Anomaly Detection in Processes Represented as a Graph

Authors:

Jesse Waite, Washington State University (jessewaite86@gmail.com)

Lawrence Holder, PhD., Washington State University (holder@wsu.edu)

NOTES: ‘anomaly-detection’ v anomaly detection. Also ‘real-world’ v real world. Grep all hyphens.

**Abstract**

Learning the structure of stochastic, noisy environments remains an important area of process and graph mining. This work presents an unsupervised, threshold-based method of process mining and anomaly detection using the SUBDUE graph-compression method and the Inductive Miner algorithm. The method generates a dendrogram of the compressing structural features of a workflow log, a taxonomical representation by which further analysis can be performed. Via this dendrogram, anomaly detection was performed by applying a Bayesian threshold to detect unusual traces and their components, and was evaluated on synthetic data over a range of parameter values and model types. Experimental results show 96% accuracy on an anomaly detection task for reasonable data and algorithmic parameters, reliable performance metrics across a range of these parameters, and competitive performance against a previously studied anomaly detection method known as the Sampling Algorithm. A real-world demonstration is also provided for software-testing log data generated from a software unit-test suite of function calls of the NASA Crew Exploration Vehicle (CEV) mission platform, with results identifying anomalous components of its design. The results are promising, and may also inform future approaches based on machine learning.

**1. Introduction**

As described in [1] by Dumas *(et al.)*, a process aware information system (PAIS) is, “a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models” (p. 5). This formalizes operational management systems as systems centralizing awareness of process data as well as the ability to prescribe tasks and activities via process models. However, many institutions instead rely on interwoven, non-interoperable systems to monitor and control processes, making the formal requirements of a centralized PAIS infeasible or intrusive. This work focuses on contexts where a PAIS is instead an abstraction integrating process data derived from disparate systems and data sources to generate a process-oriented view of the underlying processes. This is amenable to realistic processes which are embedded within a non-stationary framework of changing people, tools, resources, often in the absence of prescribed process models.

The ability to mine and analyze normative process patterns in such partially structured contexts is critical for mining regular activity and for detecting anomalies. The latter requires prior normative activity patterns; thus, anomaly detection and normative pattern mining are complementary tasks. This work presents such a method for mining process patterns from workflow logs which also possesses useful anomaly detection properties. The Inductive Miner [2] was used to construct a graphical process model from log data, and then the SUBDUE [3] graph compression method was used to generate a hierarchical dendrogram of normative log patterns executed on this model. Using this representation, anomalies and significant process features can be discovered in post-processing.

In the remainder of this paper, we cover preliminaries of terminology, process mining, graph compression and its application to process data, followed by the formal problem definition of process-based anomaly detection, and the proposed method of using graphical compression for anomaly detection. In the materials and methods section, we articulate our synthetic data generation method and several experiments intended to evaluate properties of the approach on controlled synthetic data, followed by a comparison with an existing anomaly detection method known as the Sampling Algorithm, and a real data evaluation. We then provide results for these experiments in the result section, followed by conclusions and future work.

**2. Preliminaries**

*2.1 Terminology*

From the control-flow perspective, common process mining terms can be framed in a graph theoretic manner:

* Process model: A formal semantic model of process behavior, defined later in greater detail. A canonical Petri net [4] example shown at left in figure 3.1, with places (labeled vertices) separated by transitions (black rectangles). Its simpler control-flow counterpart is shown at right and is used throughout this work to describe the structural patterns of traces. The control flow graph is derived by removing the transitions of the Petri net and directly connecting activities (places) directly. Formally, the control flow graph is defined such that if there is any path between two places via a transition on the original Petri net model, then there is an edge in the control flow graph. This removes some semantic transition information from the model but is sufficient to model structural regularity, and thereby detect anomalies.



Figure 3.1: A Petri net process model and its simpler control flow counterpart.

* Workflow trace: A single execution of a process as a partially-ordered sequence of activities, following any valid path from a START/BEGIN to an END/EXIT marking on a process model.
* Workflow log: A set of workflow traces in some canonical form, such as eXtensible Event Stream (XES) [5]. Synonymous with ‘trace log’, ‘process log’, or simply ‘log’.
* Process miner: An algorithm constructing a process model from a workflow log, with parameterized criteria for model complexity, specificity, and generality. Specificity favors restrictive models including only or even less than the behavior described by the workflow log, whereas generality favors larger models describing all traces and possibly additional behavior.
* Partial-order property: A property of workflow traces whereby activities may be randomly-ordered with respect to parallel sub-processes. ‘ABCD’ and ‘ACBD’ could be workflow traces from the model in figure 3.1, where ‘C’ and ‘B’ represent parallel sub-processes, or may recursively embody further parallel sub-processes, and ‘A’ always occurs before ‘D’.
* Process grammar: Recursively-defined constructs for common process patterns. An AND-SPLIT is a set of edges branching from a single vertex and traversing activities in parallel before synchronizing at some later activity. Other basic constructs include OR-SPLIT, XOR-SPLIT, LOOP, and JOIN [6].
* Replayability: The ability for a process model *M* to generate a partially-ordered trace *t*. For instance, the string ‘ABCD’ would be replayable on the model in figure 3.1, but ‘ACDB’ would not.
* Spaghetti model: A workflow defined by highly diverse, informal, and disorderly behavior, typically containing many scattered, repetitive events. These represent unstructured business processes, in contrast to orderly “lasagna” processes with prescriptive, stratified behavior [7].
* Inductive miner: A process mining algorithm capable of generating the most general and all-inclusive process model of the traces in some log. In this work, this model is used to convert a workflow log of traces into a collection of subgraphs as structured input for SUBDUE.
* SUBDUE: Short for “SUBstructure Discovery Using Examples”, this method implements a subgraph beam search over a graph collection and returns the top-k most compressing sub-graphs, according to a minimum-description length (MDL) heuristic [3].
* GBAD: Acronym for “graph-based anomaly detection”, this method internally calls SUBDUE, then implements methods for detecting anomalies occurring in the context of discovered patterns (Holder and Eberle, 2009).
* Dendrogram: A graphical representation of the non-disjoint compressing features (subgraphs) of some graph *G*, with respect to a set of traces. Each vertex represents a compressing subgraph of *G*, and edges between features indicate familial relationships. An explicit example dendrogram is given later and further refined.

An overview of process mining terms and methods can be found in [1] and [8].

*2.2 Processes and Representations*

Process modeling encompasses many formal languages for specifying representations of process models and their properties. These representations may encompass varying levels of complexity to describe processes or to verify certain properties such as verifiability, reachability, and completeness, which depend on the choice of representation [9]. The canonical example is the Petri net, whose properties are well-studied but remain a subject of active research due to their expressive power [4]. A formal Petri net definition is given in table 1.1. Under this construction, a process is represented graphically by a set of vertex “places” , representing activities, resources, or other entities of the process. Places are separated by “transitions” , which semantically constrain the nature of transitions between places. Edges connect places and transitions to give the model its structure, where transitions connect only to places, and places connect only to transitions, . The model is given an initial and final marking, tokens placed on one or more places to denote the initial and final state of the process.

|  |  |
| --- | --- |
| Table 1.1: Simple Petri net definition | |
|  | A set of places, where *i* = |*P*| |
|  | A set of transitions, where *j* = |*T*| |
|  | A set of edges from places to transitions and transitions to places |
|  | Initial marking, the initial set of places of the process’ execution |
|  | Final marking, the set of terminal places of the process’ execution |

An example Petri net model is given in figure 2.1 for a simplified automotive manufacturing process. A valid walk on this model is constrained by the initial marking , final marking , and the transitions between places (activities) comprising the activities of the process. A walk is given by a token movement scheme whereby no transition may “fire” until sufficient tokens have reached it, which for this simple example will require completion of all places incident to a transition. Many other semantics are possible, such as colored Petri nets, where tokens are not single items but are instead functions evaluated at places [10]. The structural semantics of Petri nets permit a variety of formal analyses, such as verifying completeness, reachability, liveness, deadlocks, and satisfiability properties, which are critical to the verification of distributed systems. An overview is provided by [11].

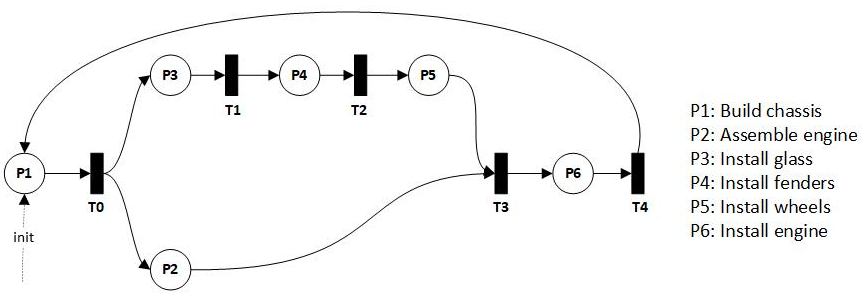


Figure: A Petri net model for a simplified automotive manufacturing process.

