5) of infix subtree t in a variant tree bank and a (p, l)-extension computes right-most occurrences of the (p, l)-extended infix subtree t', based on the RMO of t.

Definition 4.6 (Entries). Let $D \subseteq \mathcal{V}$ be a variant tree bank, minimum support $m {\in} \mathbb{N}_0$, and RMO_D^t of $t{=}(V_t, E_t, L_t, \lambda_t, r_t, <_t) {\in} F^{D,m}$. Let $(p, l) {\in} P_t$ and $t' {\in} O$ be the (p, l)-extended tree of t. For $d{=}(V_d$, $E_d, L_d, \lambda_d, r_d, <_d, n_d^{trc}) {\in} D$ and a RMO entry $(d, v_o, r) {\in} RMO_d^t$, let $v_p{=}prt_d^p(v_o)$. If p>0, let $v_{p\text{-}1}{=}prt_d^{p\text{-}1}(v_o)$. We define $LAnc \equiv \exists v {\in} \langle rml_t, \ldots, r_t \rangle [\lambda_t(v){=}{\rightarrow} \wedge lmc_t(v) {\notin} \langle rml_t, \ldots, r_t \rangle]$ and $RAnc \equiv \nexists v \in$

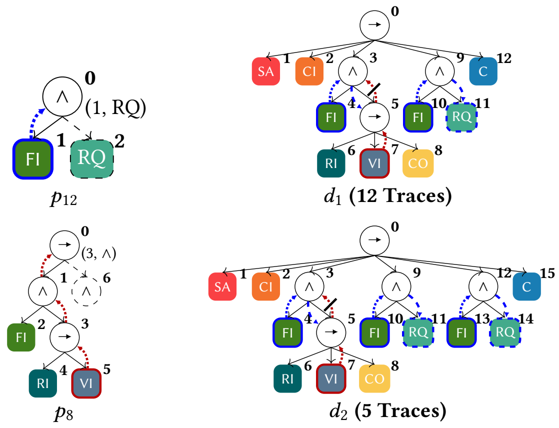


Figure 8: RMO update step for (p,l)-extensions on subtrees p_8 and p_{12}, for the variant trees d_1 and d_2 from Figure 2. To find a matching occurrence of the extended infix subtree, one backtracks to the p-th parent of the right-most occurrence of the infix subtree in the variant tree. At the p-th parent, one then searches the siblings right to the $p-1$-th parent or of the node itself if p=0. The *dotted* arcs indicate the backtracking, and the *dashed* arcs indicate the children that are searched for a matching label. To ensure sequential completeness the backtracking is stopped when passing a node that violates sequential completeness, as is the case for the backtracking for p_8 and the node CO^8 in d_1 and d_2.

$\langle v_o,\ldots,v_{p\text{-}1}\rangle(\lambda_d(v)=\dashrightarrow \wedge\ rmc_d(v)\notin\langle v_o,\ldots,v_{p\text{-}1}\rangle)$. Then, the function $Entries : (\mathbb{P}((D\times V_d\times V_d))\times P_t)\to\mathbb{P}((D\times V_d\times V_d))$ computes the entries in $RMO^{t'}_d$ based on the (p,l)-extension on $(d,v_o,r)\in RMO^t_d$.

$$Entries((d,v_o,r),(p,l)) = \begin{cases} \{(d,v_c,r)\mid v_c{=}lmc_d(v_p),\lambda_d(v_c){=}l\} & LAnc\wedge\lambda_d(v_p){=}{\dashrightarrow}\wedge p{=}0\\ \{(d,v_c,r)\mid v_c{\in}chd_d(v_p),\lambda_d(v_c)=l\} & (\neg LAnc\vee\lambda_d(v_p){\neq}{\dashrightarrow})\wedge p{=}0\\ \{(d,v_r,r)\mid v_r{=}rSib_d(v_{p\text{-}1}),\lambda_d(v_r){=}l\} & \neg RAnc\wedge\lambda_d(v_p){=}{\dashrightarrow}\wedge p{>}0\\ \{(d,v_r,r)\mid v_r{\in}rSibs_d(v_{p\text{-}1}),\lambda_d(v_r){=}l\} & p{>}0\wedge\lambda_d(v_p){\neq}{\dashrightarrow}\\ \emptyset & otherwise\end{cases}$$

Using the function *Entries*, we define *UpdateRMO* that computes the full RMO update step, i.e., computing the RMO of a (p,l)-extended subtree based on the RMO of the extended subtree. Using the RMO, the support of the extended subtree is then computed.

