ANOMALY DETECTION IN PROCESSES REPRESENTED AS A GRAPH

By

JESSE DEAN WAITE

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

JULY 2018

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of JESSE DEAN WAITE find it satisfactory and recommend that it be accepted.

Lawrence Holder, Chair

Janardhan Rao Doppa

Ananth Kalyanaraman

ACKNOWLEDGMENT

I would like to thank my adviser, Lawrence Holder, for his support and instruction over the previous five years. Not only was he a patient and informative adviser for this work, he was my first artificial intelligence instructor, and previously advised my undergraduate work on language prediction models for alternative and augmentative communication (AAC) devices, the project that established my interest in data science and machine learning applications. I also thank Janardhan Rao Doppa for his kind and formal instruction in structured prediction and machine learning. Through his course I learned the structure of some of the most difficult machine learning concepts and approaches, which I could not have done without his thorough exposition and implementation help. Likewise, I thank Ananth Kalyanaraman, whose courses give students a rigorous introduction to sequential problems and their implementation, which I continue to use frequently. Lastly, I thank Assefaw Gebremedhin, whose network science course informed much of the development of this thesis. It is because of the patience and formal preparation of WSU professors like Jana, Ananth, Assefaw, and Larry that such advanced concepts are accessible to novices like myself and can entertain our often stumbling questions. I extend them my sincere gratitude and hope they continue to set an example for their undergraduate and graduate students.

I would also like to thank the many authors in the field of process mining for their role in this thesis, since this project would not have been possible without their open-source software tools, such as the ProM process mining platform. Likewise, I thank the many other process mining, graphical, and machine learning software developers whose labor made this thesis possible through open-source tools and methods, all of which have democratized the application

of machine learning and data analysis. These anonymous library authors lead the quiet revolution in data science and their contributions to the world are remarkably under-credited.

ANOMALY DETECTION IN PROCESSES REPRESENTED AS A GRAPH

Abstract

by Jesse Dean Waite, M.S.
Washington State University
July 2018

Chair: Lawrence Holder

Learning the structure of stochastic, noisy environments remains an important area of process mining 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, providing 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. The method 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 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. Final conclusions and future work are provided in closing.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Dedication

For Harlie, a very good dog.

CHAPTER ONE: INTRODUCTION

1.1 Process Mining Overview

The field of process mining involves the extraction and analysis of process models derived from data characterizing the execution of an underlying, hidden process. Applications often involve business process management, such as tracking a manufacturing production cycle or a hospital patient admission process. However, the representation of process data and model mining has general application to natural sciences, industrial engineering, and virtually any discrete structured task environment for which a researcher wishes to understand the structural, temporal, or quantitative features of a known or unknown process. Thus, process mining bridges model construction and data analysis using a common set of formalisms for constructing models from data, and for subsequently analyzing these models according to recurring patterns and use-cases within a specific domain.

Although process mining has many other applications, its two most prominent tasks are model construction and conformance checking. Model construction is primarily used when a process is unknown, in the sense that no prior process model exists or simply no process mining system has yet been applied to the process. Construction involves inferring a prototype structured model *M'* from a set of process-oriented data *D*, where *M'* abides a formal language definition (such as a Petri net, as defined later), and *D* is specified by a process-oriented data schema and contains traces of process executions generated from the true underling process *M**. These terms are developed later in further detail. It suffices to think of model construction as the abstract task of reconstructing a structured graphical model of a hidden process from traces representing executions of the process. For example, the transaction logs of user interactions with a commerce website can be used to reconstruct the graphical model of an online shopping process and its

most frequent customer use-cases. Similarly, a log containing protocol messages for an automated distributed system can be used to reconstruct the industrial process they implement. Hence, model construction has applications that are both organic (noisy, loosely structured) as in the website example, or highly specified, as in the industrial process.

The second task, conformance checking, assumes that one has access to a process model M^* for some dataset D. It subsequently involves analyzing conformance between the behavior of D and some prior model specification M^*. For instance, a bank might specify strict policies on all aspects of a complex financial transaction to detect fraud or other deviations from the prescribed process model. In this case, conformance checking is likely very straightforward in terms of detecting missing, unexpected, or out-of-order events, such as detecting the approval of transactions without prior digital signatures or detecting a loan approval before a required background check. However, conformance checking may also encompass more continuous schemes for evaluating conformance according to the model's objectives. In this work, conformance checking is implemented via unsupervised anomaly detection, the task of identifying unusual activity occurring in the context of the normative patterns of a process. These normative patterns represent a form of model construction, but their evaluation is beyond the scope of this work, which is to use these patterns to perform anomaly detection.

Model construction and conformance checking remain open problems because the complexity of structural data usually requires heuristic approaches, hence new heuristic methods are always being discovered and better approximations are always possible. These problems also possess many natural extensions, such as conformance checking processes with various costs, constraints, or bottlenecks assigned to their structural features, which is an active area of

research. Such cases arise in operational settings for which a practitioner wishes to predict cost, resource, or other bottlenecks in a process, not just simple transition models.

Overall, the breadth of process mining is attributable to the multiple objectives it encompasses, as well as structural complexity that does not readily lend itself to algorithms that are both optimal and efficient. Most often, only one of these attributes is obtainable at the expense of the other. However, it is easy to understand the value institutions place on automated process intelligence to guide their decision making. This creates abundant opportunities, but also involves problems whose complexity is resilient to straightforward, canonical solutions and often reduce to classic intractable problems in operations research and computational theory.

1.2 Contributions

This work contributes to the fields of process mining and graph mining by devising a new method for process feature extraction and anomaly detection using graph compression. Using this unsupervised method, one can gain insight into many different processes and their structural properties. The method allows such processes to be tracked, measured, and improved in a manner which previously required prior knowledge of the process under evaluation. In complementary fashion, the method allows one to detect anomalies in unknown processes without prior domain knowledge of the process, its implementation, or its prior formal specification.

The primary objective of this research was to develop graph compression methods further for the purposes of process mining and using process mining formalisms. Recursive graph compression was applied previously by Jonyer, Cook, and Holder (2000, 2001) using a method that did not work in the same manner. The remedy for use in the process mining domain was a

modification that yielded unexpectedly good results and possesses broader application in unsupervised learning.

The method presented in this work accomplished the following:

- A method for process modeling, model refinement, and structural analyses of processes
- A method of anomaly detection capable of causal structural attribution
- A hierarchical feature representation of the structural components of unknown processes
- Open-source software tools available at https://github.com/niceyeti/PMTools
- Synthetic datasets and data generation software for benchmarking process mining tasks

The overall framework serves as a starting point for future research using other structural learning methods and potentially deep learning. Learning structure within high dimensional data embodies the ability to learn compressed representations of a high-dimensional world, a core requirement of general purpose machine learning and artificial intelligence. The method presented in this work advances this goal directly, but also provides useful problem characterizations and datasets for use by implementers of alternative solutions for similar process-oriented tasks.

1.3 Outline

The remainder of this work is organized as follows. Chapter two introduces process mining in further detail, then characterizes the task of anomaly detection within this domain. Additionally, graph compression and graph representations are introduced, as they are used throughout this work. With this context established, chapter three introduces required terminology and defines the specific problem of process anomaly detection, its evaluation metrics, and its complexity. Chapter four then lays out the proposed method of graphical trace compression and its anomaly detection method. The anomaly detection method is evaluated in

several experiments within chapter five, along with a description of the datasets and experiment methods used. The objective of these evaluations was to demonstrate the performance of the method over a range of possible inputs and internal conditions. This was done by defining the necessary data generation schemes and then evaluating the performance for these datasets over a range of parameters, followed by a discussion of the results. Real-world data evaluation is also included chapter five. Finally, conclusions and potential future work are discussed in chapter six.

CHAPTER TWO: BACKGROUND AND RELATED WORK

2.1 Process Mining

As described in (Dumas et al., 2005), 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 definition formalizes operational management systems as systems centralizing awareness of process data as well as the ability to prescribe tasks and activities via process models. These tasks form a loop by which process models can be defined, tracked, and evaluated; likewise, process models may be derived and analyzed from process data.

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 multiple systems. This is amenable to realistic scenarios in which processes are embedded within a non-stationary framework of changing people, tools, resources, and tribal knowledge, often in the absence of prescribed process models. These scenarios occur frequently, since modeling such environments may only occur due to some ad hoc objective, such as an audit or root-cause analysis of a process failure. In these scenarios, a PAIS is a disparate collection of operational systems and data sources by which to derive traces characterizing the underlying process-oriented view of a system.

The ability to mine and analyze normative process patterns in these unstructured contexts is critical for mining regular activity and 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 that

also possesses useful anomaly detection properties. The Inductive Miner (Leemans et al., 2013) is used to construct a graphical process model from log data, to which SUBDUE (Holder, 1989) is iteratively applied to extract a hierarchical dendrogram of normative patterns executed on this model. Using this representation, anomalies and significant process features can be discovered in post-processing.

This hybrid approach is practical since the Inductive Miner extracts generality from process log data, outputting a graphical model M capable of generating all traces in a process log, including noise. SUBDUE then extracts a collection of M's most informative components, constructing a hierarchy of sub-structures of M most relevant to the log as a dendrogram. Using this unsupervised method, one can mine normative process patterns, detect anomalies to those patterns, and perform other analyses.

2.2 Process Representations

Process modeling encompasses many different process algebras and formal languages, all of which have been developed to specify properties of different representations of processes. Thus, model representations may encompass varying levels of complexity to describe a process, or to query properties like verifiability, reachability, and completeness, which often depend on the choice of representation (Peterson, 1981). The canonical example is the Petri net, whose properties are well-studied but remain a perennial subject of active research because of their expressive power (Petri, 2008). 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" P, representing activities, resources, or other entities of the process. Places are separated by "transitions" T, which semantically constrain the nature of transitions between places. Edges E connect places and transitions to give the model its structure, where transitions connect to places, and places

connect only to transitions, $E \subset (P \times T) \cup (T \times P)$. The model is given an initial marking, which is a token placed on one or more places to denote the initial state of the process. Lastly, a final marking is assigned in the same manner as the initial marking, to denote the final state of the process.

Table 1.1: Simple Petri net definition

$\boldsymbol{P = \{p_1 \ldots p_i\}}$	A set of places, where $i = \lvert P \rvert$
$\boldsymbol{T = \{t_1 \ldots t_j\}}$	A set of transitions, where $j = \lvert T \rvert$
$\boldsymbol{E \subset (P \times T) \cup (T \times P)}$	A set of edges from places to transitions $(P \times T)$ and transitions to places $(T \times P)$
$\boldsymbol{P_{initial} \subset P}$	Initial marking, the initial set of places of the process' execution
$\boldsymbol{P_{final} \subset P}$	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 graphical model is constrained by the initial marking $\{P0\}$, final marking $\{P5\}$, and the transitions between places (activities) comprising the core activities of some process. A walk is given by a token moving scheme, whereby no transition can "fire" until sufficient tokens have reached it, which for this simple example could require completion of all places incident to a transition. Many other semantics are possible, such as colored Petri nets, wherein tokens are not single items but are instead functions evaluated at places (Jensen, 1987). 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 for the verification of distributed systems. An overview is provided by (Van der Aalst, W. M., 1997).