As generic state transition systems, Petri nets are amenable to a variety of derived graphical representations with practical properties, an overview of which is given by [12]. One derivation is the coverability graph of a Petri net, a landmark representation in computation theory by Karp and Miller [13] that consists of the graph of all reachable markings that can be generated by moving tokens along the arcs of a Petri net. Construction begins by placing tokens on each of the places in the initial marking set. Represented as a binary vector , the initial marking for the above Petri net is . Tokens then move forward between places and transitions, with each unique possible marking spawning a new child vertex. Hence the only next marking is given by traversing T0 to places , giving marking . From this marking, the token on P1 can reach T3, where it must wait until the other arrow pointing from P4 to T3 becomes active.

The final coverability graph is given in figure 2.2. Note that despite the parallel paths in the model, the coverability graph is simply linear. One also sees from the coverability graph that the model is infinite since the process can repeat from *T4*, depending on how the final marking is defined. This edge could characterize serial process flow, whereby new car production cannot begin until the previous cycle completes, an example of “cycle time” often used in computer processor design and business production models. The terminal conditions of this specific process were intentionally left ambiguous since the goal was only to demonstrate how the analysis of processes depends on transformative graphical representations to determine their various properties. The example demonstrates how the solution to many process problems results from the expressiveness of their representation in a new graphical space. Whereas problems in many other fields result in a search over some space of solutions, process mining often yields a search over the space of how the problem can be represented, such that it is trivially solved.

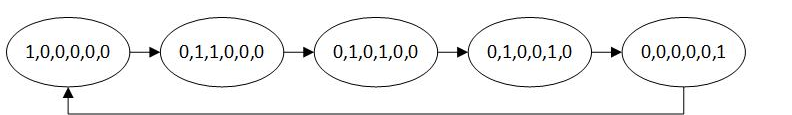


Figure 2.2: Coverability graph of the Petri net in figure 2.1.

Ordinarily, coverability graphs encompass greater complexity, and can be used for decidability problems, such as reachability, loop detection (as in figure 2.2), and other formal properties. The applicability of reachability decisions to process engineering and analysis demonstrates the direct connection between the design of critical systems and many intractable problems in computational theory. Extensive examples of such decidability problems and algorithms are given by [14].

## 

## *2.3 Anomaly Detection in Process Data*

Since the coverability graph of a Petri net model characterizes valid activity transitions through the model, the model-consistency of any trace (a single execution of a process) can be verified by checking if the trace represents a valid walk on the coverability graph: beginning from some initial marking, consuming all activities, and terminating at a final marking state. This form of design analysis can even be done without process data to answer critical *a priori* design queries, such as “Is system state *X* reachable from state *Y*?” or likewise to evaluate path-cost and boundedness queries. Previous anomaly detection methods use token counting metrics to determine the consistency between traces and models.

However, traces generally encompass significant noise and choice (branching) behavior, making such consistency-checking procedures prone to false positives, and they are often only appropriate when the process model is formally prescribed and available for evaluation. As mentioned in the introduction, however, processes often either do not obey a prescribed process model or do not possess one, such that the true model must be treated as “unknown” and estimated from data using process mining algorithms. In these cases, distinguishing anomalies, noise, and regular behavior is much less trivial since each of these categories encompass their own statistical (rather than deterministic) distribution, and these distributions rest in a high-dimensional feature space of graphical structure.

This work instead focuses on graphical approaches to anomaly detection using graph compression. Graphical compression methods lend themselves directly to the analysis of process data, and subsequently the detection of anomalies, by characterizing process data as a distribution over graphical structures. These methods were used in this work since they can extract significant, and hence normative, patterns of the graphical representation of a process with respect to the distribution of the trace data from which they were derived. However, graphical representations of anomalies must first be characterized.

In contrast to other anomaly detection applications, graphical anomaly detection encompasses the high-dimensional structural features of graphs. A comparison is given in figure 2.3, and a similar example can be found in [15] (p. 3, figure 1), as the comparison is universal. The left plot demonstrates a typical statistical setting in a simple 2-dimensional space, for a contrived anomaly detection setting of data generated from two 2-dimensional Gaussian distributions, where the black ellipses represent outlier classification boundaries, hence any point outside these boundaries would be flagged as a potential anomaly. The right plot shows a graphical representation of email communities within the Enron email dataset [16], generated by the authors using the igraph library [17]. There are many approaches for detecting outliers in this static graphical setting: disconnected/isolated components, vertex centrality metrics, community clustering metrics, and more. The distinction is that the pointwise setting and entails deriving continuous decision surfaces such as the ellipses, whereas graphical anomaly detection encompasses high-dimensional graphical properties.

|  |  |
| --- | --- |
|  |  |

Figure: Pointwise and graphical anomaly detection settings. At left, a contrived pointwise setting. At right, Enron email data communities detected and plotted using the igraph library.

In contrast to the static graphical setting, this work is concerned with structural anomalies in dynamic process data. The input is both a process dataset *D* and its corresponding mined process model *M’* encompassing all possible graphical structure (edges) in *D*. *M’* represents interlinked activities like the email graph above, however, the concern is not strictly with the stationary structural anomalies of *M’*, but rather with anomalies which are detected with respect to both *M’* and the regular, compressing structural features in *D*.

This model is appropriate when one can assume *D* contains regular structural properties, which is usually satisfied for processes since they exhibit highly regular, higher-order structural patterns. By contrast, an email network like in figure 2.3 would be less appropriate, since *D* in that setting would consist of erratic point-to-point communications with only (or mostly) first-order regularity, and few higher-order recurring substructures. This highlights a continuum of real-world graphical data settings. At one extreme are datasets with only first-order regularity, such as email networks, Markov models, and particle models. At the opposite extreme are datasets for which higher-order regularity is a constraining property, such as manufacturing processes or organic molecular structures.

For example, consider that the email graph in figure 2.3 instead represented an e-commerce activity model, since it provides an accessible contrast of structured processes. Further, our interest is restricted to a distinct subset of colored vertices, representing a sub-process as shown in figure 2.4. The mined process model is shown at top right, where transitions have been removed and activities are directly linked. This representation erases the semantic constraints of a formal process model but sufficiently captures recurring substructures and is used throughout this work. The hypothetical input traces from which this model was derived are shown at left, and its recurring activity structures S\* are shown to the right in red and blue. These represent different activity pathways, where the red path may represent optional process behavior, such as a customer applying a third-party discount during checkout.

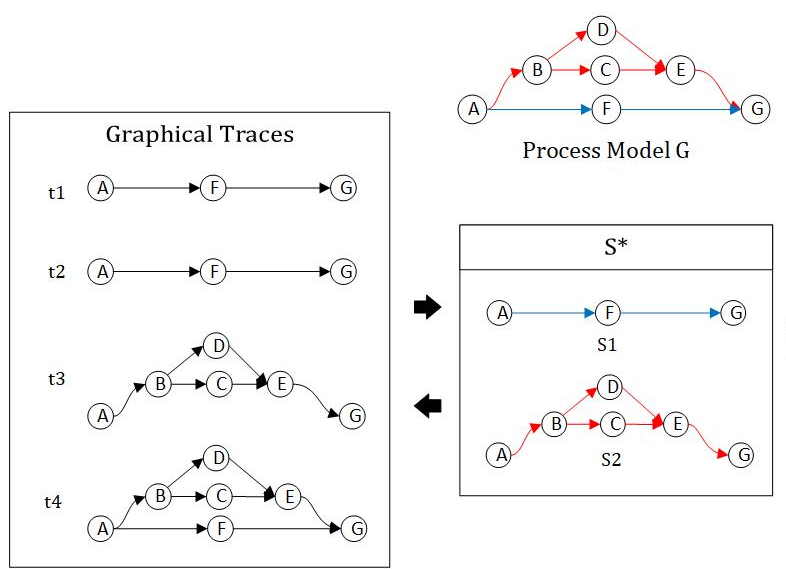


Figure 24: A graphical substructure representation of process data.

In the process setting, edges in the graph are assumed to encompass relations between the activities or resources of a process, but further, that the graphically-represented data also contains such structural regularity of arbitrarily higher-order. One expects that this data contains repeated substructures beyond simple first-order Markovian (edge) statistics or static structural features. These represent sub-tasks within a process, such as the customer checkout subprocesses of authentication, payment, and inventory transactions.

In this manner, process data is expected to contain regular structure beyond the static graphical setting depicted in figure 2.3. Rather than evaluating the dataset *D* as simply a distribution of edges and vertices and their local properties, it is assumed that *D* describes a distribution of higher-order substructures, which are overlapping components of *M’*. This property can be exploited to learn more complex patterns of the underlying process than first-order models or network-scientific approaches based on local features such as vertex or edge properties.