Definition 4.7 (UpdateRMO). Let $D\subseteq\mathcal{V}$ be a variant tree bank, minimum support $m\in\mathbb{N}_0$, and $t=(V_t,E_t,L_t,\lambda_t,r_t,<_t)\in F^{D,m}$ with RMO^t_D. Let $(p,l)\in P_t$ and $t'\in\mathcal{O}$ being the (p,l)-extended tree from t. The RMO of t' is computed as $UpdateRMO:\mathbb{P}((D\times V_d\times V_d))\times P_t\to\mathbb{P}((D\times V_d\times V_d))$

$$UpdateRMO(RMO^t_D,(p,l)) = \bigcup_{d\in D}\ \bigcup_{e\in RMO_d(t)} Entries(e,(p,l))$$

4.3 Correctness

We prove the correctness of the Valid Tree Miner by showing that the apriori-principle holds for infix subtrees and the support definitions (cf. Definition 3.3). The correctness of the general RMO approach has been shown by Asai et al. [4]. To show that the apriori-principle holds for infix subtrees and the support definitions (cf. Definition 3.3), observe that for the infix subtree relation (cf. Definition 4.1), transitivity holds. Thus, for $t,t',t''\in\mathcal{O}$, if $t\subseteq t'$ and $t'\subseteq t''$ holds, it follows that $t\subseteq t''$; observe that one can construct an infix subtree mapping $\delta_{t\to t''}\in\Delta^{Infix}_{t\to t''}$ as $\delta_{t\to t''}=\delta_{t'\to t''}\circ\delta_{t\to t'}$ for $\delta_{t\to t'}\in\Delta^{Infix}_{t\to t'}$ and $\delta_{t'\to t''}\in\Delta^{Infix}_{t'\to t''}$. Using transitivity, Lemma 4.8 follows, which shows for infix subtrees that transaction support is admissible under the apriori-principle.

Lemma 4.8 (Apriori for Transaction Support). *Let $D\subseteq\mathcal{V}$ be a variant tree bank, $t{=}(V_t,E_t,L_t,\lambda_t,r_t,<_t)$, $t'{=}(V_{t'},E_{t'},L_{t'},\lambda_{t'},r_{t'},<_{t'})\in\mathcal{O}$ with $L_t,L_{t'}\subseteq L_D$ and $t'\subseteq t$. Let sup^w_{trans} be as defined in Definition 3.3 (1). Then $sup^w_{trans}(t',D)\geq sup^w_{trans}(t,D)$.*

Proof. By the *transitivity* of $\subseteq$, $\forall d\in D[t\subseteq d\Rightarrow t'\subseteq d]$. Thus, $\{d{\in}D\mid t'{\subseteq}d\}\supseteq\{d{\in}D\mid t{\subseteq}d\}$ holds. It follows that

$$\sum_{\substack{d\in D,\\ t'\subseteq d}} w(d)\geq\sum_{\substack{d\in D,\\ t\subseteq d}} w(d)$$

and, thus, $sup_{trans}(t',D)\geq sup_{trans}(t,D)$. □

Using the construction for $\delta_{t\to t''}\in\Delta^{Infix}_{t\to t''}$, we show the admissibility of root-occurrence support in Definition 3.3.

Lemma 4.9 (Apriori for Root-Occur. Support). *Let $D\subseteq\mathcal{V}$ be a variant tree bank and $t{=}(V_t,E_t,L_t,\lambda_t,r_t,<_t)$, $t'{=}(V_{t'},E_{t'},L_{t'},\lambda_{t'},r_{t'},<_{t'})\in\mathcal{O}$ with $L_t,L_{t'}\subseteq L_D$ and $t\subseteq t'$. Let sup^w_{root} be as defined in Definition 3.3 (2). Then $sup^w_{root}(t',D)\geq sup^w_{root}(t,D)$.*

Proof. For all $d{=}(V_d,E_d,L_d,\lambda_d,r_d,<_d,n^{trc}_d)\in D$ with $t\subseteq d$ and every $\delta_{t'\to d}\in\Delta^{Infix}_{t'\to d}$, we construct $\delta_{t\to d}{=}\delta_{t'\to d}\circ\delta_{t\to t'}\in\Delta^{Infix}_{t\to d}$. For all $\delta_{t'\to d}$, that maps $r_{t'}$ to a node $\delta_{t'\to d}(r_{t'})\in roots(t',d)$, we have a $\delta_{t\to d}\in\Delta^{Infix}_{t\to d}$. Then $\delta_{t\to d}$ maps r_t to a descendant of $\delta_{t'\to d}(r_{t'})$, thus, $\delta_{t'\to d}(r_{t'}){=}prt^h_d(\delta_{t\to d}(r_t))$ for $h\in\mathbb{N}_0$. Now observe that h by infix mappings preserving child-parent relations is fixed for every $\delta_{t'\to d}\in\Delta^{Infix}_{t'\to d}$. Furthermore, observe that a node's height h descendants are unique, as there cannot exist another node with the same height h descendants, as it would imply a *cycle*. Consequently, for every $v,v'\in roots(t,d)$ with $v{\neq}v'$ there exist $v'',v'''\in roots(t',d)$ with $v''{\neq}v'''$. Thus, $|roots(t',d)|{\geq}|roots(t,d)|$ and it follows that