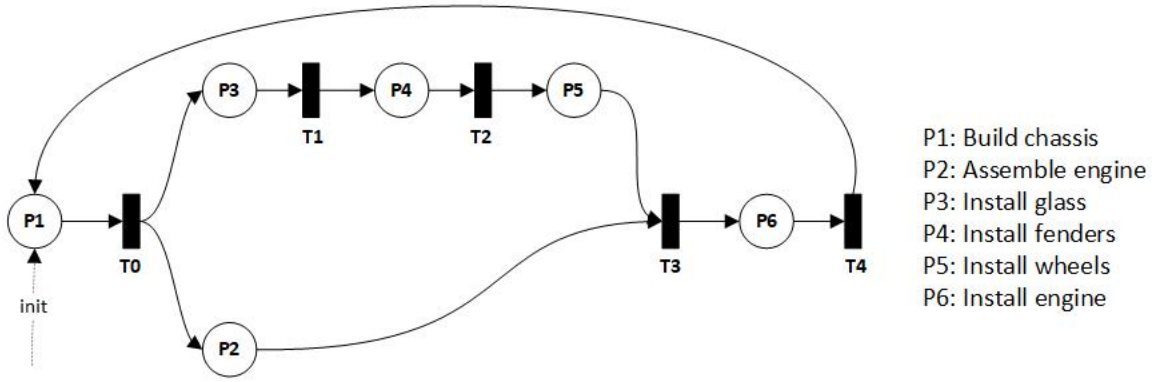


Figure 2.1: A Petri net model for a simplified automotive manufacturing process.

The visual structure of a process model itself imparts significant information as well. In this model, the path along the chassis-assembly activities $\{P3, P4, P5\}$ likely represents a bottleneck, since engine assembly $\{P2\}$ may entail significant waiting before body assembly completes at *P5*, and thus *T4* can "fire", and the engine can be installed, completing a manufacturing cycle.

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 (Murata, 1989). One such derivation is the coverability graph given by a Petri net like the one in figure 2.1, a landmark representation in computation theory by (Karp and Miller, 1969) which permits a variety of analyses, a recent summary of which is provided in (Yamamoto, 2017). The coverability graph consists of the graph of all reachable markings given by a well-defined Petri net, which can be simulated by an oracle moving tokens from places to transitions. In this example, the oracle begins by placing tokens on each of the places in the initial marking set. Representing the marked places as a binary vector $\boldsymbol{b}_{place} = < b_{P0}, b_{P1}, b_{P2}, b_{P3}, b_{P4}, b_{P5} >$, $b_{Pi} \in \{0,1\}$, the initial marking for the above Petri net without subscripts is $< 1, 0, 0, 0, 0 >$. From this marking, the oracle moves tokens 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 $\{P1, P2\}$, giving marking $< 0,1,1,0,0 >$. 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 complete 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 potentially infinite since the process can repeat from *T4*, depending on how the final marking is defined. This edge could be intended to characterize serial process flow, whereby new car production cannot begin until the previous cycle completes, a simple example of "cycle time" often used in computer processor design and business production models. The terminal conditions of this specific process are left ambiguous since the goal is only to show how processes depend on transformative graphical representations to determine many of their properties. Like many complex graphical problems, the ability to solve process problems is imbued by the expressiveness of the representation used to describe them.

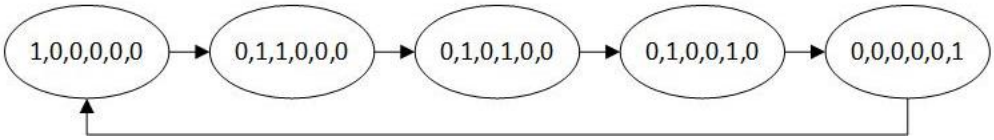


Figure 2.2: Coverability graph of the Petri net in figure 2.1.

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 (Esparza and Nielsen, 1994).

This work focuses on viewing trace data as a distribution over structures, which are connected subsets of Petri model graphs using graphical representations detailed in subsequent

sections. However, it is necessary to establish how graphical representations of processes accommodate specific mining objectives, as with the Petri net and Karp-Miller graph in the preceding examples. This may be obvious given the inherently graphical nature of processes, but it emphasizes how the solution to many problems rests on problem-transformative graphical representations. Thus, solutions in this domain often reduce to a matter of expression.

2.3 Anomaly Detection in Practice

As defined by Chandola et al. (2009), anomaly detection "refers to the problem of finding patterns in data that do not conform to expected behavior. These nonconforming patterns are often referred to as anomalies, outliers, discordant observations, exceptions, aberrations, surprises, peculiarities, or contaminants in different application domains" (p. 1-2). Accordingly, anomaly detection requires solving a prior problem of pattern recognition and analysis to describe the normative patterns of data. The definition abides multiple interpretations depending on the application domain, since the term "anomaly" is sometimes used interchangeably with "outlier" or "noise", but this work preserves their distinctions as follows: anomalies are regarded as abnormality that occurs in the context of a normative pattern. Hence, deviation in the context of normative behavior would be regarded as more "anomalous" than deviation in the context of behavior that is less typical. On the other hand, noise is an expected product of the variance of a process or of its dataset. Both anomalies and noise are 'outliers', since they deviate from normative behavioral patterns and represent low frequency behavior.

Criteria for distinguishing anomalies from noise depends on the application domain. An accessible example is a region's annual temperature plot. Surely the datapoints contain

noise in terms of daily variance with respect to the expected (average) temperature on each calendar day, with an average temperature abiding a sinusoidal pattern also influenced by latitude and elevation. However, extreme variance might be considered an anomaly (a freezing day in July), or likewise a mean value diverging from normal patterns (sustained 70-degree weather in December). But these also depend on the underlying explanatory variables: elevation can explain higher variance in temperature, and likewise latitude can explain changes in the observed average from the expected average temperature.

The dependence on domain factors means that criteria for distinguishing noise from anomalies are often not interchangeable between domains. For instance, a sudden increase in variance in a sinusoidal electrical power signal can often indicate an impending fault, and likewise sustained deviation of the observed average from some expected value could represent a catastrophic condition, such as the loss of a power generation resource (Phadke, 1993). Both of these signal conditions are anomalies in the electrical domain, but could be explained away as noise by the climate example.

As a result, the challenge of anomaly detection is that the definition of an anomaly varies by domain, and likewise distinguishing anomalies, noise, and outliers. Moreover, and as detailed by (Chandola et al., 2009), there is a lack of labeled anomaly data by which to compare and rank anomaly detection schemes, often leaving the task of anomaly detection to be solved for problem instances, rather than as a problem class.

As detailed by (Hodge et al., 2004), anomaly detection tasks can be broken into three categories: unsupervised, supervised, and semi-supervised. The emphasis of this work is the unsupervised case, for which unlabeled data is subjected to an unsupervised clustering

scheme by which outliers are detected as not strictly belonging to the derived clusters describing the data. Additional classification can also be performed on these outlier points. The second category is supervised anomaly detection, wherein one has access to labeled data and can model both normal and abnormal behaviors. Even with access to such data, these methods often also attempt to devise resilience to changes in the data distribution, due to the evolution of both normative data patterns and anomalies. The last category is semi-supervised "novelty" detection (Fawcett et al., 1999), whereby only normal behavior is learned from labeled input data, and non-normal data with respect to this model is flagged via an 'alarm' function of some form. Therefore, only the normal data class is learned, and abnormal behavior is learned as the system evolves.

This work fits within the unsupervised category of anomaly detection, whereby the input data is unlabeled, 'clusters' are learned to describe this data distribution, and anomalies are subsequently flagged via a metric intended to distinguish noise from true anomalies. This approach is often the most challenging for lack of labeled datasets, and likewise its extensibility across disparate anomaly detection domains. In addition to classifying non-fitting outlier data points with respect to derived models, the method further attempts to distinguish noise and anomalies among these outliers. The method can also be categorized as batch/stationary, since the input data is expected to completely describe the domain, and the resulting classifier does not evolve further after training.

2.4 Anomaly Detection in Process Data

Per section 2.2 on process representations, a trivial process-based anomaly detection scheme is to query the coverability graph per various decidability questions. 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 the process) can be verified by simply checking whether 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. Notably, this form of design analysis can 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 for different models. This is the most straightforward method for performing conformance checking, as defined previously.

However, in general traces encompass significant noise and choice (branching) behavior, making the simple consistency-checking procedure prone to false positives. Such a well-defined anomaly detection scheme only applies to scenarios for which the process model is formally prescribed for a process and available for evaluation. In fact, this points to an important design consideration for critical systems or high security contexts: the complexity of anomaly detection can be vastly reduced by designing process models (policies) such that detection is a trivial decision task. It is simply easier to design systems for which anomaly detection is straightforward, than to devise complicated anomaly detection methods after the fact for a process that was poorly designed, if designed at all.

As mentioned at the beginning of this chapter, 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 extracted 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 is because such

processes exist in a volatile context, and the process itself may even be non-stationary. For all but critical systems scenarios, it is rare for a process to be mindfully modeled and tracked according to the rigors of process mining. Instead, processes develop organically from changing staff, resources, and prototype *ad hoc* methods that evolve over time into the ingrained structure of a process. Software development provides an excellent example, wherein the rigors of process tracking are required and even well-enforced. But in reality, 100% conformance is regarded by developers and managers alike as unachievable textbook lore with every software bug submission and modification to the development process (the author's opinion). The formal study of these problems is addressed by "delta analysis", the task of empirically evaluating the alignment of processes with their prescribed rules and models, a formal description of which is provided by (Van der Aalst, 2005).

Formalizing the anomaly detection context of this work, one is given a graphical dataset as a log of unlabeled workflow traces *L* and must derive a model of normal system behavior with respect to which anomalies are detected. One also has the graphical process model *M* describing all traces in *L*, which can be mined from *L* using a process mining algorithm to convert an initial input log of trace sequences into their graphical representation. This is a canonical instance of unsupervised anomaly detection:

- Input: graphs $g_i \in L$ for i in $1 \ldots |L|$, where $g_i = (V_i, E_i) \subset \mathrm{M}$
- Output: $\{t_j\}$, $t_j \in \{1 \ldots |L|\}$ the set of flagged anomalous trace id's
- Optional output: $\{g_a\}$ the anomalous structures, where $g_a \subset g_i$

Viewing the input as a distribution over graphical substructures, the task is to flag anomalous substructures which are graphical components of one or more g_i. This formulation differs from

traditional unsupervised anomaly detection, since the input does not consist of fixed size input vectors $x_i \in R^d$ for some modest-sized *d*, but instead consists of graphs $g_i \subset M$. These graphs are components of *M*, but may contain labels or quantified loop executions, which complicates their reduction to fixed-size representations. Such structured and high-dimensional input is not amenable to traditional unsupervised anomaly detection methods.

Two alternative approaches are possible using data conversion, which are not evaluated in this work but bear mentioning because the conversion allows the use of canonical anomaly detection methods. Firstly, feature engineering could be applied via some function ϕ mapping the graphical inputs into fixed-sized real-valued vector "datapoints" $x_i \in R^d$, and thereby traditional unsupervised anomaly detection methods could be applied: $x_i = \phi(g_i),\ x_i \in R^d$. The function ϕ could be engineered manually or derived using deep learning or other unsupervised methods to extract graphical features, although it would likely be incapable of representing certain graphical features such as multigraphs or quantified loop executions.