The remaining question is how to discover these structures efficiently. Graphical compression methods provide a straightforward method of extracting normative patterns and thereby detecting anomalies. The chosen method was SUBDUE [3], an acronym for “SUBstructure Discovery Using Examples”, which is a search-based method for compressing a graphical dataset. Using the minimum description length (MDL) compression criterion, SUBDUE maintains a *b­*-length beam of the most compressing substructures in a graphical dataset, iteratively expanding these substructures in all possible ways, sorting them via the compression criterion, truncating the beam, and continuing for *k* iterations or until no further compression is possible. The method returns the sorted beam containing the most compressing substructures found upon termination. SUBDUE can internally compress the dataset, or these substructures may be used as representative graphical features of the original input space, as in this work.

## *2.4 Previous Work*

Process anomaly detection has focused primarily on the mining process itself and on trace-scoring schemes. Scoring schemes were detailed by Bezerra [18], by which work traces were replayed on a discovered model, assigned a numeric fitness score, and anomalies were flagged based on a discriminative threshold. A later work [18] examined anomaly detection using several threshold-based approaches within the process mining algorithm itself. Bezerra decomposed this family of process-based anomaly detection into three groups: score based, iterative, and sampling. The approach in this work does not fit into these categories since it is compression/feature-based: a generic process model is mined, graphical features detected, and anomalies are detected and reported in post-processing. Likewise, whereas previous works focused on individual traces, the feature-based approach provides structural insights into normative patterns and anomalous features. This work replicates Bezerra’s data generation scheme, but otherwise adds a new method to this previous work.

SUBDUE was previously used for knowledge representation systems [20], and more recently in security applications for intrusion detection [21]. Using SUBDUE as a process mining tool was successfully performed by [22] and more recently by [23], whose results demonstrated the method’s utility for “spaghetti processes” describing more realistic institutional processes. An extensive overview of graphical compression and anomaly detection is provided by [15]. The shared insight of these works is that by viewing process data as a distribution over structures, one can derive unsupervised approaches of process analyses with general application to process mining and task-learning.

## 

## *2.5 Process Anomaly Detection Problem Definition*

In this work, the input is a log consisting of traces generated from some unknown process model *,* a graphical process miner , and a graph-compression method . The formal problem is to mine a graphical process model by which to convert into a collection of graphs via (a preliminary step), and then to iteratively mine the normative patterns of using s. The patterns can then be used for post-processing tasks, such as anomaly detection: given , identify and return anomalous traces .

**Compression-Based Process Anomaly Detection Problem Definition**

**Input** *mine*: A process mining algorithm

*L*: A trace log from some process

*compress:* A graph compression method, e.g. SUBDUE

*detect:* An anomaly detection method

**Output** : The set of normative graphical patterns

: A set of anomalous traces in L

1. = *mine*(*L*) #mine the graphical process model
2. traceGraphs = convert(, *L*) #convert the log traces to activity graphs, via
3. = *compress*(traceGraphs)
4. = *detect*(, *L*)
5. return ,

Outputting with reflects that in many cases an anomaly detection method can provide normative pattern information along with any anomalies it discovers. Akoglu *(et al*.) [15] refer to the application of anomaly detection for normative pattern extraction as “data cleaning”, whereby pattern information could be used for model refinement.

## *2.6 Evaluation Metrics*

Using labeled data in the form of traces with anomalous/non-anomalous class labels, an anomaly classifier’s performance can be evaluated using standard binary classification performance metrics: accuracy, precision, recall, and -score. Alternatively, one could evaluate a solution in terms of to determine how well a method compresses a log according to a compression-evaluation function. This is important evaluating structural learning approaches in terms of the compactness of their learned representations but is outside the scope of this work.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 3.1: Binary classifier outcomes for anomaly detection | | | |
| ***C1= anomaly (+)***  ***C2= normal (-)*** | | **Actual Class** | |
| C1 | C2 |
| Predicted Class | C1 | True Positive (TP) | Type I Error (FP) |
| C2 | Type II Error (FN) | True Negative (TN) |

Accuracy, precision, recall, and -score provide adequate baseline measures for anomaly detection by framing it as a binary classification task of classifying traces per two classes, C1 for the “anomalous” positive class of traces and C2 for the negative “normal” class of traces. Due to the common application of anomaly detection for security or critical system applications, an additional evaluation criterion may assign costs to the matrix of possible outcomes for a binary classifier. Letting denote the distribution of errors for a given classifier given the table above, the positive or negative cost per each outcome, the cost function is given by the expected value of the classifier:

The importance of a cost measure is that an anomaly detection method can appear to perform well under canonical binary classification metrics due to the low prior probability of anomalies (C1) in the data. Since anomalies are infrequent, binary classification evaluation metrics can suggest overly optimistic performance for a classifier which often fails to find anomalies, but nearly always classifies traces as normal (C2). In the worst case, consider evaluating a classifier which simply classifies every trace as “normal” on data for which the prior probability of an anomalous trace is 0.001, or 1 in 1,000. On average, this classifier will possess accuracy of 0.999 (99.9%), despite being senseless.

Precision, recall, and f1-measure remedy this defect by measuring the true-positive class (anomalies) explicitly. However, defining and evaluating the cost function is the most important anomaly detection evaluation method for real applications, such as for critical systems. For instance, in a customer service center process the type II misclassification of an anomaly may be a low-cost occurrence (the annoyance and loss of a customer), whereas for an aerospace mission system, the cost of misclassification could be the cost of the entire mission (the loss of a spacecraft). In this work, accuracy, precision, recall, and f1-measure were used since they provide baseline information for reproducibility and results comparison, whereas cost evaluation is application specific.

**3. Proposed Method**

Our method for anomaly detection decomposes to four tasks: mining a model, converting a trace log to a collection of activity subgraphs via this model, compiling descriptive normative graphical patterns of the log with respect to this model as a dendrogram via SUBDUE, and lastly detecting outliers and anomalous behavior. The data-flow of the approach is shown in figure 4.1.

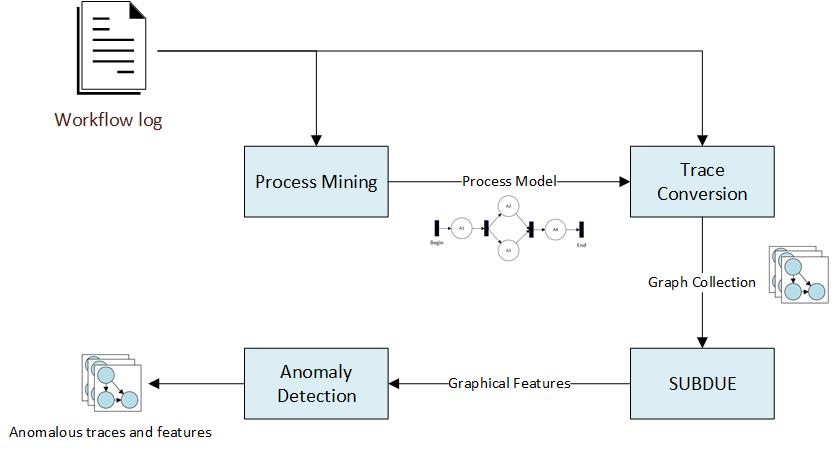


Figure 4.1: Data-flow model of the approach.

The Inductive Miner [2] was suitable for the mining task since it can provide the most general graphical process model described by a log. This is a property of our use of the Inductive Miner and not of the miner itself, which provides built-in refinements. The third and fourth tasks discover the patterns and features more precisely relevant to the log. Decoupling the mining and feature extraction steps allows for swapping the component algorithms, offering greater tuning for noisy or poorly structured data. This facilitates fitting the method to more realistic and informal “spaghetti” model scenarios in which processes are ad hoc and highly unstructured, such as enterprises, communication networks, software system executions, or natural processes.

## *3.1 Using Graph Compression to Discover Patterns and Cluster Traces*

Prior work showed strong anomaly detection results when running SUBDUE iteratively on a set of graphs [21], with instances of the most compressing subgraph replaced with a single meta-vertex at each iteration. At the end, the authors obtained a hierarchical description of the input, by which they modeled their anomaly detection scheme. Previous works by Jonyer, Holder, and Cook [24, 25] also used SUBDUE for hierarchical graphical clustering, applying the same pattern of iteratively replacing compressing substructures with single-vertex prototypes to generate a hierarchical graphical decomposition of the data.