$$\sum_{d\in D} w(d)\cdot|roots(t',d)|\geq\sum_{d\in D} w(d)\cdot|roots(t,d)|$$

Hence, $sup^w_{roots}(t',D)\geq sup^w_{roots}(t,D)$. □

4.4 Runtime Analysis

We derive a worst-case runtime bound of the Valid Tree Miner. Let $D\subseteq\mathcal{V}$ be a variant tree bank, then for a frequent infix subtree, in the worst case, we have $|L_D|{\cdot}s$ many (p,l)-extensions, where s is the size of the largest $d{\in}D$. To see this, consider that one can attach a node with every label along a right-most path of maximal length s. Then for the update of the RMOs observe the size of a tree's RMO is bound by $|V_D|$, the number of nodes of variant trees in D. Thus, for at most $|L_D|{\cdot}s$ many (p,l)-extensions we need to compute the extended trees RMO entries based on at most $|V_D|$ entries of the frequent infix subtree's RMO. Doing this for all $|F^{D,m}|$ frequent infix subtrees of a variant tree bank leads to a bound of $O(|F^{D,m}|{\cdot}|L_D|{\cdot}|V_D|{\cdot}s)$. This bound is quite pessimistic as the number of occurrences of a frequent subtree is expected to be much smaller than $|V_D|$. Furthermore, by pruning extensions and assuming a non-pathological structure of the variant trees, the number of (p, l)-extensions likely is much lower than $|L_D| \cdot s$. Finally, consider that the number of frequent subtrees of a tree bank grows exponentially when the minimum support threshold approaches zero [10]. Consequently, the computation of the set of frequent infix subtrees becomes ill-conceived as the output grows exponentially.

5 TOOL SUPPORT & APPLICATION SCENARIO

We implemented the proposed Valid Tree Miner in the open-source process mining tool Cortado [26], a tool for interactive process discovery and visual analytics. The Valid Tree Miner algorithm in Cortado is integrated as a tool that allows users to mine, explore, and filter the infixes of an event log's concurrency variants. The goal of infix mining in Cortado is to aid users in visual analytics and support process model discovery.

We present the tool based on the screenshot of the infix mining functionality in Cortado shown in Figure 9. Image section (1) shows the parameter selection, allowing a choice of support definition and the minimum support threshold used for mining. The size parameters allow for limiting the size of the mined infixes by stopping the mining early. Additional options allow for preprocessing of the variant tree bank before mining; for example, one can add artificial start and end activities to the concurrency variants. An infix containing a start or end activity corresponds to a prefix or suffix, respectively. Image section (2) shows further information about the infixes. The information includes the size, support, closeness, and a visualization of the infix. The infix subtree for visualization is transformed into a concurrency variant. If prefixes and suffixes are mined in addition, the dots preceding and following the pattern allow distinguishing prefix, suffix, and infix patterns. In addition, alignments between the infix and an existing process model, i.e., if the infix conforms to the process model, can be computed [23]. The infixes can be sorted in the table and filtered using the interface in image section (3) based on different properties, such as support, size, closeness, and maximality.[1]

Besides visual analytics, frequent infix mining is used in Cortado to support users during the incremental discovery of process models. Process discovery is concerned with learning a process model from an event log and is an essential step in process mining methodologies [31, 36]. Conventional discovery approaches [5] are fully automatic and thus return a process model given an event log as input. Therefore, the process model depends on the choice of the initial parameters and the event log's data quality. Furthermore, it does not allow human intervention during the discovery process to improve the model's quality. Consequently, interactive approaches have emerged to use human guidance during process discovery [25]. The approach in Cortado works by a user incrementally choosing variants and infixes that are not fitting the current model and automatically improving the process model to fit the chosen variants and infixes. The limiting factor of this approach is that the user needs to find interesting variants and infixes out of the large number of variants of real-world event logs. Frequent infix mining presents one avenue to discover interesting variants and infixes, showing common and shared behavior.

In a short example, we show the application scenario of infix mining for incremental process discovery. For the event log BPI Challenge 2012, which covers a loan-application process, we discover an initial process model covering the 25 most frequent variants. Approximately 70% of the event log's traces fit the resulting process model. To improve the model using Cortado, we mine frequent infixes of the event log with minimum trace-weighted transaction support of 5%, 600 out of 11,998 traces, and explore the frequent infixes not fitting the process model. We discover the not-fitting closed infix with a support of 1,122 traces in Figure 10 representing a subprocess of the loan application process not yet covered by the event log. Using the infix as input, Cortado improves the process model such that the model fits the infix. The modified model is capable of replaying $\approx 75\%$ of traces.