A second alternative approach is to convert the unsupervised input into supervised data using pseudo-labels, thereby permitting traditional supervised classification methods. One way to do so is to encode the input graphs as vectorized binary adjacency matrices and append positive class labels to these fixed size vectors, and negative labels to their negation, effectively mapping the unsupervised input into a traditional supervised learning dataset. This encoding is detailed later in section 2.8. Interestingly, the input itself could contain anomalies, which would then be labeled as the 'normal' non-anomalous class. By underfitting models of 'normal behavior' to this data, the anomalies would be effectively filtered out of the classifier's predictions of 'normal' behavior.

Lastly, since traces are simply partially-ordered activity sequences, a third approach is to represent them as such to a sequential model such as a hidden Markov model or various recurrent neural networks. Two recent works using neural approaches are given by (Evermann et al., 2017) and (Tax et al., 2017). The former provides a method that works without an explicit process model using a Long Short Term Memory (LSTM) network architecture, and by leveraging sequential representations developed from highly successful neural approaches to natural language processing.

If such data conversions are employed to permit supervised anomaly detection, then traditional supervised binary classification methods can be employed and evaluated as such. If conversion to unsupervised anomaly detection is used, using the mapping $x_i = \phi(g_i),\ x_i \in R^d$ mentioned above, then traditional unsupervised classification methods can be used: *k*-nearest neighbors, neural networks, clustering methods, decision forests, and many other methods (Omar et al., 2013). These approaches represent the traditional framework of deriving model and then defining a metric of "normative" behavior by which anomalies are determined.

Recent work uses a framework wherein the classifier is derived from the learning process itself, by training underfitting or overfitting models (Dietterich et al., 2015). By training deliberately underfitting models, it is expected that anomalous datapoints will not be contained by the model and are classified as anomalous. Overfitting approaches are less straightforward, but an example is the Repeated Impossible Discrimination Ensemble (RIDE) approach by (Senator et al., 2013). In this approach, all datapoints are randomly assigned a 0/1 binary class label, a supervised model is trained, and each datapoint is scored via its probability of belonging to class 1: $score(x_i) = |0.5 - P(y_i = 1)|$. This process is repeated and each datapoint's score is

accumulated over many iterations. At the end of this process, the datapoints can be ordered by score, providing a ranking over the anomalousness of the datapoints.

In contrast to these methods and frameworks, graphical compression methods operate directly on the unmodified input representation $g_i \in L$ to generate a descriptive compressing set of graphical features representing the entire set of graphs, and therefore lend themselves directly to the problem of process-based anomaly detection. Anomalous traces can subsequently be classified as outliers or statistical anomalies with respect to these representations, via their anomalous structural components. This means that unlike the previously described methods, graphical compression methods are capable of attributing anomaly classifications to anomalous structural components and returning other structural patterns in a human-readable form. In this manner, graphical methods can be used to detect anomalies, by applying these methods to graphical process representations. In this work, graphical compression methods were used 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. Hence graphical compression methods lend themselves directly to the analysis of processes and to the detection of anomalies.

Graphical representations of anomalies must then be characterized. In contrast to other anomaly detection applications, graphical anomaly detection encompasses high-dimensional structural features of graphs. A comparison is given in figure 2.3, and a similar example can be found in (Akoglu et al., 2015) p. 3, figure 1, as the comparison is universal. The left plot demonstrates a typical statistical anomaly detection setting in a modest 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 an outlier. The right plot shows a graphical representation of email communities within the Enron email dataset (Shetty and Adibi, 2004), evaluated by the author using the igraph library (Csardi, 2006). There are abundant methods for detecting outliers in this static graphical setting, such as disconnected/isolated components, vertex centrality metrics, community clustering metrics, and so forth. The distinction between these examples is that the left is pointwise and usually entails the derivation of continuous decision surfaces, such as the ellipses shown, whereas on the right graphical anomaly detection encompasses structural characterizations. The depth of network-oriented methods is explored in (Easley and Kleinberg, 2010).

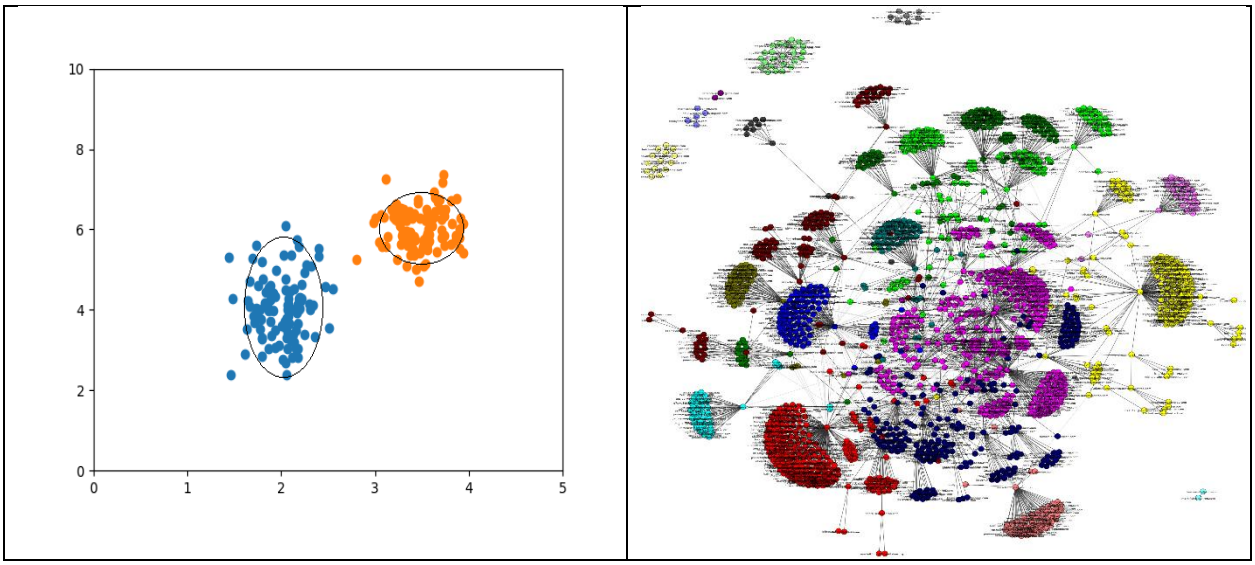


Figure 2.3: Pointwise and graphical anomaly detection settings. At left, a contrived pointwise setting. At right, Enron email data communities detected and plotted using the igraph library, where disconnected components and inter-cluster vertices could represent anomalies or outliers.

The data model shown in figure 2.3 (right) is what (Akoglu et al., 2015) refer to as the "interlinked objects" model, which emphasizes the static structural characteristics of graphical patterns. In contrast to the static graphical setting, this work is concerned with structural anomalies in dynamic process data. In this setting, 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 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* to contain regular structural properties, which is satisfied for processes since they exhibit highly regular structural patterns of higher order. By contrast, an email network like in figure 2.3 would be less appropriate, since in that setting *D* would consist of erratic point-to-point communications with only (or mostly) first-order regularity, but few recurring substructures of higher order. However, it is important to point out that this implies 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 other extreme are datasets for which higher-order regularity is to be expected or even mandated, such as manufacturing processes or organic molecular structure.

For instance, in contrast to the first-order, point-to-point quality of email communication, imagine 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, imagine our interest is restricted to only a distinct subset of the colored vertices, representing some sub-process within this system, as shown in figure 2.4. The mined process model is shown at the top right, where transitions have been removed and activities are directly linked to one another. This erases the semantic constraints of a formal process model but is sufficiently expressive since we are solely concerned with recurring activity-based substructures. This representation is used throughout this work, since it sufficiently describes graphical structure, the domain of this work. The hypothetical input traces from which this model was derived are shown at left, and the 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.

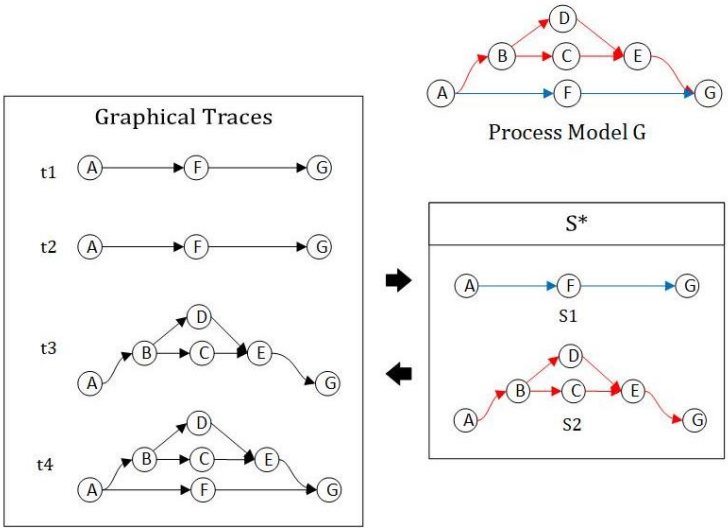


Figure 2.4: A graphical representation of process data.

In the process setting, one assumes that the edges in this graph encompass relations between the activities or resources of a process, but further, that the process data also contains such structural regularity of arbitrarily higher-order. In simple terms, one expects that this graphical data contains regularity in terms of repeated substructures beyond simple first-order Markovian (edge) statistics or static structural features. These represent sub-tasks within a process, such as a customer modifying their online shopping cart, or the checkout subprocess consisting 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 substructures, which are overlapping subsets of vertices and edges. This property can be exploited to learn more complex patterns of the underlying process than first-order

models and other network scientific approaches based on local features such as vertex or edge properties. The remaining question is how to efficiently discover these patterns, which is developed in the following sections.

2.5 The Applicability of Graph Compression Algorithms

The purpose of this section is to introduce the context for graph compression algorithms, and to define the problem of compressing a dataset of graphical traces assumed to have been generated from some underlying graphical process model *M*. The ability to infer the graphical structure of a process from trace data may seem to possess limited application outside of process mining, business process management, or similar operations fields. However, the induction of graphical structure from data encompasses many classical formal problems spanning machine learning, planning, and artificial intelligence.

Consider a planning problem of a robot navigating a two-dimensional discrete state space $s_{x,y} \in (x \times y),\ x, y \in Z^+$, using elementary reinforcement-learning formalisms. Assume the action set contains four discrete actions $a = \{left, down, right, up\}$, the environment is known, the transition model $\pi(s_t|s_{t-1}, a_{t-1})$ may be known or unknown, and the robot's task is to navigate to some positive reward location while avoiding obstacles with negative rewards. These problem formulations are amenable to a wide variety of approaches in a class of algorithms modeling the sequential one-step dynamics of discrete action sets, for relatively modestly sized action and state spaces. Many canonical solutions are rooted in Bellman equation formalisms or Monte Carlo methods, an overview of which is provided by (Sutton and Barto, 1998).