A similar approach using GBAD [26] was tested, where workflow traces were iteratively recompressed, and the three anomaly detection methods of GBAD were used to detect anomalies at each iteration. But the anomaly detection method suffered a high false positive rate, which was ultimately due to iterative recompression: the most highly compressing subgraph was often only a small alteration (vertex substitution, deletion, or insertion) to the previous iteration. As a result, GBAD’s anomaly detection methods were repetitively applied to very similar regions of the input space. Hence, the search space was highly redundant, repeatedly analyzing highly similar graphical features, progressing very slowly toward the outlying model regions where compressing structure decays and where anomalies lie.

The remedy was simply to delete all instances of the most-compressing subgraph from the traces at each iteration. This forces SUBDUE to evaluate new regions at each iteration, and thus to discover dissimilar graphical features. Compressing substructures are compressed away in order of decreasing information, as shown in figure 4.2.

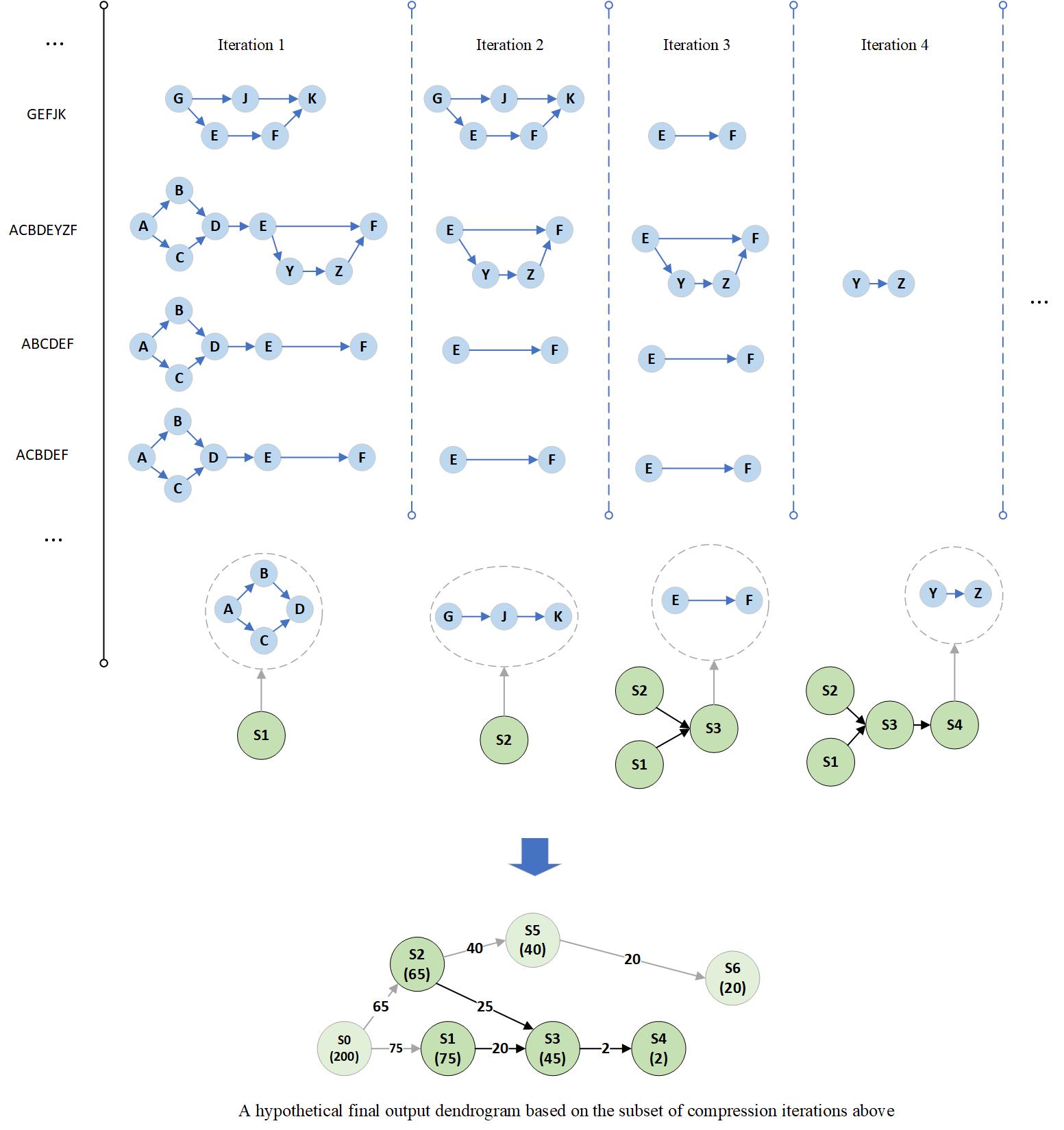


Figure 4.2: Constructing a dendrogram of graphical features from graphical process data. At bottom, the final hypothetical dendrogram, a directed acyclic graph composed of compressed components.

In figure 4.2, iterations are portrayed from left to right, with the original activity traces shown leftmost as strings, and their graphical counterparts in blue. Proceeding from left to right, the log shrinks as compressing substructures are discovered and deleted, hence each vertical column represents the state of the log at each iteration. In the first iteration shown, S1 is discovered (in green), a graph of four vertices. S1 is then deleted from all traces, notably along with any incident or outgoing edges. Next, S2, a substructure of three vertices, is discovered and deleted. Then S3, a substructure of two vertices is discovered and deleted, and its substructure vertex in green is linked to both S1 and S2, since these are the immediate substructure predecessors of the traces compressed by S3; in contrast, S1 and S2 were not linked, since their traces shared no previous substructures in the iterations portrayed. In this manner, the entire log is compressed away, building a hierarchical dendrogram of substructures as a directed acyclic graph, as shown in green at the bottom of figure 4.2 with hypothetical substructure frequency labels.

Formally, the dendrogram is a directed acyclic graph of weighted vertices and edges. Each vertex represents a compressed graphical substructure, each with trace frequency and a set of trace ids (e.g., ) it partially compressed, hence . Edges represent immediate substructure ancestry via trace ids. Consider two substructures, and , where was compressed prior to , thus . Substructures and are ancestors if , indicating that compresses one or more trace ids also compressed by in a previous iteration. Therefore, the Boolean valued substructure ancestry function is defined by:

But only immediate ancestors are linked, such that an edge only exists if is the most recent ancestor satisfying :

The righthand clause denotes that is the most immediate ancestor satisfying . In code, this amounts to a simple look up at each iteration: for each trace id compressed in the current iteration, one looks up in the trace ids of previously compressed substructures and returns the most recent substructure containing .

Finally, dendrogram edge weights simply equal the number of trace ids shared between such substructures:

Since these equations may be unclear, an implementation example is provided via Table 4.1. Assume that compression has been applied iteratively to a log, generating the hypothetical record table shown.

|  |  |  |
| --- | --- | --- |
| **Table 4.1: Substructure relation example** | | |
| Iteration | Substructure | Trace-ids |
| … | … | … |
|  |  | {4,5,**6**} |
|  |  | {1,2,3,4,5,**6**} |
|  |  | {**6**} |
| … | … | … |

The substructures are listed in the center column in the order they were compressed, and each substructure is accompanied by the traces ids from which it was deleted. Here, and both evaluate to true, since and both contain trace id 6, in bold. However, is only true for since it is the most recent. Therefore, would be linked to with , and and would not be linked.

Incorporating the definitions for and , the algorithm for appending a substructure to the dendrogram is defined as:

**AddSubstructure Pseudocode Definition**

**Input** *dendrogram*: The current dendrogram of connected substructures

*newSubstructure*: A compressing substructure containing its associated trace id’s

**Output** *dendrogram*: The updated dendrogram, with *substructure* added to it

//add the new substructure as a vertex

1. *dendrogram.vertices* = *dendrogram.vertices* newSubstucture
2. j = *newSubstructure.index*

//add edges as described previously

1. for existingSubstructure in *dendrogram.substructures*:
2. i = existingSubstructure.index
3. if :
4. edge = Edge(i,j)
5. edge.weight =
6. *dendrogram.edges* = *dendrogram.edges* edge
7. return *dendrogram*

The relationship between substructures via this construction demonstrates a heuristic dependency and a globally decreasing substructure information “value”. From the trace perspective, this possesses a local property such that the ordered removal of substructures relates them in some way, obviously because those structures represent component tasks that were executed within that trace: a significant subtask is removed first, followed by a less relevant task, and finally by anomalous and outlier tasks. There is a local dependency between substructures, which relates through the traces via their id’s.

The approach loosely resembles data dimensionality reduction, in which data is compressed via a set of vectors ordered by decreasing information. Except vectors are replaced by graphical substructures, forming a lossy hierarchical derivation of process substructures as a dendrogram. Lossy, since any edges incident on a compressing substructure are deleted with the substructure and in general cannot be deterministically reconstructed from the dendrogram. Otherwise the dendrogram comprises the entire behavior of the log, with the ancestral components reflecting its most relevant and compressing graphical features. This is amenable to anomaly detection since the less compressing a feature is, the greater its deviation from normative patterns and normal overall behavior, and hence it will lie deeper in the dendrogram and with sharply lower frequency than its parents.