6 EVALUATION

We evaluate the proposed Valid Tree Miner regarding runtime, memory usage, and the number of mined frequent valid infix subtrees. Furthermore, we explore how the runtime and the results change using different support definitions.

6.1 Experimental Setup

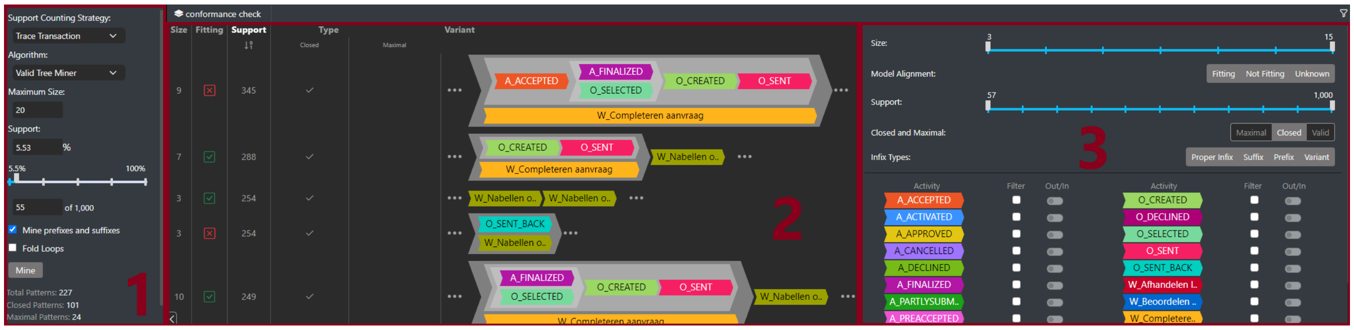
For the evaluation, we use the open access, real-world event logs: BPI Challenge 2012 [33], 2017 [34], 2020 [35], and the Sepsis Cases log [14]. To the best of the authors' knowledge, currently, no synthetic event log generator for concurrency variants that provides a known ground truth, i.e., the number of frequent infixes of the event log, exists. We focus on two event logs in the discussion. *BPI 2017* covers a loan application process and contains 1,202,267 events following 5,937 concurrency variants. *Sepsis Cases* covers sepsis treatment in a hospital with 15,214 events, many concurrent, following 694 concurrency variants. The variant trees of BPI 2017 have a median size of 25 out of the min-max interval $[8, 64]$, with a median height of 4 out of $[2, 6]$ and a median max degree - the number of children of a node - of 14 out of $[4, 51]$. For Sepsis Cases, the trees have a median size of 10 out of $[4, 256]$, a median height of 3 out of $[1, 3]$, and a median max degree of 11 out of $[3, 86]$. Plots regarding BPI 2012, a smaller log from the same process as BPI 2017, and BPI 2020, a small log from a travel cost reimbursements process with little concurrency, can be found online.[2]

We compare the proposed Valid Tree Miner, abbreviated as VTM, against a baseline algorithm based on the FREQT algorithm [4] that is state-of-the-art for frequent induced subtree mining. We implemented a version of the FREQT algorithm utilizing the adapted RMO update function presented in Definition 4.7 that mines frequent infix subtrees and uses the support definitions as introduced in Definition 3.3.

As parameters, we use relative minimum support computed based on the event log's number of traces/variants depending on the weighting used by the support function. As the results regarding transaction- versus root-occurrence support are nearly identical for

[1]For this, we have adopted the introduced closeness and maximality in Subsection 3.1.5 for the set of frequent *valid* infix subtrees.

[2]We provide further plots, especially for event logs BPI 2012 and 2020, at https://github.com/fit-daniel-schuster/Mining-Frequent-Infix-Patterns-from-Concurrency-Aware-Process-Execution-Variants


Support Counting Strategy: Trace Transaction

Algorithm: Valid Tree Miner

Maximum Size: 20

Support: 5.53 %

55 of 1,000

☑ Mine prefixes and suffixes

☐ Fold Loops

Total Patterns: 227
Closed Patterns: 101
Maximal Patterns: 24

Figure 9: Overview of infix mining in Cortado used on the BPI 2012 event log. (1) encompasses the parameter selection for the algorithm. In (2), the visualization and information for each infix are shown. Last, (3) shows the filter interface for the infixes.

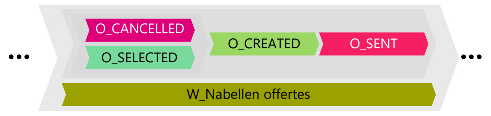


Figure 10: Frequent infix of the BPI 2012 event log, denoting the cancellation of a loan offer to a customer (*O_CANCELLED*) and the creation (*O_CREATED*) and sending (*O_SENT*) of a new offer during a customer call (*W_Nabellen offertes*).