Now instead consider that the rewards and system dynamics are determined by *k*-step bounded dynamics, where the robot achieves goals only by executing specific partially-ordered sequences of actions of length i, for $i \leq k$. For example, $(up, up, down)$ or

$(up, right, down, left)$; additionally, these action structures may be interspersed with actions for unrelated concurrent tasks. Moreover, tasks may themselves entail parallel activities, but their independent pathways cannot always be separated without prior knowledge of the underlying process. These representations mimic real-life tasks, where actions possess long-range interdependencies, and tasks recursively decompose to subsets of actions, or "subtasks", which do not necessarily entail one another despite their sequential adjacency. Moreover, such tasks often can only be described graphically, by process model formalisms such as Petri Nets (Peterson, 1981).

These structured environments violate the clean, Markovian one-step sequential dynamics required by many classical reinforcement learning formulations, and result in exponential search complexity. For instance, the long-range activity set of action sequences in the above example, even excluding parallelism, is exponential in the number of activities, $|a| = 4$, and the bound on sequence lengths k, resulting in 4^k possible sequences. This is an intractable space even for this modest activity set, and without any assumptions about the complexity of the action space, which is inherently graphical and non-sequential.

These problems are the domain of classical planning problems, such as the block world domain of (Nilsson, 1980), for which a complexity analysis is provided by (Gupta et al., 1992). More accessible examples are the games of Go and Chess, with their high branching factor and multiple game strategies composed of long-range action dependencies. In these problems, the action space, action-sequence space, state space, or combinations of these are intractably large for traditional, sequential learning formulations without search heuristics or cleverly engineered state space reductions. Current approaches often implement techniques such as Monte Carlo Tree

Search (Brown et al., 2012), which still entail intensive search behavior before upper confidence bounds on action-value estimates yield satisfactory performance.

Graph compression addresses these problems heuristically by extracting the subgraphs characterizing advantageous behavior. Intuitively, a highly-compressing graphical pattern discovered via compression is also a task-meaningful pattern. For instance, in a highly complex combinatorial space such as chess, sub-graphs of useful actions could be learned via long sequences of user actions. Likewise, non-adversarial everyday discrete tasks such as performing home chores or driving may be learned from examples characterizing the underlying graphical model of the task. Given large amounts of such user data for this and similar planning problems, repeated subgraphs can be extracted and can bootstrap learning algorithms to bias their activities toward advantageous structural features. Conversely, the extracted information can be used for tasks such as anomaly detection.

Thus, many process-oriented tasks can be learned heuristically by learning graphical features of processes. These representations can then be used to efficiently learn complex behavior from compositions of subgraphs representing subtasks within a domain. As a result, methods for compressing and extracting structural patterns from graphical data have general application in machine learning, planning, and artificial intelligence, and encompass generic task-learning.

2.6 The Optimization Problem of Graphical Data Compression

The problem of graphical trace compression is to maximally compress a set of graphical data representing directed, possibly cyclic traces generated from an unknown underlying graphical model G. In the domain of process mining, these models typically have regular, compression-favorable structural properties, such as defined *begin/entry* and *end/exit* points for

traces, modest overall average degree, few or zero cycles, and generally, if not always, behaving like directed, acyclic graphs (DAGs). The field of process mining provides many algorithms for mining G from some input log L; G can then be used to convert a log of partially-ordered traces into a set of corresponding graphical traces $T_{L,G}$ (hereafter simply T).

The goal of graphical compression is to reduce a set of graphical traces T to a minimal collection of prototype subgraphs $S = \{s_1, \ldots s_k\}$, or edge subsets, maximally compressing T:

$$Input: a\ graphical\ trace\ log\ T, of\ size\ d = |T|$$

$$Objective: S^* = \{s_1, \ldots s_k\}\ \ for\ min_k(S|T)$$

This objective defines the search for the minimum *k*-length set S, given log T, reducing all of T to a minimal subset of subgraphs maximally compressing *T*. Once the set S is found, each trace t_i can be encoded as a *k*-length binary vector b_i indicating its member subgraphs, and thus the trace can be encoded and decoded via S and b_i. Thus, a lossless compression method can be devised by finding the minimal S^*, converting each trace to a bit vector indicating its subsets in S^*, and transmitting these vectors along with the edge-subsets S^* by which to decode them.

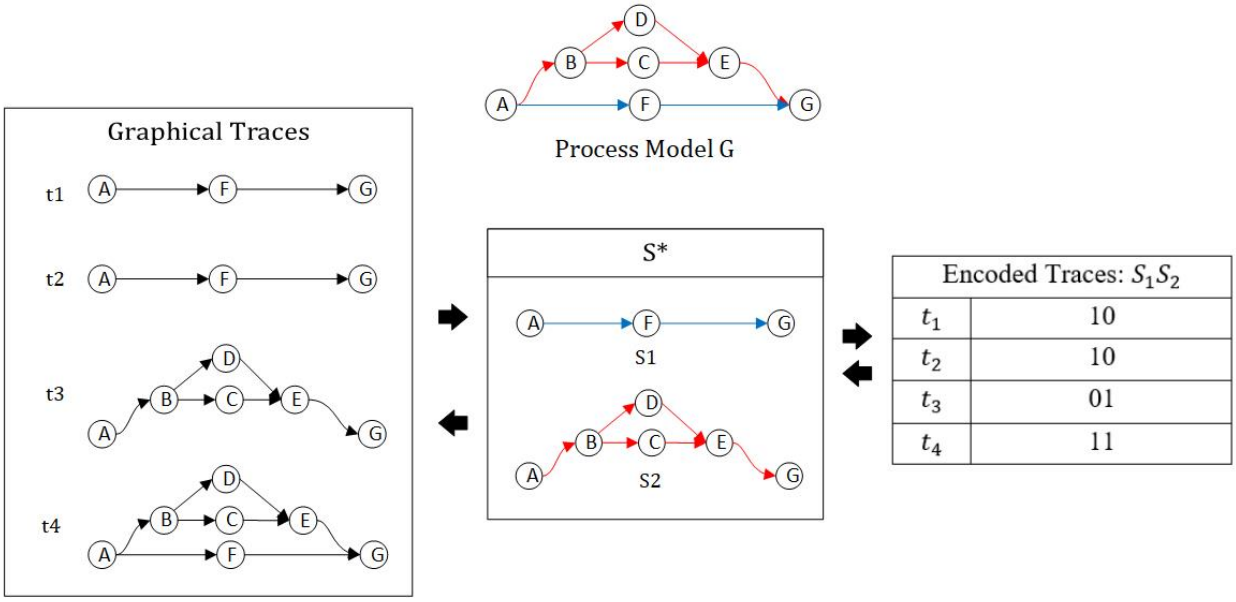


Figure 2.5: Graphical encoding and decoding. Trace set T of four traces is shown at left, compressing set S (center), and the encoding vectors for each of the four traces (right), where a '1' indicates the presence of S_i. S* provides the encoding/decoding dictionary.*

The general method is illustrated in figure 2.5, from the example in the previous section. Assume that any trace executes only one (or both) of the colored paths shown as the process graph G. Then the colored components, shown in red and blue, form substructures by which the entire log of traces may be compressed (encoded), by converting each trace to a binary vector. The colored substructures provide the mapping to and from these vectors.

This is also a very simple instance of graph grammars and graph parsing. Figure 2.5 provides an example of a simple graph parsing output, where the encoded binary vector high-bits indicate production rules by which to convert any trace back into its graphical form via the production rule set *S*. The similarity is coincidental, since graph grammars encompass more complex recursive rules than this example. However, many graph grammar induction algorithms have been published, given their relevance to tasks such as compiler construction and various mathematical problems. An example of heuristic graph grammar induction is given by (Jonyer et

al., 2001) using a similar approach to the one demonstrated later in this work for dendrogram-induction from process data.

2.7 Simplified Problem Formulation for Complexity Analysis

The formal problem of finding the compressing set S reduces to iteratively selecting columns from the vectorized binary adjacency matrices of all traces in the directed graphical data. For a dataset in the form of a trace log T of size $d = |T|$, each trace t_i is a subgraph g_i of the super-graph $G = (V, E)$. G represents the graphical behavioral model inclusive of all traces, which is the union of all edges in T. Each trace can be represented as an adjacency matrix A_i whose rows and columns are composed of the vertices of G. Concatenating the rows of this matrix in row-major order (the vectorization of A_i) gives a binary vector x_i whose non-zero components indicate the directed relations (edges) present in the trace, as shown in figure 2.6. This representation is solely for illustration, since this binary representation only captures first-order structure. For instance, it does not quantify edge transitions, such as to quantify loop executions. However, it is sufficient to describe graphical structural properties via labeled, directed graphs with unweighted, unattributed edges. It is also intended to motivate the view of graphical trace data as a distribution of substructures, rather than individual vertices and edges.

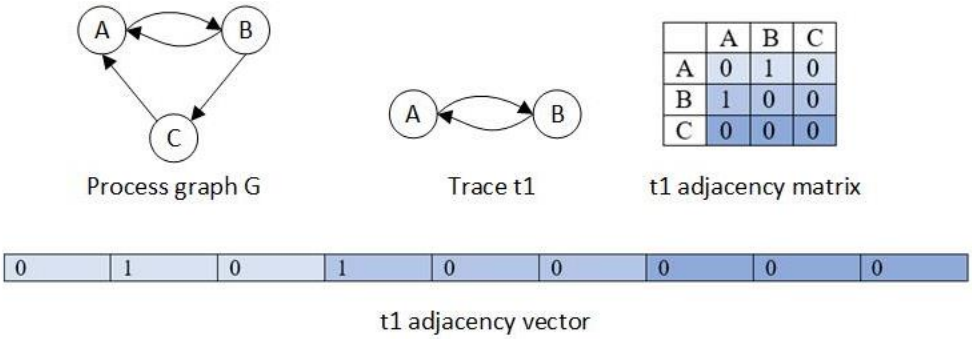


Figure 2.6: Shown above are the graph G, the process model inclusive of all traces. Trace t1 is an example subgraph of the execution of this process, with its associated adjacency matrix and adjacency vector representation.

The entire set of traces can be converted to a matrix $\boldsymbol{A}_L$ of adjacency 'vectors' x_i like the one shown in figure 2.6. A hypothetical matrix $\boldsymbol{A}_L$ is shown in figure 2.7, whose shaded regions are explained later. This data representation is illustrative because it demonstrates the dimensionality of the input space, which for a directed, complete graph *G* is quadratic in the number of vertices, i.e., $|V|^2$. Hence this quadratic memory complexity would only be computationally feasible for modest-sized graphs.

	Edges, $v \times v \in V_G$								
Traces		A-A	A-B	A-C	B-A	B-B	B-C	C-A	C-B	C-C
	t1	0	1	0	1	0	0	0	0	0
	t2	0	1	0	0	0	1	1	0	0
	t3	0	1	0	1	0	0	0	0	0
	t4	0	1	0	1	0	0	0	0	0
	t5	0	1	0	1	0	0	0	0	0
	t6	0	1	0	0	0	1	1	0	0

Figure 2.7: A data representation consisting of vectorized adjacency matrices. Every trace's adjacency matrix is vectorized into a row. The light blue regions represent column and row selection, and the darker blue regions are their intersection.