This gives the following process-oriented pattern-mining algorithm, where AddSubstructure was defined previously:

**Algorithm 1: SUBDUE-based Process Log Compression**

**Input** *mine*: A process mining algorithm (e.g., the Inductive Miner)

*log*: A trace log from some process

**Output** *dendrogram*: A graphical decomposition of the log’s structural features

1. model = *mine*(*log*) #mine the graphical process model
2. traceGraphs = convert(model, *log*) #regenerate the log traces as graphs, using *model*
3. *dendrogram* = {}
4. while not empty(traceGraphs):
5. bestSubstructure = MineBestSubstructure(SUBDUE, traceGraphs)
6. *dendrogram* = AddSubstructure(*dendrogram*, bestSubstructure)
7. traceGraphs = DeleteSubstructure(traceGraphs, bestSubstructure)
8. return *dendrogram*

The DeleteSubstructures method is straightforward removal of a substructure from the set of remaining trace graphs at each iteration:

**DeleteSubstructure Definition**

**Input** *traceGraphs*: The set of remaining trace graphs

*substructure*: A substructure (graph) to be deleted from all trace subgraphs

**Output** *traceGraphs*: The remaining trace graphs after removing *substructure*

1. for trace in *traceGraphs*:

#delete substructure from this graph

1. trace.vertices = trace.vertices \ *substructure.vertices*

#delete any edges within or incident on this substructure

1. for edge in trace.edges:
2. if edge.destVertex in *substructure.vertices*:
3. trace.edges = trace.edges \ edge
4. if edge.sourceVertex in *substructure.vertices*:
5. trace.edges = trace.edges \ edge
6. return *traceGraphs*

Lines 3-7 are the specific steps by which the method becomes lossy, since any edges incident on a substructure are deleted, but only the substructure itself encodes that portion of a trace. Hence the trace cannot be reconstructed completely, in general.

As described in Algorithm 1, the Inductive Miner takes a workflow log and returns a process model by which the entire log is converted into a collection of graphs. This collection is iteratively fed to SUBDUE to discover the most compressing substructure, which is appended to the dendrogram and deleted from all traces. This step repeats until all traces have been compressed to their most elementary substructures. The dendrogram is returned, whose vertices represent compressed substructures, and whose edges represent immediate ancestry between compressing substructures.

The strength of this method lies in the dendrogram as a description of the substructure distribution within the input log. The dendrogram can be analyzed in post-processing for frequent process features, redundant behavior, outliers, and anomalies. The work by [27]successfully implemented a variety of uses for similar SUBDUE-based dendrograms, especially in the context of spaghetti processes, using these representations for visual exploration of process data in their “ESUB” tool. This method of subgraph mining of workflow logs belongs to the family of dendrogram- or tree-induction methods in the process mining literature [28], for which anomaly detection is only one application. For instance, while the low-frequency, outlier components of the dendrogram characterize anomalies, outliers, and noise, the ancestral components encode the most relevant substructures of a log.

## *3.2 Anomaly Detection Method*

Anomaly detection illustrates an effective use of the dendrogram because of its structural properties as a representation of the distribution of substructures. A structural property of the dendrogram is that the size of the dendrogram components decreases smoothly, and then drops suddenly, such that the only remaining traces/subgraphs are those representing anomalies, outliers, and noise. Given that anomalies are assumed to be infrequent and unusual events in the context of regular structure, subgraphs containing anomalies will be among its lower-frequency and lower-information components.

Given these properties, many discriminating anomaly-detection metrics can be devised to distinguish anomalies, noise, and regular patterns. Given that anomalies occur in the context of regular behavior, the anomalous structures tend to have sharply lower frequency than their parent substructures. They are also distinguished from noise in the input log, which tends to result in poorer structural decomposition of a trace, and as such, substructures characterized by noise and their parents tend to both have lower frequency.

A local Bayesian metric was selected to capture this property, based on the frequency of substructures and their parent (immediate ancestor) substructures in the dendrogram. The metric discriminatively quantifies the relationships between parent and child substructures such that unusual child substructures have low probability. Each substructure is assigned a Bayesian probability defined as:

Where unconditional prior probabilities like are defined by the global probability within the log as , where the ‘#’ operator returns the frequency of substructure *s*. For parentless root vertices , although root vertices are nearly always highly compressing and frequent enough to exceed the anomaly threshold.

Characterizing involves defining the probability of a parent relating to one of any of its children, where parents may have multiple children, and children may have multiple parents. One must sum over all parents of a given child, which are a set of independent events, and weight each event by its contribution to the sum:

Each summation component is weighted by to obtain a proper probability distribution, such that . Each is equal to , since that is the proportion of time that is a parent of . Substituting ‘c’ for child, ‘p’ for ‘parent’, and ‘P’ for ‘parents’, the fully-defined metric becomes:

A substructure is flagged as anomalous when , where is the anomaly threshold in . Traces containing the anomalous substructure are flagged, as follows:

**Algorithm 2: Dendrogram-based Anomaly Detection, Using a Bayesian Threshold**

**Input** *dendrogram*: A dendrogram, as output by Algorithm 1

*bayesThreshold*: The anomaly detection threshold

**Output** *anomalyIds*: A set of trace id’s flagged as anomalous

*anomalousStructures*: The set of anomalous substructures

1. *anomalyIds* = {}
2. *anomalousStructures* = {}
3. for substructure in *dendrogram*.*substructures*:
4. bayesProbability = (substructure | parents)
5. if bayesProbability < *bayesThreshold*:
6. *anomalyIds* = *anomalyIds* substructure.traceIds
7. *anomalousStructures* = *anomalousStructures* substructure
8. return *anomalyIds, anomalousStructures*

The method returns both anomalous structures and their corresponding trace ids. Thus, our approach resolves to building a dendrogram via Algorithm 1, then discovering and flagging anomalies via Algorithm 2, which also returns the normative patterns discovered by SUBDUE.

## 4. Materials and Methods

## *4.1 Data Generation Algorithm*

Although real process-oriented datasets are available, they lack the controlled conditions sufficient to evaluate and compare algorithms over a range of data parameters. Instead, a synthetic data generation algorithm was used, as found in appendix A of [19], modified to generate data directly from probability distributions embedded into the generated models. This approach generates random process models from which synthetic traces are generated, and thus the performance of an anomaly detection method can be assessed with respect to a known model and known trace-generation parameters. The same experimental set up was used as described in [19]: 60 randomly-generated process models and 1000 traces per log. Additionally, the Sampling Algorithm, a known algorithm of the same form, was implemented and evaluated on the same log data. Lastly, an evaluation was performed on real-world data derived from software execution logs.

Data generation consisted of two steps: generating process models and then generating traces from them. This process encompassed three primary parameters: ,determining the complexity and distribution of structures within the models; , determining the execution of structural components; and , determining both the generation of anomalous structural components and their execution. The set of parameters described the probability of recursively generating various structural features, including SEQ, OR-SPLIT, AND-SPLIT, and LOOP, defined as follows:

* SEQ: The appending of a single activity.
* OR-SPLIT: A single activity splitting to one of two successors.
* AND-SPLIT: A single activity splitting to two parallel activities, and both are traversed.
* LOOP: An activity splitting to an optional loop, then returning to the activity.

These recursive operators generate directed, potentially cyclic graphs of arbitrary complexity, with the constraint that the graph start at a single START vertex and all paths eventually terminate at a single END vertex. Additional complexity results from including null transitions in the set of “activities”, and as such the splitting constructs may divert to more than two activity paths or may bypass components of a model. The parameters constrain model complexity to a probability distribution over these operators, replicated from [19] and fixed throughout this work.

Probabilistic model-generation allows for unusual or task-trivializing models, so basic checks were applied to ensure sufficient complexity. This included verifying that models contained a minimum START to END path length of one, maximum of one anomalous structures, maximum of four anomalous edges within an anomaly (to constrain anomaly size), minimum of 10 unique activities, and minimum of 10 unique paths from START to END for adequate model complexity. An example model is shown in figure 5.1, with a single insertion anomaly represented by the yellow edges within the region of the black square. Thus with probability , traces along the ‘N’ activity path might end up containing additional behavior, representing unexpected behavior in the context of normal process flow. The non-trivial nature of detecting this anomaly is due to the dependence on the probability of all structural components prior to ‘N’, which may in fact make the prior probability of adjacent paths very low, and thus difficult to flag the anomaly whose observed probability might differ little from its structural neighborhood.