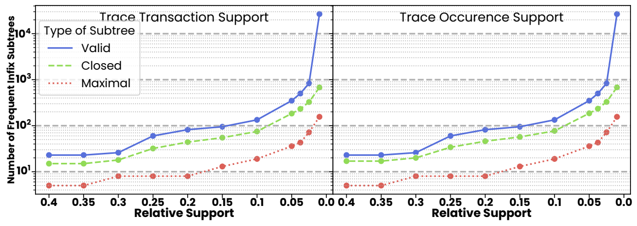
the tested event logs regarding the number of discovered patterns, the runtime, and memory usage, we mainly differentiate between trace- and variant-weighting during the discussion. The reason for the close-to-identical results is based on the observation that the transaction support of an infix subtree is always less or equal to its root-occurrence support, cf. Definition 3.3. Thus, only in the case of many distinct root occurrences, an infix subtree that is frequent for root-occurrence support but infrequent for transaction support is discovered, resulting in different sets of frequent infixes.

Due to the nature of the result set to grow exponentially, we stop after 5 minutes or a memory usage over 10 gigabytes. All experiments are conducted on a Windows 10 PC with 16 GB of physical memory and an i7-4790 CPU with 3.60 GHz. The mining algorithm runs multiple times per experiment to outbalance the impact that fluctuations in the system's performance have.

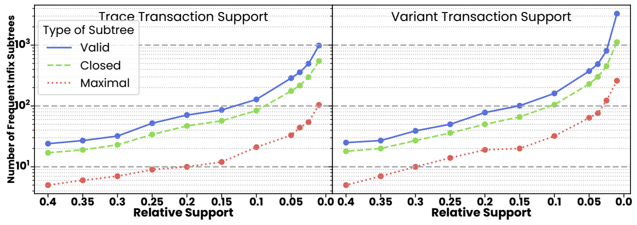
6.2 Results

This section presents the experiments' results, ordered by the number of frequent patterns, runtime, and memory consumption.

6.2.1 Number of Patterns. To practically show the correctness of the pruning strategies, we compared the size and equivalence of the sets of frequent valid infix subtrees mined by both FREQT and the VTM using canonical strings [12]. The resulting sets of frequent valid infix subtrees were *identical* for all tested parameterizations and event logs, as is expected for exact algorithms. Consequently, in the following, we present the number of frequent valid infix subtrees independent of the algorithm. Looking at Figure 11, one observes that the theoretical increase in frequent infixes as described in Subsection 4.4 does occur for a decreasing minimum support threshold. This increase is observed in Figure 11a and Figure 11b.



(a) Sepsis Cases event log



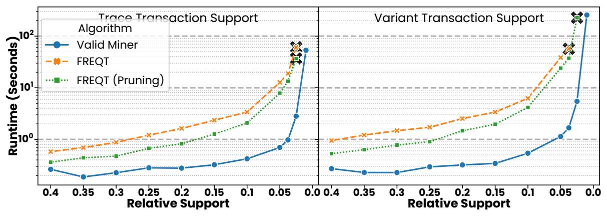
(b) BPI 2017 event log

Figure 11: Number of frequent valid infix subtrees for different support definitions and minimum support values for various event logs.

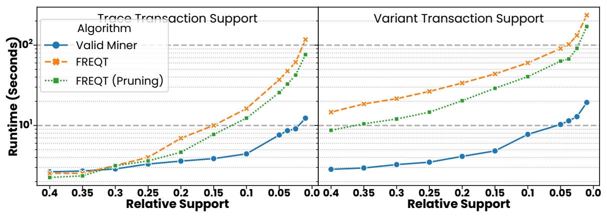
The number of valid closed and valid maximal infix subtrees[3] develops similarly, with an exception for low minimum support values for the Sepsis Cases event log, where high growth in the number of frequent valid but non-closed infix subtrees is observed, cf. Figure 11a. The latter is related to Sepsis Cases containing many concurrent activities. As infix subtrees allow for concurrent activities to be skipped, cf. Subsection 4.1, more frequent subpatterns of larger frequent infixes exist.

6.2.2 Runtime. The results in Figure 12 show that the proposed VTM can outperform the FREQT algorithm on the presented real-world event logs. The margin between the algorithm increases with a decreasing minimum support threshold. For the BPI 2017 event log in absolute numbers, using variant-weighted support for relative support of 1%, equivalent to 59 variants out of the 5,936 variants, the VTM mines the 3, 290 frequent valid infix subtrees in

[3]Closeness and maximality considering only frequent *valid* infix subtrees.