Finding the maximally compressing subgraph within this data representation reduces to finding the largest subset of columns containing all 1's and encompassing the greatest number of rows. Such a column set of edges represents a subgraph $s_{v\times v} \in G$, if it is a connected component of G. The size of the column set represents the size of the subgraph s in edges, likewise, the number of rows is its frequency.

In this manner, a maximum-compressing subgraph can be found at each iteration by searching the columns of this matrix for the largest collection of all 1's. The shaded columns in figure 2.7 depict such a search; the highlighted rows are rows containing all 1's for the given column selection. For any fixed choice of columns, all d traces must be traversed to count the number of rows for which the conjunction of this fixed choice of columns evaluates to true, and there are $\binom{|E|}{i}$ possible choices of columns. Defining the set of all edges in graph G as $E =$

$\{(u, v) \in G\}$, the set of all vertices $V = \{v \in G\}$ and using the identity $\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$ (Cormen et al., 2001), the complexity of this procedure is given by:

$$\sum_{i=1}^{|E|} d \binom{|E|}{i} = d \sum_{i=1}^{|E|} \binom{|E|}{i} = d(2^{|E|} - 1)$$

Equation 1: Search complexity

Thus, this column selection procedure is exponential in the number of edges, which in the worst-case for a directed, fully-connected, and non-reflexive graph, is:

$$|E| = |V|(|V| - 1) = \Theta(|V|^2)$$

Hence,

$$d(2^{|E|} - 1) = d(2^{\Theta(|V|^2)} - 1)$$

Equation 2: Bounded search complexity

This search procedure illustrates the problem's structure and brute force complexity for only a single compression iteration. But it is flawed, as it searches needlessly over all combinations of subsets of edges of size *i*, whereas the only relevant search is over the connected components of G. Graphical problems frequently involve sparse graphs, so complexity can be reduced by restricting iterations to the subsets of columns representing connected components. For special kinds of graphs like DAG's, components can be enumerated using basic, elementary graph-search procedures. Lastly, real graphical data typically has high redundancy, such that d can also be reduced significantly by a de-duplication strategy of storing each unique row with its frequency, and further by omitting columns of all zeroes, such as the 'C-B' and 'C-C' columns in figure 2.7.

Critically, this search characterization overlooks how subgraph size and subgraph frequency affect the optimality of the resulting set S. The search over columns (substructures)

and rows (frequency) introduces a compression tradeoff between size and frequency: it may be easy to find a very large and infrequent subgraph, or conversely a very small but frequent subgraph (such as a single edge). Consider two candidate prototypes, s_i and s_j. Let s_i have $size(s_i) = 4$ and $frequency(s_i) = 3$, and let s_j have $size(s_j) = 3$ and $frequency(s_j) = 4$. Which of s_i or s_j should be chosen to obtain optimal compression S^*? How do criteria of $size$ and $frequency$ affect the optimality of the resulting set S? A straightforward heuristic is to select the subset with the largest sum of 1's as a measure of information gain.

Worse yet, dependencies exist between the selection of subgraphs since candidate subgraphs for compression are not disjoint. The selection of a prototype s_t at the tth iteration can affect which candidates are available in subsequent iterations, which may impact the compression of the resulting set S. Optimal compression is defined as minimizing the description length of the trace-graph codes sufficient to losslessly decode all trace-graphs from their encodings via S. Due to these tradeoffs, optimally encoding the subgraphs requires correctly making the complete sequence of decisions per the size and frequency of each prototype subgraph. This problem is akin to bin-packing (Korte and Vygen, 2000), an NP-hard combinatorial problem, but harder due to the overlapping dependencies between prototype selection. Loosely, each prototype's size and frequency define each object's abstract dimensions, while the objective is to fit as many of these objects as possible into the smallest bin, thereby globally minimizing $|S|$. The similar sequential problem of generating the smallest context-free grammar generating a sequence of symbols (e.g., traces) is also known to be NP-hard (Charikar et al., 2005). Fortunately, the optimal formulation of this problem is not the subject of this work. But understanding the problem structure of the simple brute-force graph compression formulation given in this section is important when designing alternative methods.

2.8 The Heuristic View of Graphical Data Compression

In a relaxed version of the trace-graph compression problem the encoding need not be optimal but may instead heuristically generate an approximately optimal encoding S' much faster than the optimal S^*. Such procedures can still be lossless, such that any trace can be completely reconstructed from its encoding and the set S'. Many heuristics are possible, since the task is relaxed to that of iteratively finding frequent subgraphs under some compression criterion, such as favoring the size or frequency of graphs. Many of these heuristics can be viewed as greedy methods for reducing the $2^{\Theta(|V^2|)}$ column selection search or for reducing the leading constant d in equation 2. An overview of such methods can be found in (Maneth et al., 2017). Many graph problems involve graphs with vertex attributes or other additional information that may also be incorporated into the information-theoretic definition of their encoding.

Notably, the adjacency-matrix based data representation in the previous section is amenable to a wide range of supervised and unsupervised learning approaches. Unsupervised approaches, such as neural autoencoders, hold great promise in terms of automating graphical pattern discovery. This framework trains a neural network using the input as the target output. Using various training and architectural strategies, these networks learn the hidden structural patterns of the data, by which normative and anomalous patterns can be determined. A recent example is given by (Nolle et al., 2016), in which the authors used a denoising autoencoder model for both anomaly detection and normative pattern discovery. The authors presented the traces to their networks as linear activity sequences, the canonical process mining trace representation, rather than trace adjacency matrices. They reported that their method perfectly split the trace log into normal and anomalous traces. Similar work is possible using recurrent neural networks (Ellman, 1990) (Jordan, 1986) by presenting the traces to the network as linear

activity sequences, although the hidden layers of such models are difficult to interpret. More recent neural models combine recurrent networks and autoencoders for a variety of similar structured input/output tasks, such as the seq2seq model of (Kalchbrenner et al., 2013) or the long-short-term memory (LSTM) autoencoder framework described in (Srivastava, 2015).

Traditional supervised learning models can also be easily adapted to unsupervised pattern discovery, and often provide simpler and more accessible model characteristics. By appending a +1 to each binary adjacency 'vector' x_i as a dummy target "output" for a learning model, the unsupervised dataset $\boldsymbol{A}_L$ can be mapped to a supervised learning representation. Each vector can likewise be replicated by its negation, possibly with additive noise, to generate a semi-synthetic supervised learning dataset $\boldsymbol{A'}_L$ dividing the input space into two classes: positive examples, and 'negative' synthetic examples sampled outside the set via some distribution facilitating a specific learning model. Some distant examples of such data extension/generation strategies are the negative sampling used by some implementations of the word2vec algorithm (Mikolov et al., 2013), various structured learning algorithms like the DAgger algorithm (Bagnell, 2015), or (very distantly) the generative adversarial networks of (Goodfellow et al., 2014).

The benefit of such a supervised data representation is that many supervised learning models have been developed, especially generative ones, by which normative patterns or other model parameters can be learned to perform secondary tasks like anomaly detection and normative pattern extraction. The simplest approach is to run linear regression on the preceding semi-synthetic dataset $\boldsymbol{A'}_L$. The result output is a weight vector w^* whose non-zero components correspond to the collection of edges which maximally "compress" the data by minimizing the mean-squared error (MSE) loss. The corresponding columns would then be removed from the input data, and the procedure would be re-run on the remaining examples to find the next set of

such edges, and so on, until the data is completely compressed. Notably, the edge collection found on any iteration might not represent a connected subgraph, but regularization strategies could be devised to bias the learning algorithm toward connected, rather than disconnected edge subsets.

2.9 SUBDUE

While many data representations and strategies could be explored, this work's primary focus is on the SUBDUE method for discovering the maximally compressing components of graphical data. In contrast to matrix representations of graphical data, SUBDUE is search-based and focuses on the vertex perspective to search for compressing substructures. In this manner, SUBDUE proceeds by "growing" candidate substructures within some search beam of size *b*, maintaining only the *b* most highly compressing components in the beam at any time. Compression is measured by the reduction in the description length of the data with respect to compressing components, via the minimum-description length (MDL) principle (Cook and Holder, 1994). This metric can encompass additional information, such as vertex labels, which is beyond the scope of this work since activities are treated as single symbols, each activity is represented by a unique vertex, and every vertex label is the same effective description length.

From the SUBDUE perspective, compressing components are found not by solving a brute-force global search over edges, but rather by growing compressing components around the neighborhood surrounding promising vertices. This work's focus is on structural compression, hence SUBDUE's ability to efficiently discover compressing patterns maps directly to the compression of process data to its most relevant structural features, and thereafter the discovery of anomalies. SUBDUE performs well for anomaly detection because, when using this process-feature discovery mechanism to compress process data, eventually all that remains are the

"unusual" poorly compressing components of the data. Additionally, the beam size provides a tunable hyperparameter for constraining search complexity, where a larger beam size allows SUBDUE to discover more compressing patterns but increases search complexity.

2.10 Previous Work

The SUBDUE graph-compression method mines normative patterns from graph data, working as a graphical feature detector. It was previously used for knowledge representation systems (Djoko et al., 1997), and more recently in security applications for intrusion detection (Noble and Cook, 2003). Using SUBDUE as a process mining tool was successfully performed by (Diamantini, 2013) and more recently by (Genga et al., 2016), 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 (Akoglu et al., 2015).

GBAD (Eberle and Holder, 2007) builds on SUBDUE to provide anomaly detection capabilities, particularly within the immediate proximity of normative graphical patterns. This is appropriate for safety-critical and security contexts possessing some underlying process model by which normative patterns can be assumed to have a ground-truth behavioral policy, but less so when there is no such policy or model. An application is given by (Holder and Eberle, 2009), in which GBAD was used for insider threat detection by combining three anomaly detection algorithms.

Process anomaly detection focuses primarily on the mining process itself and on trace-scoring schemes. Scoring schemes were detailed by (Bezerra et al., 2009), by which work traces were replayed on a discovered model, assigned a numeric fitness score, and anomalies were flagged based on a discriminative threshold. Bezerra's work (2013) 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.

Summarizing, this chapter was theoretically-oriented to establish the core concepts of this work that follow in the applications found in the next two chapters. Process mining formalisms and models were introduced, including their graphical representation and anomaly detection methods. Where this work deviates from previous approaches is in viewing process log data as a distribution of substructures, whereby graphical compression methods provide an effective means of extracting features and detecting anomalies, as were discussed. This was then used to introduce the context for graphical compression methods, albeit an extended interpretation of their application to machine learning, followed by a brute-force examination of the problem structure of graphical compression and its complexity analysis. Subsequently, the brute force method gives way to heuristic graph compression methods, of which SUBDUE is an effective example. The next chapter refines the problem terminology and statement of applied process anomaly detection, followed by the proposed approach in chapter four and its experimental evaluation in chapter five.

CHAPTER THREE: PROBLEM DEFINITION

Whereas the previous chapter characterized the formal problem of graphical compression, this chapter and the remainder of this work involves its application to process data and anomaly detection.