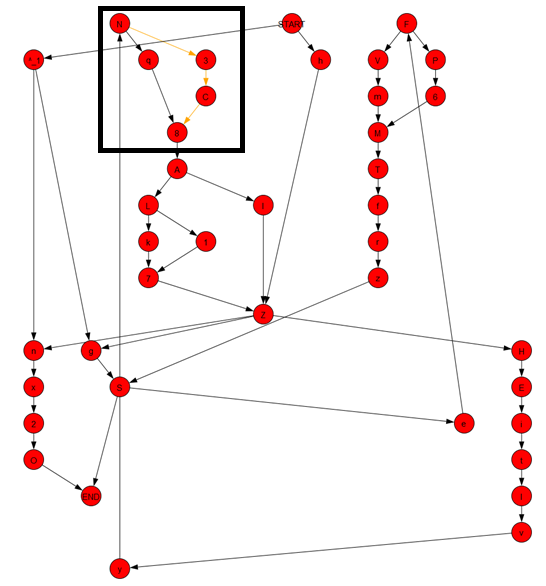


Figure 5.1: A stochastically generated model with a single anomaly. Except for AND constructs, vertices with more than one outgoing edge represent choice vertices, with embedded stochastic parameters determining the choice of edge.

A second parameter set defined the trace-generation distribution constraining the graphical walks of traces, which is only defined for choice operators OR-SPLIT and LOOP. A value of assigned a 0.9 probability of traversing one branch and a of its alternative. By varying from very uniform (0.5) to very non-uniform (0.9), one derives a less uniform distribution of traces, making anomaly detection more difficult as anomalies and regular behavior become ambiguous. To ensure maximum partial-order entropy, activities lying ambiguously within the same timestep were shuffled.

Lastly, the parameters defined the probability of generating anomalies, which encompassed both the generation of anomalous structures and the embedded probability of executing those paths during trace generation. Anomalies in this context are defined as unusual behavior occurring in the context of regular behavior, hence in this work the desire was to generate insertion, substitution, or deletion anomalies in the context of frequent behavior. LOOP and OR constructs were marked as anomalous with fixed probability 0.3. Anomalous paths were marked with a traversal probability that was experimentally varied between 0 and 0.2 in increments of 0.02. This method generated insertion, substitution, and deletion anomalies (since OR branches may include null transitions). Embedding anomalies probabilistically also allowed for generated logs with zero anomalies. This was allowed, to verify that the method was not simply over-generalizing and flagging anomalies even with none present. In this manner, the models output by this method achieved sufficient complexity, and generated an exponential distribution of unique traces with a distribution over substructures determined by the parameters , , and .

## *4.2 Experiment 1: Sensitivity*

In the first experiment, 60 models were generated under fixed , from which 1000 traces were generated separately for values in , and fixed to 0.05. This complete dataset is referred to as D1. For each value, the method was evaluated for values of in 0.02 increments: . A runtime bottleneck is that the method is currently implemented via script components that construct complete application processes (such as ProM [29]) for each log and is not yet implemented in a standalone execution environment. The run-time per log was around 1 minute, and the experiment required about 6 hours total, much of which was merely result compilation but the runtime indicates that this is currently a batch-only method. Accuracy, precision, recall, and f1-measure were averaged over the 60 test models for the cross product of and values, giving the plots and ROC curve shown in figure 5.2.

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | .4 |
| 5 | Figure 5.2: Dataset D1, experiment 1 result plots for accuracy (1), f1-measure (2), recall (3), and precision (4). The x/y axes of all curves were oriented for best visualization, so note their orientation carefully. ROC curve (5) is shown at left. |

For all four performance curves, performance degraded only slightly along the axis for larger values, indicating the method worked well for very skewed trace distributions. Clearly, impacted performance most strongly. From the top-left, accuracy maximized around , then tapered gradually as larger values increased the false positive (FP) rate. Recall, the ability to flag all anomalies, maximized quickly for low values of ,as one might expect. Precision and the f1-measure were more informative, since precision clearly dominated the f1-measure in contrast with recall. Precision, the proportion of flagged traces that were anomalies, was likewise maximized for small values and informs us where the greatest saturation of anomalies occurred. The f1-measure results are most informative in terms of selection, suggesting one choose an value of around 0.07, with a corresponding accuracy of 96%. Additionally, the receiver-operator characteristic (ROC) curve TPR/FPR values was plotted for all values of in 0.02 increments, averaged over all 60 models and all values of . The area under the ROC curve is very near 1.0, indicating a high true-positive rate.

The results show lower values (0.04-0.10) of were preferable and can be tuned to suit recall vs. precision objectives. From a risk perspective, recall is the most important in terms of capturing all anomalies, at the expense of decreasing accuracy and precision. On the other hand, precision and f1-measure provide better comparative performance metrics, and f1-measure shows room for improvement. However, the sharp maximum of the precision curve along the axis indicates the method and the Bayesian metric worked as intended, distinguishing anomalies from regular structure with a sharp boundary, for the synthetic data parameters.

Finally, this experiment analyzed performance over a range of and , but with fixed to 0.05. The results demonstrated that an algorithmic parameter, ,determined the results far more than the data parameter, . Thus, more experiments were required to stress properties of the data.

## 

## *4.3 Experiment 2: Sensitivity*

A second experiment evaluated the sensitivity of the method with respect to ,and with fixed. This was to verify that the previously selected did not trivialize the task of anomaly detection. In this case, and trace component of was varied between 0.0 and 0.2 in increments of 0.02, then in increments of 0.05 between 0.25. The choice of was to yield a more uniform distribution of traces, and likewise recall that experiment 1 results showed little impact across the parameter’s range. The low range of was expected to trivialize the discovery of anomalies due to their low expected frequency, whereas the high range approached , making anomalous substructures and normal substructures ambiguous. The same models as dataset D1 were used, but new traces were generated for each parameter value, generating dataset D2.

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |
| 5 | Figure 5.3: Dataset D2, experiment 2 results for (from top-left) accuracy (1), f1-measure (2), recall (3), and precision (4). The x/y axes of all curves were oriented for best visualization, so note their orientation carefully. ROC curve is shown at left (5). |

The results are shown in figure 5.3. As expected, the four metrics show performance worsening along the axis approaching 0.5, with performance diminishing rapidly above approximately 0.3. However, the decay was smooth, showing the method worked satisfactorily for a range of very rare and somewhat frequent anomaly occurrence rates, with respect to a somewhat regular process. The ROC curve bears this out, and likely anticipates expected performance on real-world data, for which  and are not known in advance.

Whereas the previous datasets D1 and D2 included insertion, substitution and deletion anomalies, they excluded a specific type of substitution: anomalous structure consisting of existing model activities. Such anomalies represent activities occurring out of context. Experiments 1 and 2 were repeated on the same data, but with the anomalous activities in each model replaced by an existing activity randomly chosen from the model’s non-anomalous activities. New traces were generated as for D1, 60 models and 1000 traces for , giving dataset D3. Dataset D4 was generated likewise but for various values of , as per experiment 2.

Results for experiment 1 using dataset D3, and likewise for experiment 2 using for D4, showed very nearly the same performance curves as for the previous, with just slightly lower performance. For brevity, we do not reproduce the performance curves, and instead provide the ROC curves for each experiment, which capture the slight performance decrease experienced in both experiments. The decrease was due to the less obvious nature of anomalies when replaced with existing activities.

|  |  |
| --- | --- |
|  |  |

Figure 5.XX: The ROC curves for experiment 1 (left) repeated on dataset D3, and experiment 2 (right) repeated for dataset D4.

## *4.4 Experiment 3: Multiple Anomalous Structures*

Experiments 1 and 2 analyzed performance sensitivity with respect to and , but dealt with models for which the number of anomalous structures was distributed according to the model generation algorithm. As a result, models contained between 0 and 2 anomalous structures by which to generate anomalous traces. These model classes are denoted as , where *k* defines the number of anomalous structures in model *M*. Despite that larger values of in experiment 2 generated more anomalous traces, they were generated from models containing few anomalous structures. Therefore, a remaining task was to analyze performance sensitivity to a controlled range of .

This required generating a final model-oriented dataset D5 consisting of models with *k* anomalous structures for . Thirty models were generated for each *k*, each with a single log generated from fixed trace parameters: and . These values were not truly fixed, since the outgoing edge probabilities required normalization after anomaly generation. For example, while the model generator might generate an activity vertex with two equiprobable outgoing “OR” edges, it might also add an anomalous edge to this vertex, as described later. This vertex’ outgoing edge probabilities would then be, which do not sum to 1.0, so the edge probabilities at each vertex were normalized after anomaly generation. An example model with 16 anomalies is shown in figure 5.6.



Figure 5.6: A synthetic model with 16 anomalies (yellow paths) and approximately 40 activity vertices. Vertices prefixed “^\_” represent null activity transitions, where such a path immediately resolves to its successor.