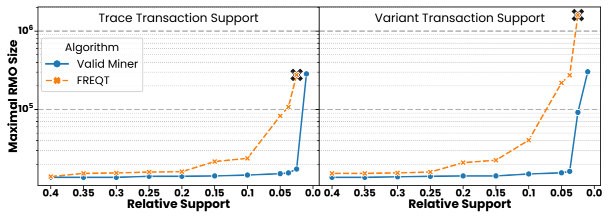


(a) Sepsis Cases event log

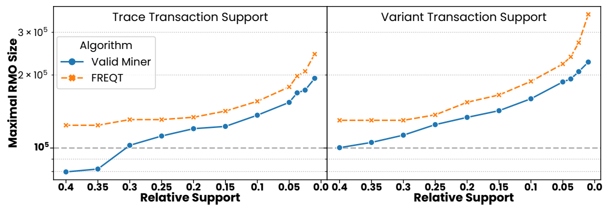


(b) BPI 2017 event log

Figure 12: Runtime comparison between algorithms for different relative support values and support definitions for different logs. Lower values indicate a faster runtime.



(a) Sepsis Cases event log



(b) BPI 2017 event log

Figure 13: Max RMO size comparison for different relative minimum support values and support definitions for different logs. Lower values indicate a lower memory footprint.

20 seconds. For the same task, the FREQT algorithm with pruning took 170 seconds, and without pruning took 235 seconds.

Looking at higher minimum support values, cf. Figure 12b, one observes that FREQT outperforms the VTM for trace-weighted support. The likely reason is that the number of frequent valid infix subtrees is small for high support, cf. Figure 11b. Thus, the higher initial cost of computing pruning sets of VTM, e.g., the frequent relation sets (cf. Subsection 4.2.1), are not paying off over the long run compared to the faster computable pruning sets used by FREQT (Pruning). For variant-weighted support, cf. Figure 12, the VTM outperforms FREQT directly. For Sepsis cases, cf. Figure 12a, one can observe the impact of the rapid growth in frequent valid infixes for low minimum support, cf. Subsection 6.2.1; the runtime of the VTM rapidly increases, while FREQT times out.

6.2.3 Memory. To measure memory usage, we track the size of the RMOs for each set of frequent subtrees of a given relative support. Observe that both FREQT and VTM traverse the rightmost enumeration tree in a breadth-first manner. Consequently, the RMOs for the trees of size k are only needed when computing the RMOs for the candidate subtree of size k+1. Thus after computing the RMOs for the candidate subtrees of size k+1, the RMOs of the frequent subtree of size k can be deleted. As such, to get an approximation of peak usage, i.e., the maximum number of entries in the RMOs that need to be maintained, we sum the size of the RMOs and take the maximum out of the sums for the different sizes of frequent subtrees. As both versions of the FREQT algorithm compute the identical frequent infix subtrees, we compare only the FREQT Algorithm against the VTM. In Figure 13, we observe that the VTM needs to maintain fewer RMO entries than the FREQT algorithm. The lower memory usage is caused by the VTM being able to prune invalid subtrees earlier instead of filtering them out after the mining. Furthermore, the blow-up, cf. Subsection 6.2.1, in the number of frequent valid subtrees for the Sepsis Cases event log causes a increase in the number of RMO entries.

7 CONCLUSION

This paper addresses the problem of frequent infix mining from concurrency variants. Infixes allow for visual analytics and other down-stream application on large event logs in process mining. This paper presented a novel algorithm for mining frequent infixes from concurrency variants. The presented *Valid Tree Miner* algorithm builds on the established field of frequent subtree mining [10] to discover a new type of frequent subtrees called valid infix subtrees, representing infixes of concurrency variants. Furthermore, we presented an example application scenario, i.e., incremental process discovery, in which frequent infixes, as considered in this paper, are used to learn a process model. We have fully implemented the proposed algorithm into an interactive process mining tool called Cortado that allows visual exploration of the mined frequent infixes. We evaluated the algorithm's efficiency in terms of runtime and memory usage. We compared it to a state-of-the-art algorithm for mining frequent induced subtrees on real-world event logs. Our results show runtime improvements of the proposed algorithm compared to the state-of-the-art.

Future research regarding frequent infix mining could explore other subtree definitions besides infix subtrees. The adherence to sequential completeness and the tree structure in infix subtrees causes behavioral patterns crossing multiple tree levels to be missed. *Embedded* subtrees [10, 38] present a suitable alternative for this task but are more computationally expensive to mine. Furthermore, additional applications of the infix patterns are to be considered. Foremost, infix patterns can be leveraged to cluster traces and variants, a common task in process mining [27]. Similarly, they can be used for prediction tasks in process mining [6, 9, 16].