3.1 Terminology

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

- Process model: A graph with vertices representing activities, and edges representing one-step transitions between activities. Processes can contain many constructs representing linear and non-linear behaviors, and a variety of notations and languages have been defined over the space of process models. A canonical example is the Petri net (Petri, 2008), shown at left in figure 3.1. 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 connecting activities (places) directly. Formally, the control flow graph is defined such that if there is a path $\{(v', t'), (t', v'')\}$ on the original Petri net model, then there is an edge (v', v'') in the control flow graph. This removes some semantic transition information from the model but is sufficient to model and detect structural regularity and anomalies.

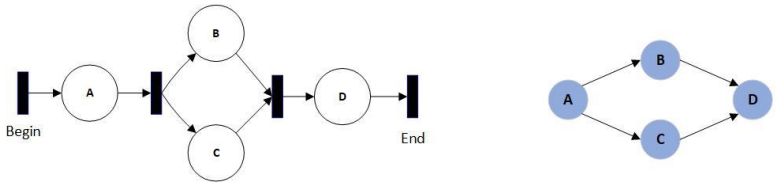


Figure 3.1: A Petri net process model and its simpler control flow counterpart.

- 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 vertex on a process model. These are represented as a string composed of letters representing the activities traversed.
- Workflow log: A set of workflow traces in some canonical form, such as eXtensible Event Stream (XES) [1], and for which various measures are taken to mitigate properties of noise or incompleteness. Synonymous with 'trace log', 'process log', or simply 'log'.
- Process miner: Any algorithm constructing a process model from a workflow log, and often incorporating 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 but 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' might be workflow traces from some model, such as the one 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'. The primary challenge of process mining algorithms is disambiguating these partial-orderings, applying rules and heuristics to generate models with desired properties of complexity, specificity, and generality. The enormous space of possible graphical models defined over a set of partially-ordered traces constitutes the primary search problem for these algorithms.
- 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 (Russell et al., 2006). See also (Kiepuszewski et al., 2003) and www.workflowpatterns.com, a joint university effort of (Van der Aalst et al.).

- 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 (Van der Aalst, 2011).
- Inductive miner: A process mining algorithm capable of generating the most general, 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, by applying the minimum-description length (MDL) heuristic, returns the top-k most compressing sub-graphs of the collection (Holder, 1989).
- 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, such as indicating that some trace includes two separate compressing subgraphs. Such a construction can be represented multiple ways: directed or undirected, and with or without edge weights. Devising edge/vertex quantifications of this representation is a crucial task, since it facilitates how the representation can be used. An example is quantifying edge weights per trace counts between two compressing subgraphs, by which regularity statistics can be measured, and thus unusual behavior detected. The term "dendrogram" is used here only for its hierarchical implications, since this graphical representation can contain vertices with multiple parents, whereas most dendrogram descriptions only allow disjoint single child-parent relationships. An explicit example dendrogram is given later and further refined.

An in-depth overview of process mining terms and methods can be found in (Dumas et al., 2005) and (Van der Aalst, 2011).

3.2 Process Anomaly Detection Problem Definition

In this work, the input is a log L, consisting of traces generated from some unknown process model M^*, a graphical process miner $mine$, and a graph-compression method $compress$. The formal problem is to mine a graphical process model $M := mine(L)$ by which to convert L into a collection of graphs $G_{L,M}$ via M (a preliminary step), and then to iteratively mine the normative patterns $P_{L,M} = \{p_0, .. p_i\}$ of G_L using $compress$s. The patterns P can then be used for post-processing tasks, such as anomaly detection: given P, identify and return anomalous traces $T_{anom} \in L$.

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
detect: An anomaly detection method

Output $P_{L,M}$: The set of normative graphical patterns $P_{L,M} = \{p_0, .. p_i\}$
T_{anom}: A set of anomalous traces in L

1. M_L = *mine*(*L*) #mine the graphical process model
2. traceGraphs = convert(M_L, *L*) #convert the log traces to graphs, using *model*
3. $P_{L,M}$ = *compress*(traceGraphs)
4. T_{anom} = *detect*($P_{L,M}$, *L*)
5. return $P_{L,M}$, T_{anom}

This problem definition is overly verbose and could be reduced to solely outputting the set of anomalous traces T_{anom} to reflect the original problem statement. Outputting $P_{L,M}$ reflects that in many cases an anomaly detection method can also provide normative pattern information along with the anomalies it discovers, though this is not always the case. Akoglu et al. (2015) refer to similar application of anomaly detection for normative pattern extraction as "data cleaning" (p.2), whereby pattern information could be used for model refinement, for instance. Likewise, both *compress* and *detect* could be reduced to the single task of anomaly detection.

3.3 Evaluation Metrics

Based on this problem definition, and 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 f_k-score. Alternatively, one could ignore the binary classification problem of anomaly detection and evaluate a solution in terms of $P_{L,M}$ to determine how well a method compresses a log according to a compression-evaluation function. This is the compression-evaluation criterion, a subject of future work requiring different experimental designs for evaluating log compression performance with respect to a known model describing the distribution of structures. But it is important to

distinguish these formal tasks: anomaly detection and compression-evaluation. The latter is an important task for evaluating machine learning approaches to structural learning in terms of the compactness of their learned representations.

Table 3.1: Binary classifier outcomes for anomaly detection

C1= anomaly (+)		**Actual Class**	
C2= normal (-)		C1	C2
Predicted	C1	True Positive (TP)	Type I Error (FP)
Class	C2	Type II Error (FN)	True Negative (TN)

Accuracy, precision, recall, and f_k-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 be used to assign costs to the matrix of possible outcomes for a binary classifier. Let $P(*)$ denote the distribution of errors for a given classifier given the table above, $V(*)$ the positive or negative cost per each outcome, the cost function is given by the expected value of the classifier:

$$V(TP)P(TP) + V(FP)P(FP) + V(FN)P(FN) + V(TN)P(TN)$$

The importance of this expression is that anomaly detection 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 usually rare, 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, imagine evaluating a classifier which simply classifies every trace as "normal" on data for which 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. Thus, evaluation metrics in security or critical system contexts must incorporate cost analyses to be effective for their domain.

Precision, recall, and f1-measure also remedy this defect by measuring the true-positive class (anomalies) explicitly. For a given application, however, defining and evaluating the cost function is likely the most important anomaly detection evaluation method. 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. For instance, in a customer service center process the type II misclassification of an anomaly may be an ordinary and low-cost occurrence (the annoyance and loss of a customer), whereas in a mission critical process such as an aerospace mission, the cost of a type II misclassification could equal the cost of the entire mission (the loss of a spacecraft).

3.4 Problem Complexity

The complexity of finding anomalies is a function of their structural complexity. Whereas many anomaly detection works focus on straightforward statistical events, this work deals with graphical anomalies: the unusual modification of graphical structure. The term "unusual" refers to the fact that anomalies are defined in the context of normative patterns, hence the complexity of finding graphical anomalies is typically dominated by the complexity of finding normative graphical patterns, which was established in the previous chapter. That is, even for the case of unquantified/unweighted edge data containing only binary structural information, the optimal brute-force search for a single normative pattern iteration was shown to be $d(2^{\Theta(|V|^2)} - 1)$ where d was the size of the log $d = |L|$, and V was the number of vertices of the graphical model.

Thus, in the problem domain of process anomaly detection, V is the number of activities in a graphical process model, and therefore complexity is most strongly determined by the size of

a process model. Again, the complexity bound is an over-estimate, which simple heuristics can overcome. Additionally, processes in this domain are expected to be of modest size, consisting of fewer than 50 activities on average. Data containing models larger than this would likely need to be decomposed to specific processes, such as the complete log data of an entire department would need to be decomposed to its sub-processes or teams. Oftentimes the number of activities in process data is view-specific to how "activities" are represented and what they represent in a domain. For instance, within some data one might choose for activities to represent event types versus finer event names, such as a patient's transition between the different areas of a hospital versus the fine-grained treatment tasks associated with their complete passage through a hospital. The former evaluates traces with respect to an organizational view, and is likely more appropriate, whereas the latter is very fine grained and will likely contain lots of noise and fewer recurring patterns. Another good example is process modeling an operating system for which activities represent function calls. A typical operating system contains thousands of functions, so it would be more appropriate to model its interfaces independently, each of which likely implements a modest number of high-level functions.

In closing, the complexity of anomaly detection in processes using graph compression reduces to the size of the underlying model, in terms of the number of activities (vertices) it encompasses. In the worst-case, this bound is exponential in the number of activities, but most processes will be below this bound due to their regular structure. Real-world processes usually meet these requirements, having fewer than 50 activities or so, although this threshold is loose and view-specific, and different views can be selected to accommodate better task encapsulation. Heuristic normative graphical pattern methods will surely vary in terms of how accurately and

how quickly they discover normative patterns, and subsequently anomalies, as processes grow in size.

CHAPTER FOUR: PROPOSED ALGORITHM

The previous chapter specified the problem definition of anomaly graphical anomaly detection within process data, detailing its performance evaluation and complexity. This chapter is implementation specific and explicitly defines the proposed approach. This requires detailing the generation of the dendrogram, a hierarchical description of the distribution of substructures within some process data, and likewise deriving an anomaly detection metric, as follows.

4.1 Proposed Method Overview

Our method for anomaly detection in processes decomposes to four tasks: converting a trace log to a collection of subgraphs via a mined process model, extracting descriptive normative graphical patterns of the log with respect to this model, compiling these patterns into a dendrogram of graphical features, and lastly detecting outliers and anomalous behavior. The overall data-flow of the approach is shown in figure 4.1.

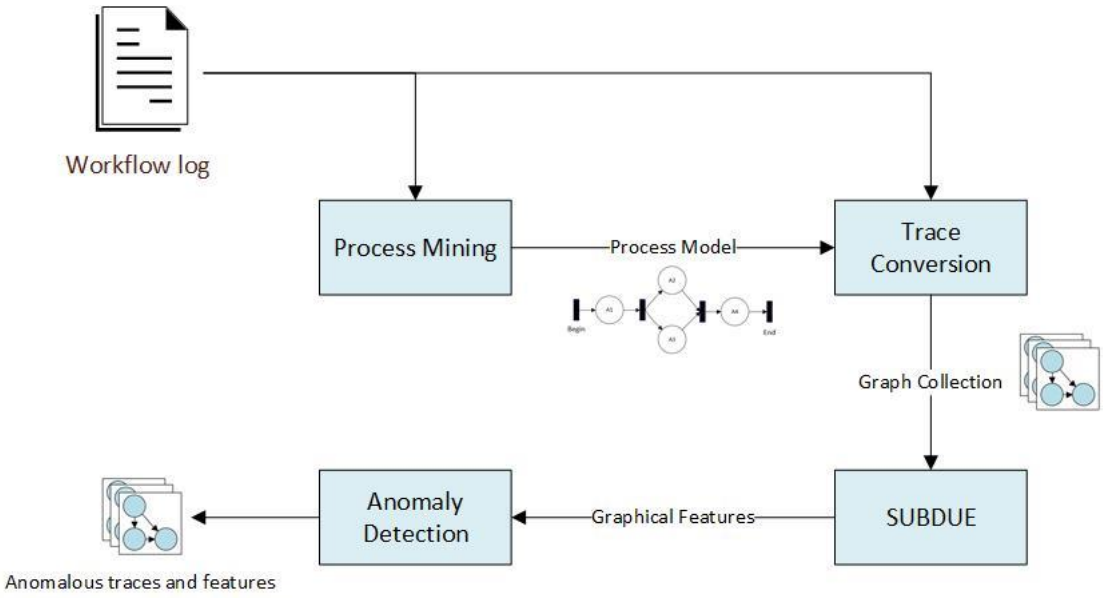


Figure 4.1: Data-flow model of the approach.