Models were generated using the same procedure as for previous experiment until one satisfied the target number of anomalous structures. Models with were very unlikely under the previous model generation parameters, so in this case anomalous structures were added until the target was reached. To achieve a nearly uniform distribution of insertion, deletion, and substitution anomaly structures, anomaly types were selected by the stochastic decision tree shown in figure 5.7.



Figure 5.7: The anomalous structure generation decision tree and three anomaly types. Null transitions '^' represent execution paths that bypass (delete) normal behavior. LOOP structures transition and return from a newly inserted activity, creating an insertion. Lastly, OR branches decompose to two types of substitution, whereby normal behavior is replaced by either some existing activity () or a new activity ().

With uniform probability, the method chose to add an OR, LOOP, or null-transition anomalous structure. A null transition consisted of an edge from a randomly selected vertex to some downstream vertex, resulting in a deletion anomaly. A LOOP was inserted into the model by adding edges between a new activity vertex and a randomly selected vertex, creating an insertion anomaly. For OR, a new edge was added to a randomly selected vertex, rejoining the model at a random downstream vertex. However, along this OR edge, a single vertex was inserted which was either a new activity or a randomly selected existing (non-anomalous) activity. Both cases generate substitution anomalies, but the former adds a new activity that might be easily detectable, whereas the latter adds ambiguity by using an existing activity. Adding anomalous structures via this stochastic decision tree resulted in models with a target number of *k* anomalies, each of which was generated under a uniform distribution of insertion, deletion, and substitution anomalies.

Given dataset D5, generated by the procedure described above, the method was re-run and evaluated over the cross product of values in terms of accuracy, recall, precision, and f1-measure, averaged over thirty models/logs. Here and in increments of 0.02. The results are shown in figure 5.8.

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |
| 5 | Figure 5.8: Performance results for D5, experiment 3 over a range of k-anomaly values for (from top-left) accuracy (1), f1-measure (2), recall (3), and precision (4). These visuals were oriented to convey the curvature of each metric, so attention must be paid to the orientation of the x/y axes. ROC curve is shown at left (5). |

Overall, these results were most similar experiment 1, since and were fixed, and experiment 1 showed little variance over the range of . The curvature shows accuracy decaying in linear fashionalong the ‘Anomalous Structures’ axis for larger *k,* due to the larger numbers of anomalous traces in the logs. Again, recall quickly maximized since small values detected most anomalies. Precision remained high across a range of *k* anomalous structures despite decreasing accuracy, since for larger *k* values the set of flagged traces became saturated with real anomalies simply because there were many more in the data. The ROC curve bears out the overall performance, which did not suffer substantially compared with the previous experiments.

In sum, the results demonstrated that the method can handle multiple anomalous structures, and subsequently many anomalous traces. However, the method arguably benefited from the ease of finding anomalous structures when more anomalous structures were added, but the anomalies arose from separate-but-rare events. This potentially benefited results for larger *k* by simply adding more of the anomalous class to the data, but under a distribution that remained favorable to finding them. Hence there could be more anomalies to find, but without substantially decreasing the probability of finding them. By increasing one can expect worsening performance for larger k, as experiment 2 demonstrated. This implies that has the greatest negative impact on the performance of the algorithm, which is to be expected since larger values make anomalies indistinguishable from normal behavior and noise.

## *4.5 Comparison with Existing Methods*

To provide further context for these results, the Sampling Algorithm from [19] was also evaluated, the same work upon which we based our data generation scheme. The authors reported their best results using this algorithm with optimized parameters, using similar synthetic data and anomalies. The Sampling Algorithm is defined as:

|  |  |
| --- | --- |
| **Algorithm 3: Sampling Algorithm** | |
|  | **Input** L: A process log  s: Sampling proportion,  mine: A process mining algorithm |
|  | **Output** TA: The set of traces flagged as anomalous |
| 1. | T = set of all unique traces from the log L |
| 2. | TC = { } #used to contain anomalous candidate traces |
| 3. | TA = { } #used to contain traces flagged as anomalous |
| 4. | **for** **do:** |
| 5. | if then: |
| 6. |  |
| 7. | **for**  **do:** |
| 8. | S = sample of s% of traces of L |
| 9. | M = mine(S) |
| 10. | if t is not replayable on M: |
| 11. | TA = TA t |
| 12. | **return** TA |

The Sampling Algorithm anticipates that an anomalous trace will deviate significantly from the expected process model derived from a randomly selected subset of the traces. It begins by gathering low-frequency outlier traces from a log. For each trace in this set, a process model is mined from a randomly selected subset of the traces. The trace is added to the anomalous trace set if it is not replayable on the mined model. In contrast with Algorithm 2, the Sampling Algorithm flags anomalies without providing causal, structural context for the flag, and does not provide the complete feature model which the dendrogram provides.

The Sampling Algorithm was implemented and evaluated on datasets D1 and D2, for using a frequency threshold of 0.02 and a sampling rate of 0.7, parameters suggested by the authors’ highest performance results. The goal was only to derive a straightforward performance baseline with which to compare the results of Algorithm 2, which did not require exhaustively testing over a range of values. Exhaustive testing was difficult because of implementation dependencies on ProM, which required significant test run time. This was not a fault of the Sampling Algorithm nor of ProM, but of the test system’s integration with ProM’s command line features. As a result, performance was tested and averaged over only 30 of the 60 models, which was still a confident number of models. Results for D1 and D2 were virtually identical, so only D1 results are shown for brevity.

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |

Figure 5.9: From top-left, dataset D1 Sample Algorithm results, (1) accuracy, (2) f1-measure, (3) recall, and (4) precision for .

Table 5.1 compares performance of Algorithm 2 with a generic value of , beside the Sampling Algorithm for dataset D1. The column shows the difference between the algorithms. Results were simply averaged over all values under test.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 5.1: Dataset D1 Performance Comparison** | | | |
| Algorithm | Algorithm 2 | Sampling |  |
| Accuracy | 0.972 | 0.717 | +0.255 |
| Recall | 0.892 | 0.924 | -0.032 |
| Precision | 0.700 | 0.186 | +0.514 |
| F1-Measure | 0.721 | 0.233 | +0.488 |

As shown in table 5.1, the Sampling Algorithm performed well, but below Algorithm 2 for all metrics except recall. The performance discrepancies are attributable to their different objectives, since Algorithm 2 targets anomalous behavior in the context of regular behavior with greater precision, whereas the Sampling Algorithm is concerned with detecting deviation with respect to expected model structure. Its authors described high recall as a primary performance objective, which is justified by these results and explains the low precision.

## 

## *4.6 Real Data Evaluation*

For real system data evaluation, the method was applied to a dataset consisting of 2,566 traces representing activity-sequences of code function calls made during software unit-tests of the NASA Crew Exploration Vehicle (CEV) [30]. The CEV system software implements a structured aerospace mission model, and the unit-test suite executes components of this implementation. Thus, the test data provided a description of called components and code paths, albeit triggered by unit test calls, providing a context to use this pattern mining and anomaly detection method to evaluate discrepancies between system design and behavior, and to detect unusual code executions.

Unit-testing emphasizes code coverage, hence the structural distribution of these traces modeled the test suite design, whereas traces representing normal system operation would be more desirable. Nonetheless, the dataset comprised a normative view of the system from the design perspective and provided a suitable demonstration of the method’s model checking and anomaly detection potential. The demonstration is unsupervised, since the traces in this dataset were not labeled as anomalous/non-anomalous. The findings of the method are ‘anomalous’ per unit test design or system behavior, in terms of unusual behavior in the context of normative patterns. The number of anomalies detected were tracked for various Bayesian thresholds, shown in table 5.2. The total run time for these results was approximately two minutes.

|  |  |
| --- | --- |
|  |  |
| (1) The mined process model of function calls (blue vertices), highly connected in terms of core functionality. | (2) The mined dendrogram of 33 substructures. Each red vertex represents a subset of blue vertices/edges from the model at left. |
|  |  |
| (3) A highly compressing substructure, SUB2, demonstrating the encapsulated structure of rocket burn functionality. | (4) SUB27, an anomaly including a call to an error function amidst critical calls to a rocket burn function. |

Figure 5.10: Models generated by the method on the NASA CEV software test dataset.

|  |  |  |
| --- | --- | --- |
| **Table 5.2: NASA CEV dataset test results** | | |
| Number of traces: 2,566 | | |
| Number of activities: 34 | | |
|  | Anomalies detected | % of log |
| 0.01 | 18 | 0.7 |
| 0.03 | 183 | 7.1 |
| 0.05 | 966 | 37.6 |
| 0.07 | 1,348 | 52.5 |
| 0.09 | 1,620 | 63.1 |

The results revealed that from the synthetic data experiments did not generalize well for this log, flagging over half of the traces in this data. This was due to an extremely heavy tailed trace distribution, with many low frequency outliers. Evidently this was because the data consisted of white-box unit tests, rather than data reflecting the distribution of function calls during normal system operation. By reducing in increments of 0.02, the subset of anomalous traces was reduced to those of more critical importance; such as substructure 27 (figure 5.10.4), which exhibited error behavior by an error-log call between rocket burn calls. Conversely, the most compressing patterns, such as figure 5.10.3, represented well-encapsulated code components.