REFERENCES

[1] Wil M. P. van der Aalst, Arya Adriansyah, Ana Karla Alves de Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter van den Brand, Ronald Brandtjen, Joos Buijs, et al. 2011. Process mining manifesto. In *International conference on business process management*. Springer, 169–194.

https://doi.org/10.1007/978-3-642-28108-2_19

[2] Rakesh Agrawal and Ramakrishnan. Srikant. 1995. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE Comput. Soc. Press, 3–14. https://doi.org/10.1109/ICDE.1995.380415

[3] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, Vol. 1215. Santiago, Chile, 487–499. https://doi.org/10.5555/645920.672836

[4] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. 2004. Efficient substructure discovery from large semi-structured data. *IEICE Transactions on Information and Systems* 87, 12 (2004), 2754–2763. https://doi.org/10.1137/1.9781611972726.10

[5] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. 2019. Automated Discovery of Process Models from Event Logs: Review and Benchmark. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (2019), 686–705. https://doi.org/10.1109/TKDE.2018.2841877

[6] Kristof Böhmer and Stefanie Rinderle-Ma. 2020. LoGo: combining local and global techniques for predictive business process monitoring. In *Advanced Information Systems Engineering: 32nd International Conference, CAiSE 2020, Grenoble, France, June 8–12, 2020, Proceedings 32*. Springer, Springer, Cham, 283–298. https://doi.org/10.1007/978-3-030-49435-3_18

[7] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. 2009. Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models.. In *Business Process Management Workshops*, Vol. 43. Springer, 170–181. https://doi.org/10.1007/978-3-642-12186-9_16

[8] Josep Carmona, Boudewijn F. van Dongen, Andreas Solti, and Matthias Weidlich. 2018. *Conformance Checking*. Springer. https://doi.org/10.1007/978-3-319-99414-7

[9] Michelangelo Ceci, Pasqua Fabiana Lanotte, Fabio Fumarola, Dario Pietro Cavallo, and Donato Malerba. 2014. Completion time and next activity prediction of processes using sequential pattern mining. In *Discovery Science: 17th International Conference, DS 2014, Bled, Slovenia, October 8-10, 2014. Proceedings 17*. Springer, Springer, Cham, 49–61. https://doi.org/10.1007/978-3-319-11812-3_5

[10] Yun Chi, Richard R. Muntz, Siegfried Nijssen, and Joost N. Kok. 2005. Frequent subtree mining - An overview. *Fundamenta Informaticae* 66, 1-2 (2005), 161–198. https://doi.org/10.5555/1227174.1227182

[11] Yun Chi, Yi Xia, Yirong Yang, and Richard R. Muntz. 2005. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering* 17, 2 (2005), 190–202. https://doi.org/10.1109/TKDE.2005.30

[12] Yun Chi, Yirong Yang, and Richard R. Muntz. 2005. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowledge and information systems* 8 (2005), 203–234. https://doi.org/10.1007/s10115-004-0180-7

[13] Remco Dijkman, Juntao Gao, Alifah Syamsiyah, Boudewijn F. van Dongen, Paul Grefen, and Arthur ter Hofstede. 2020. Enabling efficient process mining on large data sets: realizing an in-database process mining operator. *Distributed and Parallel Databases* 38, 1 (2020), 227–253. https://doi.org/10.1007/s10619-019-07270-1

[14] Felix Mannhardt. 2016. Sepsis Cases - Event Log. https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460

[15] Peter C Fishburn. 1970. Intransitive indifference with unequal indifference intervals. *Journal of Mathematical Psychology* 7, 1 (1970), 144–149. https://doi.org/10.1007/978-3-319-19069-3_19

[16] Philippe Fournier-Viger, Ted Gueniche, and Vincent S. Tseng. 2012. Using partially-ordered sequential rules to generate more accurate sequence prediction. In *Advanced Data Mining and Applications: 8th International Conference, ADMA 2012, Nanjing, China, December 15-18, 2012. Proceedings 8*. Springer, Springer, Berlin, Heidelberg, 431–442. https://doi.org/10.1007/978-3-642-35527-1_36

[17] Shohei Hido and Hiroyuki Kawano. 2005. AMIOT: induced ordered tree mining in tree-structured databases. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, IEEE, 8–17. https://doi.org/10.1109/ICDM.2005.20

[18] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review* 28, 1 (2013), 75–105. https://doi.org/10.1016/j.procs.2015.03.198

[19] Sander J. J. Leemans, Sebastiaan J. van Zelst, and Xixi Lu. 2023. Partial-order-based process mining: a survey and outlook. *Knowledge and Information Systems* 65, 1 (2023), 1–29. https://doi.org/10.1007/s10115-022-01777-3

[20] Jing Lu, Weiru Chen, Osei Adjei, and Malcolm Keech. 2008. Sequential patterns postprocessing for structural relation patterns mining. *International Journal of Data Warehousing and Mining (IJDWM)* 4, 3 (2008), 71–89. https://doi.org/10.4018/978-1-61520-969-9.ch049