For the first task, the Inductive Miner (Leemans et al., 2013) was suitable for mining the most general graphical process model described by some log. This model is typically overly-inclusive,

whereas the second and third tasks discover the patterns and features precisely relevant to the log. For this, the SUBDUE graph-compression method (Holder, 1989) was used to discover normative behavioral patterns, subsequently enabling anomaly detection.

The workflow extends the discovery of process characteristics irrespective of prior constraints, such as a prescribed process model or a formal PAIS. It is tolerable for the process mining algorithm to produce overly general models, since significant graphical features are extracted in post-processing, rather than within the mining algorithm itself. Decoupling the feature extraction and mining steps offers greater tuning for noisy or poorly structured process data. This facilitates more realistic and informal "spaghetti" model scenarios in which processes are ad hoc and highly unstructured, such as enterprises, communication networks, distributed systems, software system executions, or natural processes. For instance, the mining algorithm or the graph-feature extraction components shown in figure 4.1 can be readily exchanged with other components to better fit the statistical properties or scale of a specific dataset.

4.2 Using Graph Compression to Discover Patterns and Cluster Traces

SUBDUE discovers compressing patterns in graph data via the minimum description length (MDL) principle and a beam search over candidate subgraph patterns. This satisfies the requirement for an unsupervised method of discovering a hierarchy of meaningful components of a graphical process model, since a workflow log is also a set of subgraphs generated by a process model. The Inductive Miner complements this approach by providing the super-graph for converting a log of partially-ordered traces into subgraphs, which are then passed to SUBDUE.

Prior work showed strong anomaly detection results when running SUBDUE iteratively on a set of graphs (Noble and Cook, 2003). In these authors' work, instances of the most compressing subgraph were replaced with a single meta-vertex at each iteration, and the method

was repeated until no further compression was possible. At the end, the authors obtained a recursive and hierarchical description of a set of graphs, by which they modeled their anomaly detection scheme. Several previous works by Jonyer, Holder, and Cook (2000, 2001) also used SUBDUE for hierarchical graphical clustering, applying the same pattern of iteratively replacing compressing substructures with single-vertex prototypes to generate a hierarchical decomposition of graphical data.

A similar approach using GBAD (Eberle and Holder, 2007) was also tested, by which workflow traces were iteratively recompressed using the most-compressing subgraph found at each iteration. The three anomaly detection methods of GBAD were used to detect anomalies at each iteration. While successful at discovering patterns, this method suffered a high false positive rate for anomaly detection. Ultimately the issue was iterative recompression: on successive iterations, the most highly compressing subgraph was often only a small alteration (vertex substitution, deletion, or insertion) to a compressing subgraph found by the previous iteration. GBAD's primary deficiency in this context is that its anomaly detection methods apply to the local vicinity of the compressing pattern discovered by SUBDUE. Hence, the search space was highly redundant, repeatedly analyzing the same regions of the process model, progressing very slowly toward the outlying model regions where compressing structure decays and where anomalies lie.

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

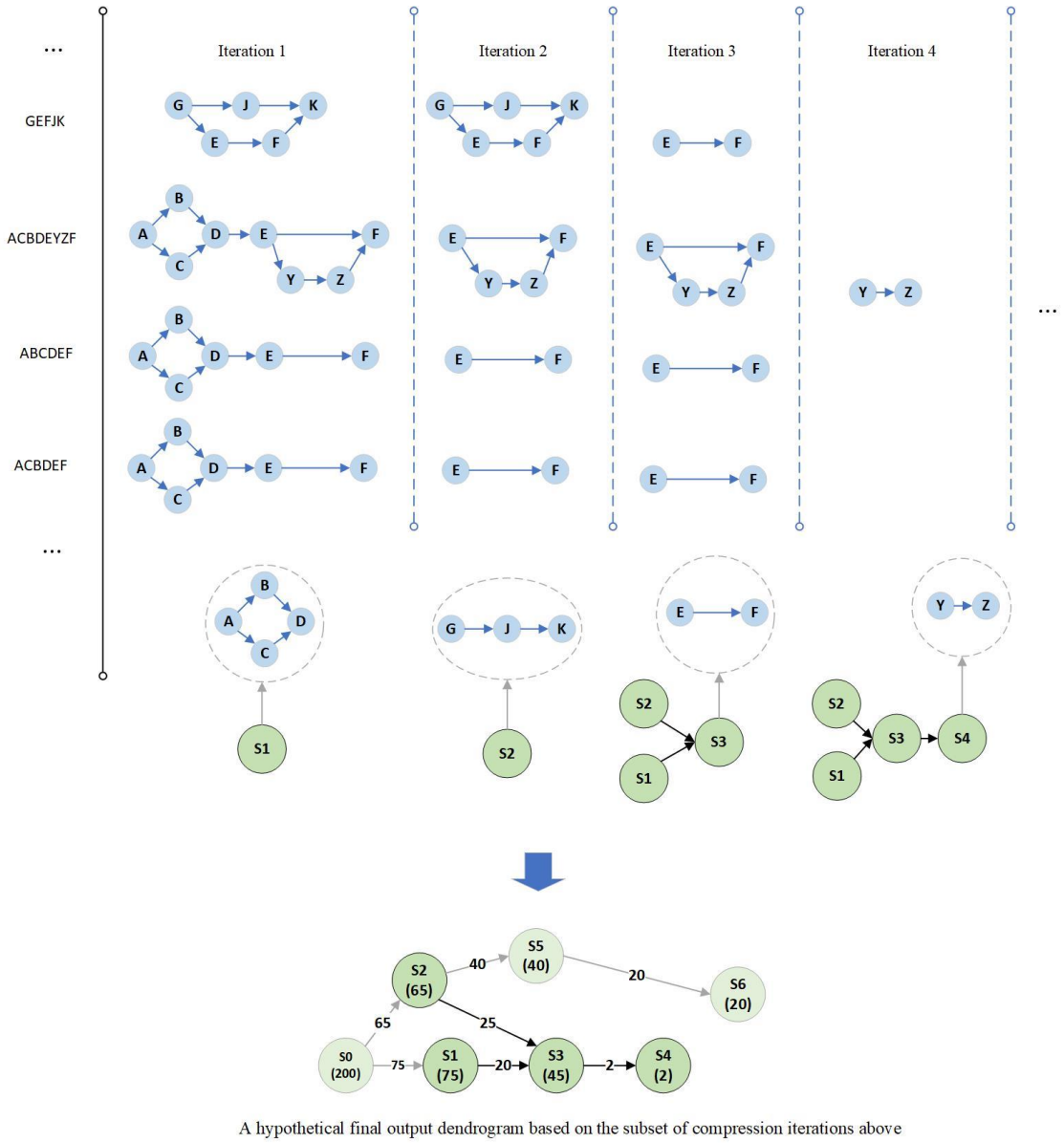


Figure 4.2: At top, constructing a dendrogram of graphical features from graphical process data. At bottom, the final dendrogram, a directed acyclic graph composed of compressed components. This includes the sub-structures found in the four iterations above, and additional hypothetical components (shaded) from later compression iterations. The numbers in parentheses represent the hypothetical frequency of each substructure.

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, S1 is discovered (in green), a

graph of four vertices. Then S1 is 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 fashion, 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 $G_{dendrogram} = (V, E, w_v, t_v, w_e)$ of weighted vertices and edges. Each vertex $v \in V$ represents a compressed graphical substructure, each with trace frequency w_v and a set of trace ids t_v (e.g., $t_v = \{2, 17, 45 \ldots\}$) it partially compressed, thus $w_v = |t_v|$. Edges represent immediate substructure ancestry via trace ids and are the most difficult to portray. Consider two substructures, s_i and s_j, where s_i was compressed prior to s_j, thus $i < j$. Substructures s_i and s_j are ancestors if $|t_i \cap t_j| > 0$, indicating that s_j compresses one or more trace ids also compressed by s_i in a previous iteration. Therefore, the Boolean valued substructure ancestry function $a(i, j)$ is defined by:

$$a(i,j) = \left|t_i \cap t_j\right| > 0 \wedge (i < j)$$

However, only immediate ancestors are linked, such that an edge only exists if s_i is the most recent ancestor satisfying $a(i, j)$:

$$e(i,j) = a(i,j) \wedge \forall k, a(k,j) \rightarrow (i \geq k)$$

The righthand clause denotes that s_i is the most immediate ancestor satisfying $a(i, j)$. In code, this amounts to a simple look up at each iteration: for each trace id $t' \in t_i$ compressed in the

current iteration, one looks up t' in the trace ids of previously compressed substructures, and returns the most recent substructure containing t', if one exists.

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

$$w_{e_{i,j}} = \left|t_i \cap t_j\right|, if\ e(i,j)$$

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 record table rows shown.

Table 4.1: Substructure relation example

Iteration	Substructure	Trace-ids
…	…	…
$i-2$	s_{i-2}	{4,5,**6**}
$i-1$	s_{i-1}	{1,2,3,4,5,**6**}
i	s_i	{**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, $a(i, i-2)$ and $a(i, i-1)$ both evaluate to true, since s_{i-1} and s_{i-2} contain trace id **6**, in bold. However, $e(i,j)$ is only true for $e(i, i-1)$, since it is the most recent.

Given this record format, the intuitive code implementation is simply to iterate over the trace-ids of a substructure, traversing up the table rows and returning the first substructure (if any) containing each id. Thus, an edge $e_{i,j} \in E$ represents a subset of a row's trace ids contained by the first previous row also containing those ids. The edge weight $w_{e_{i,j}}$ is just the size of this subset. A substructure may have multiple immediate ancestors, so substructures can have multiple parents or multiple children in the dendrogram.