Such structures allow system evaluators to identify exception-like behavior unintentionally included in a system, a critical task in software model verification. For instance, “anomalies” and normative patterns provide useful insights into the distribution of risk across code regions, by identifying “spaghetti” code in need of more rigorous testing, refactoring, or code interfaces straying from design. Non-anomalous substructures, such as figure 5.10.3, identify reusable components defined by recurring subgraphs of function calls (frequent code paths), hence, high-cohesion. Loosely, the graphical similarity of substructures with respect to design components lends insight into code quality (high similarity, strong cohesion), or the need to refactor (low similarity, poor cohesion), and provide a valuable grey-box test aid.

## 

## *4.7 Result Summary*

Summarizing the results, the evaluation of Algorithm 2 showed very good performance for datasets under a range of controlled conditions, as well as promising outputs on a real dataset. The parameter was shown to impact performance far less than , for which performance decreased as the value approached . Additionally, the method showed comparable performance to a previously published anomaly detection method, the Sampling Algorithm from [19]. Finally, Algorithm 2 was tested on a real dataset containing no anomaly labels, but its output provided many useful properties of the input data and demonstrated the use of the method for systems verification and analysis. The results show strong performance on the task of anomaly detection, but also represent an algorithmic approach engineered for the specific objective of anomaly detection.

**5. Conclusions and Future Work**

Overall, the results indicate the anomaly detection method presented in this work succeeded for a range of model complexity, trace complexity, and anomaly characteristics, and demonstrates the value of graphical compression for process mining applications. Additionally, the method is tunable via the parameter to suit different datasets and performance objectives. Another notable advantage is that such an unsupervised approach requires no prior process model, nor exceptional tuning to derive normative patterns. This makes it an extensible analysis tool when applied to processes with no prior definition or pre-defined policy. Such scenarios occur frequently for computer networks, distributed systems, and communication protocols, for which detecting anomalies in system behavior is crucial. A final advantage is that the method is capable not only of flagging anomalous traces, but also of returning their anomalous features.

There are several potential drawbacks to this method, firstly that it is a batch-oriented approach, since deriving the feature model of a trace log requires a non-trivial amount of time and sufficient traces to discover regular structural patterns. A second drawback is its noise intolerance, a recurring problem faced by mining algorithms. The graphical patterns discovered by SUBDUE become the only patterns by which the log is recompressed, such that even small deviations to a normative pattern are ignored and may later be flagged as anomalies. This strongly discriminatory property is in fact why the method works as desired, and the high dimensionality of graphical data typically requires such heuristics. However, often the goal of process mining is to create a more noise-tolerant balance between specificity and generality. The substructure decomposition of the approach is both a strength and a potential criticism, consistent with the recurring process-mining discussion on specificity-generalization tradeoffs.

A final detraction is that the method is a strongly purpose-built algorithmic approach, designed from the perspective of searching through discrete graphical representations. The use of the Bayesian anomaly detection metric also incorporates foreknowledge of expected data characteristics. Although the method is unsupervised, its design was heavily tailored to its intended purposes. However, graphical deep learning models, such as auto-encoders and recurrent networks with graphical input representations, can potentially encode normative patterns in real-valued parameters, softening the problems of discrete-search approaches [31]. These models generally require less manual tuning and fewer hyper-parameters, a primary objective of deep learning. From a theoretical perspective, the most promising work lies in adapting such learning models to process mining by leveraging the view of process data as a distribution over substructures. The value of these approaches is that they can potentially perform end-to-end normative pattern mining and anomaly detection without as much manual engineering.

References

1. Dumas, M., Van der Aalst, W. M., & Ter Hofstede, A. H. (2005). Process-aware information systems: bridging people and software through process technology. John Wiley & Sons.
2. Leemans, S. J., Fahland, D., & van der Aalst, W. M. (2013, August). Discovering block-structured process models from event logs containing infrequent behaviour. In *International Conference on Business Process Management* (pp. 66-78). Springer, Cham.
3. Holder, L. B. (1989). Empirical substructure discovery. In *Proceedings of the sixth international workshop on Machine learning* (pp. 133-136).
4. Petri, C. A., & Reisig, W. (2008). Petri net. *Scholarpedia*, *3*(4), 6477.
5. http://www.xes-standard.org/
6. Russell, N., Ter Hofstede, A. H., Van Der Aalst, W. M., & Mulyar, N. (2006). Workflow control-flow patterns: A revised view. *BPM Center Report BPM-06-22, BPMcenter. org*, 06-22.
7. Van der Aalst, W. M. (2011, April). Process mining: discovering and improving Spaghetti and Lasagna processes. In *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on* (pp. 1-7). IEEE.
8. Van der Aalst, W.M.P. *Process Mining Discovery, Conformance and Enhancement of Business Processes.* Heidelburg, Germany: Springer, 2011.
9. Peterson, J. L. (1981). Petri net theory and the modeling of systems.
10. Jensen, K. (1987). Coloured Petri nets. In *Petri nets: central models and their properties* (pp. 248-299). Springer, Berlin, Heidelberg.
11. Van der Aalst, W. M.P. (1997, June). Verification of workflow nets. In *International Conference on Application and Theory of Petri Nets* (pp. 407-426). Springer, Berlin, Heidelberg.
12. Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541-580.
13. Karp, R. M., & Miller, R. E. (1969). Parallel program schemata. Journal of Computer and system Sciences, 3(2), 147-195.
14. Esparza, J., & Nielsen, M. (1994). Decidability issues for Petri nets. *Petri nets newsletter*, *94*, 5-23.
15. Akoglu, L., Tong, H., & Koutra, D. (2015). Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, *29*(3), 626-688.
16. Shetty, J., & Adibi, J. (2004). Enron email dataset. USC Information Sciences Institute, Tech. Rep.
17. Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems*, *1695*(5), 1-9.
18. Bezerra, F., Wainer, J., & van der Aalst, W. M. (2009). Anomaly detection using process mining. In Enterprise, Business-Process and Information Systems Modeling (pp. 149-161). Springer, Berlin, Heidelberg.
19. Bezerra, F., & Wainer, J. (2013). Algorithms for anomaly detection of traces in logs of process aware information systems. Information Systems, 38(1), 33-44.
20. Djoko, S., Cook, D. J., & Holder, L. B. (1997). An empirical study of domain knowledge and its benefits to substructure discovery. IEEE Transactions on Knowledge and Data Engineering, 9(4), 575-586.
21. Noble, Caleb C., & Cook, D. J. (2003, August). Graph-based anomaly detection. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 631-636). ACM.
22. Diamantini, C., Genga, L., Potena, D., & Storti, E. (2013, May). Pattern discovery from innovation processes. In *Collaboration technologies and systems (CTS), 2013 international conference on* (pp. 457-464). IEEE.
23. Genga, L., Potena, D., Martino, O., Alizadeh, M., Diamantini, C., & Zannone, N. (2016, September). Subgraph mining for anomalous pattern discovery in event logs. In *International Workshop on New Frontiers in Mining Complex Patterns* (pp. 181-197). Springer, Cham.
24. Jonyer, I., Cook, D. J., & Holder, L. B. (2001). Graph-based hierarchical conceptual clustering. Journal of Machine Learning Research, 2(Oct), 19-43.
25. Jonyer, I., Holder, L. B., & Cook, D. J. (2000, May). Graph-Based Hierarchical Conceptual Clustering. In FLAIRS Conference (pp. 91-95).
26. Eberle, W. and Holder, L., Anomaly Detection in Data Represented as Graphs, *Intelligent Data Analysis, An International Journal*, Volume 11(6), 2007.
27. Diamantini, C., Genga, L., & Potena, D. (2015). Esub: Exploration of subgraphs. Proceedings of the BPM Demo Session, 70-74.
28. Song, M., Günther, C. W., & Van der Aalst, W. M. (2008, September). Trace clustering in process mining. In *International Conference on Business Process Management* (pp. 109-120). Springer, Berlin, Heidelberg.
29. ProM: The Process Mining Toolkit. Version 6.6. Retrieved from http://www.promtools.org/doku.php.
30. Leemans, Maikal. (2017). NASA Crew Exploration Vehicle (CEV) software event log. http://doi.org/10.4121/uuid: 60383406-ffcd-441f-aa5e-4ec763426b76, 2017
31. Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge: MIT press.