[21] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. 1997. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery* 1, 3 (1997), 259–289. https://doi.org/10.1023/A:1009748302351

[22] Lars Reinkemeyer. 2020. *Process Mining in Action*. Springer. https://doi.org/10.1007/978-3-030-40172-6

[23] Daniel Schuster, Niklas Föcking, Sebastiaan J van Zelst, and Wil M. P. van der Aalst. 2022. Conformance Checking for Trace Fragments Using Infix and Postfix Alignments. In *International Conference on Cooperative Information Systems*. Springer, Springer, Cham, 299–310. https://doi.org/10.1007/978-3-031-17834-4_18

[24] Daniel Schuster, Lukas Schade, Sebastiaan J. van Zelst, and Wil M. P. van der Aalst. 2022. Visualizing Trace Variants from Partially Ordered Event Data. In *Process Mining Workshops*. LNBIP, Vol. 433. Springer, 34–46. https://doi.org/10.1007/978-3-030-98581-3_3

[25] Daniel Schuster, Sebastiaan J. van Zelst, and Wil M. P. van der Aalst. 2022. Utilizing domain knowledge in data-driven process discovery: A literature review. *Computers in Industry* 137 (2022), 103612. https://doi.org/10.1016/j.compind.2022.103612

[26] Daniel Schuster, Sebastiaan J. van Zelst, and Wil M. P. van der Aalst. 2023. Cortado: A dedicated process mining tool for interactive process discovery. *SoftwareX* 22 (2023), 101373. https://doi.org/10.1016/j.softx.2023.101373

[27] Minseok Song, Christian W Günther, and Wil M. P. van der Aalst. 2008. Trace clustering in process mining. In *International conference on business process management*. Springer, Springer, Berlin, Heidelberg, 109–120. https://doi.org/10.1007/978-3-642-12186-9_16

[28] Henry Tan, Tharam S. Dillon, Fedja Hadzic, Elizabeth Chang, and Ling Feng. 2006. IMB3-Miner: Mining Induced/Embedded subtrees by constraining the level of embedding. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, Springer, Berlin, Heidelberg, 450–461. https://doi.org/10.1007/11731139_52

[29] Shirish Tatikonda, Srinivasan Parthasarathy, and Tahsin Kurc. 2006. TRIPS and TIDES: new algorithms for tree mining. In *Proceedings of the 15th ACM international conference on Information and knowledge management*. ACM, Association for Computing Machinery, 455–464. https://doi.org/10.1145/1183614.1183680

[30] Niek Tax, Natalia Sidorova, Reinder Haakma, and Wil M. P. van der Aalst. 2016. Mining local process models. *Journal of Innovation in Digital Ecosystems* 3, 2 (2016), 183–196. https://doi.org/10.1016/j.jides.2016.11.001

[31] Wil M. P. van der Aalst. 2016. *Process Mining: Data Science in Action*. Springer. https://doi.org/10.1007/978-3-662-49851-4

[32] Wil M. P. van der Aalst. 2020. On the Pareto Principle in Process Mining, Task Mining, and Robotic Process Automation.. In *Proceedings of the 9th International Conference on Data Science, Technology and Applications - DATA*. INSTICC, SciTePress, 5–12. https://doi.org/10.5220/0009979200050012

[33] Boudewijn F. van Dongen. 2012. BPI Challenge 2012 - Event Log. https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

[34] Boudewijn F. van Dongen. 2017. BPI Challenge 2017 - Event Log. https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b

[35] Boudewijn F. van Dongen. 2020. BPI Challenge 2020 - Event Log. https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51

[36] Maikel L. van Eck, Xixi Lu, Sander J. J. Leemans, and Wil M. P. van der Aalst. 2015. PM^2: A Process Mining Project Methodology. In *Advanced Information Systems Engineering*. LNCS, Vol. 9097. Springer, 297–313. https://doi.org/10.1007/978-3-319-19069-3_19

[37] Yongqiao Xiao and J-F Yao. 2003. Efficient data mining for maximal frequent subtrees. In *Third IEEE International Conference on Data Mining*. IEEE, IEEE, 379–386. https://doi.org/10.1109/ICDM.2003.1250943

[38] Mohammed J. Zaki. 2002. Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. IEEE, 71–80. https://doi.org/10.1109/TKDE.2005.125

[39] Fareed Zandkarimi, Jana-Rebecca Rehse, Pouya Soudmand, and Hartmut Hoehle. 2020. A generic framework for trace clustering in process mining. In *2020 2nd International Conference on Process Mining (ICPM)*. IEEE, IEEE, Padova, 177–184. https://doi.org/10.1109/ICPM49681.2020.00034