Incorporating the functions $e(i,j)$ and $w_{e_{i,j}}$, an algorithm for adding a substructure to the dendrogram is therefore:

AddSubstructure Pseudocode Definition

Input *dendrogram*: The current dendrogram of connected substructures
newSubstructure: A 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 any edges as described previously
3.  for existingSubstructure in dendrogram.substructures:
4.      i = existingSubstructure.index
5.      if e(i,j):
6.          edge = Edge(i,j)
7.          edge.weight = w_{e_{i,j}}
8.          dendrogram.edges = dendrogram.edges ∪ edge
9.  return dendrogram
```

This construction is subject to criticism based on $e(i,j)$: why not account for edges to all ancestors, not just the most recently compressing one? This is a valid criticism if one assumes that the relationship between substructures and their relative iteration order is arbitrary. Under this interpretation their relations would likely also be represented as undirected. However, the significance of compression-order is one of decreasing information, where highly compressing substructures are removed from the log first, followed by substructures of decreasing compression with respect to the original log.

Hence, the relationship between substructures 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. Many alternative dendrogram semantics could be conceived for other objectives, this representation was chosen because it proved suitable for anomaly detection by exploiting these dependencies between substructures removed from traces. Alternative representations are worthy of future work, such as using graphical methods to explore the undirected model for which dendrogram edges are defined by $a(i,j)$ instead of $e(i,j)$.

Overall, this approach loosely resembles data dimensionality reduction, in which data is compressed via an ordered set of vectors of 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 be located "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, and is explicitly defined here for reproducibility:

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
2.     trace.vertices = trace.vertices \ substructure.vertices
       #delete any edges within or incident on this substructure
3.     for edge in trace.edges:
4.         if edge.destVertex in substructure.vertices:
5.             trace.edges = trace.edges \ edge
6.         if edge.sourceVertex in substructure.vertices:
7.             trace.edges = trace.edges \ edge
8. 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 each trace is converted to a graph, and hence 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 descriptive model of the input log. The dendrogram can be analyzed in post-processing for frequent process features, redundant

behavior, outliers, and anomalies. The work by (Diamantini et al., 2015) 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 (Song et al., 2009), 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. Using this information, the process model returned by the Inductive Miner could be reduced for greater specificity. When applied to a real-world setting, the recurrent behavior of an unstructured institution can be discovered, and thereby important processes can be identified, measured, and improved via business process formalisms.

In sum, coupling SUBDUE with the generality feature of the Inductive Miner provides a framework for concise modelling of unstructured "spaghetti" process environments. Similarly, an analyst may examine highly similar components of the dendrogram, likely indicating duplicate work or poor cohesion among business processes. The dendrogram offers a range of pattern mining and other enterprise uses beyond anomaly detection.

4.3 Anomaly Detection Method

Anomaly detection illustrates an important use of the dendrogram because of the structural characteristics of the dendrogram: 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 components. A second quantitative feature is that the size of the dendrogram components decreases smoothly, and then drops suddenly, such that the only

remaining traces/subgraphs represent anomalies, outliers, and noise. This effect is a property of the regularity of data as a distribution of substructures, hence the dendrogram representation benefits from data with structural regularity.

This property is useful for anomaly detection since many discriminating metrics can be devised to differentiate 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 both tend to have lower frequency. As a result, detecting anomalies reduces to finding these sharp boundaries between high frequency substructures and relatively lower frequency substructures adjacent to them.

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 was chosen because it discriminatively quantifies the relationships between parent and child substructures such that unusual child substructures have very low probability. Under this model, each substructure is assigned a Bayesian probability defined as:

$$p_{bayes}(child|parents) = \frac{p(parents|child)\, p(child)}{p(parents)}$$

Equation 3: The Bayesian child posterior probability metric for anomaly detection.

Where unconditional prior substructure probabilities like $p(child)$ are defined in terms of the global probability of a substructure in any trace, or $p(s) = \frac{\#(s)}{|traces|}$, where the '#' operator returns the frequency of substructure *s*. It is worth mentioning that this metric is subject to criticism due to its local nature: the edge-relationships in the dendrogram only loosely represent parent-child

relationships via their compression order, whereas a child's probability could be estimated from all its ancestor vertices for a global characterization. For parentless root vertices $p_{bayes}(child|parents) = p(child)$, although root vertices are nearly always highly compressing and frequent enough to exceed the anomaly threshold.

Characterizing $p(parents|child)$ 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. Hence, one must sum over all parents of a given child, a set of independent events (since substructures are treated as independent), and weight each event by its likelihood $p(parent_i|child)$:

$$\frac{p(parents|child_i)}{p(parents)} = \sum_{j}^{|parents|} \frac{p(parent_j|child_i)}{p(parent_j)} * \lambda_j$$

Equation 4: The likelihood and evidence components of equation 3.

Each summation component must be weighted by λ_j to obtain a proper probability distribution, such that $\sum_j \lambda_j = 1.0$. Each λ_j is also equal to $p(parent_j|child_i)$ since that is the proportion of time that $parent_j$ is a parent of $child_i$. Substituting 'c' for '$child$', 'p' for '$parent$', and 'P' for '$parents$', the fully-defined metric becomes:

$$p_{bayes}(c_i|P) = p(c_i) * \sum_{j}^{|P|} \frac{p(p_j|c_i)}{p(p_j)} \lambda_j$$

Equation 5: The explicit expression for equation 3, by incorporating equation 4.

Anomalous substructures are expected to have a low value for p_{bayes}, and a substructure is flagged as anomalous using the anomaly threshold when $p_{bayes} < \alpha_{bayes}$, where α_{bayes} is the

anomaly threshold in (0,1.0]. Traces containing the anomalous substructure are then 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 = p_{bayes}(substructure | parents)
5. if bayesProbability < *bayesThreshold*:
6. *anomalyIds* = *anomalyIds* ∪ substructure.traceIds
7. *anomalousStructures* = *anomalousStructures* ∪ substructure
8. return *anomalyIds, anomalousStructures*

The method returns both the anomalous structures and their corresponding trace ids. The anomaly detection method is motivated by the properties of the dendrogram, whereby substructures that are highly-compressing and frequent tend to cluster near around the ancestor vertices, while anomalies are attached to these vertices with far lower frequency. This is the core property exploited by the Bayesian metric in Algorithm 2 to perform anomaly detection.

Summarizing, this chapter introduced the use of graphical compression on dynamic process data, and specifically the application of SUBDUE. SUBDUE was deemed appropriate given its ability to estimate frequent substructures, by which a dendrogram of substructures can be built from process data, as detailed in Algorithm 1. Using this dendrogram, further analyses can be performed, and Algorithm 2 lends one such application for anomaly detection. The next chapter evaluates these algorithms under a variety of conditions, including a comparison with an existing method, and a real-world data evaluation.

CHAPTER FIVE: ANOMALY DETECTION EVALUATION

This chapter evaluates the anomaly detection method described in the previous chapter on several synthetic and real-world datasets. The synthetic data provides a supervised evaluation setting for which data contains anomaly class labels, unlike a real-world setting. The algorithm is first tested on several synthetic datasets under controlled conditions, and then compared with an implementation of an existing method. Finally, the method is tested on real-world data without anomaly class labels.

5.1 Algorithm Evaluation

Although real process-oriented datasets are available, they do not offer controlled conditions sufficient to compare the characteristics of algorithms over a range of data parameters. Instead a synthetic data generation algorithm was used, as found in appendix A of (Bezerra and Wainer, 2013), 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. To cohere to a stable performance baseline, the same experimental set up was used as described in (Bezerra and Wainer, 2013): 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 is performed on real-world data derived from software execution logs.

5.2 Data Generation Algorithm

Data generation consisted of two steps: generating process models and then generating traces from them. The set of parameters $\boldsymbol{\theta}_{\boldsymbol{model}}$ 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 the 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 $\boldsymbol{\theta_{model}}$ constrain model complexity to a probability distribution over these operators, replicated from (Bezerra and Wainer, 2013) and fixed throughout this work.

Since probabilistic model-generation allows for the possibility of unusual or task-trivializing models, additional basic tests were applied to ensure sufficient complexity. These 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 containing a single anomalous structure is shown in figure 5.1.

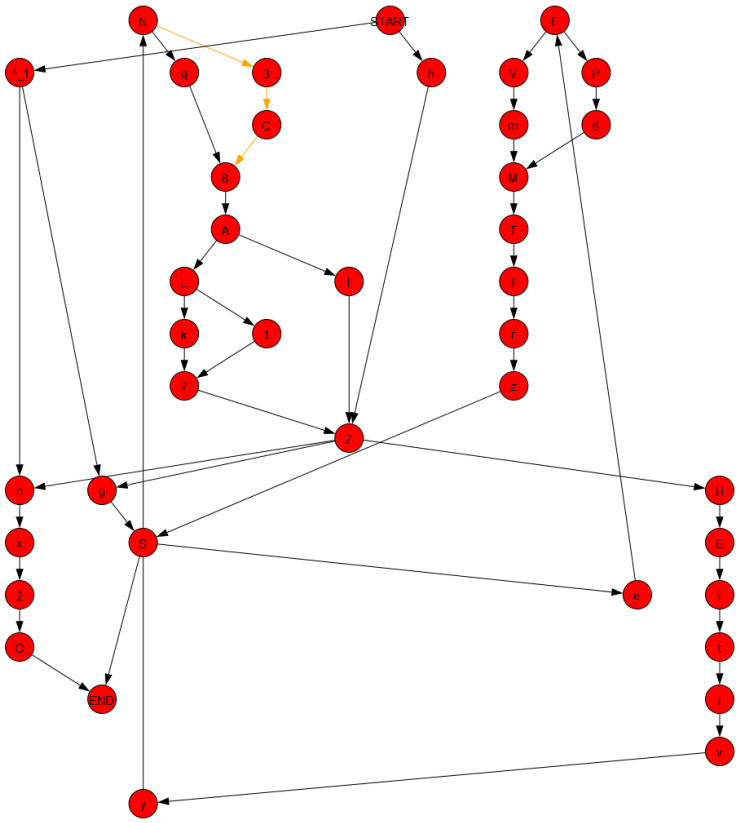


Figure 5.1: A stochastically generated model with a single anomaly, given by the yellow edges at top center-left, starting from vertex 'N'. Black edges represent normal behavior. Except for AND constructs, vertices with more than one outgoing edge represent choice vertices, with embedded stochastic parameters determining the choice of edge.

A second parameter set $\boldsymbol{\theta_{trace}}$ defined the trace-generation distribution constraining the graphical walks of traces, which is only defined for the choice operators OR-SPLIT and LOOP. These parameters determined trace diversity, from uniform to very non-uniform. A value of $\boldsymbol{\theta_{trace}} = 0.9$ implies the trace-generation scheme assigned a 0.9 probability of traversing one branch and a $1.0 - \boldsymbol{\theta_{trace}} = 0.1$ of its alternative. By varying $\boldsymbol{\theta_{trace}}$ 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 for uniform random partial-order.

Lastly, the parameters $\boldsymbol{\theta_{anomalies}}$ defined the probability of generating anomalies, which encompassed both the generation of anomalous structures and the embedded traversal probability of those structures 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). Notably, embedding anomalies probabilistically allows for generated logs to contain zero anomalies. This was important to include in synthetic data, 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 were guaranteed to achieve sufficient complexity, and to generate an exponential distribution of unique traces.

5.3 Experiment 1: $\boldsymbol{\theta_{trace}}$ Sensitivity

In the first experiment, 60 models were generated under fixed $\boldsymbol{\theta_{model}}$, from which 1000 traces were generated separately for $\boldsymbol{\theta_{trace}}$ values in $\{0.5, 0.6, 0.7, 0.8, 0.9\}$, and $\boldsymbol{\theta_{anomaly}}$ fixed to 0.05. This complete dataset is referred to as D1. For each $\boldsymbol{\theta_{trace}}$ value, the method was evaluated for values of $\boldsymbol{\alpha_{bayes}} \in [0,1.0]$ in increments of 0.02: $\{0.0, 0.02, \ldots 1.0\}$. A large runtime bottleneck is that the method is currently implemented via script components that construct complete application processes (such as ProM [2]) 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, some of which was merely result compilation.

Accuracy, precision, recall, and f1-measure were averaged over the 60 test models for the cross product of $\boldsymbol{\theta_{trace}}$ and $\boldsymbol{\alpha_{bayes}}$ values, giving the plots and ROC curve shown in figure 5.2.

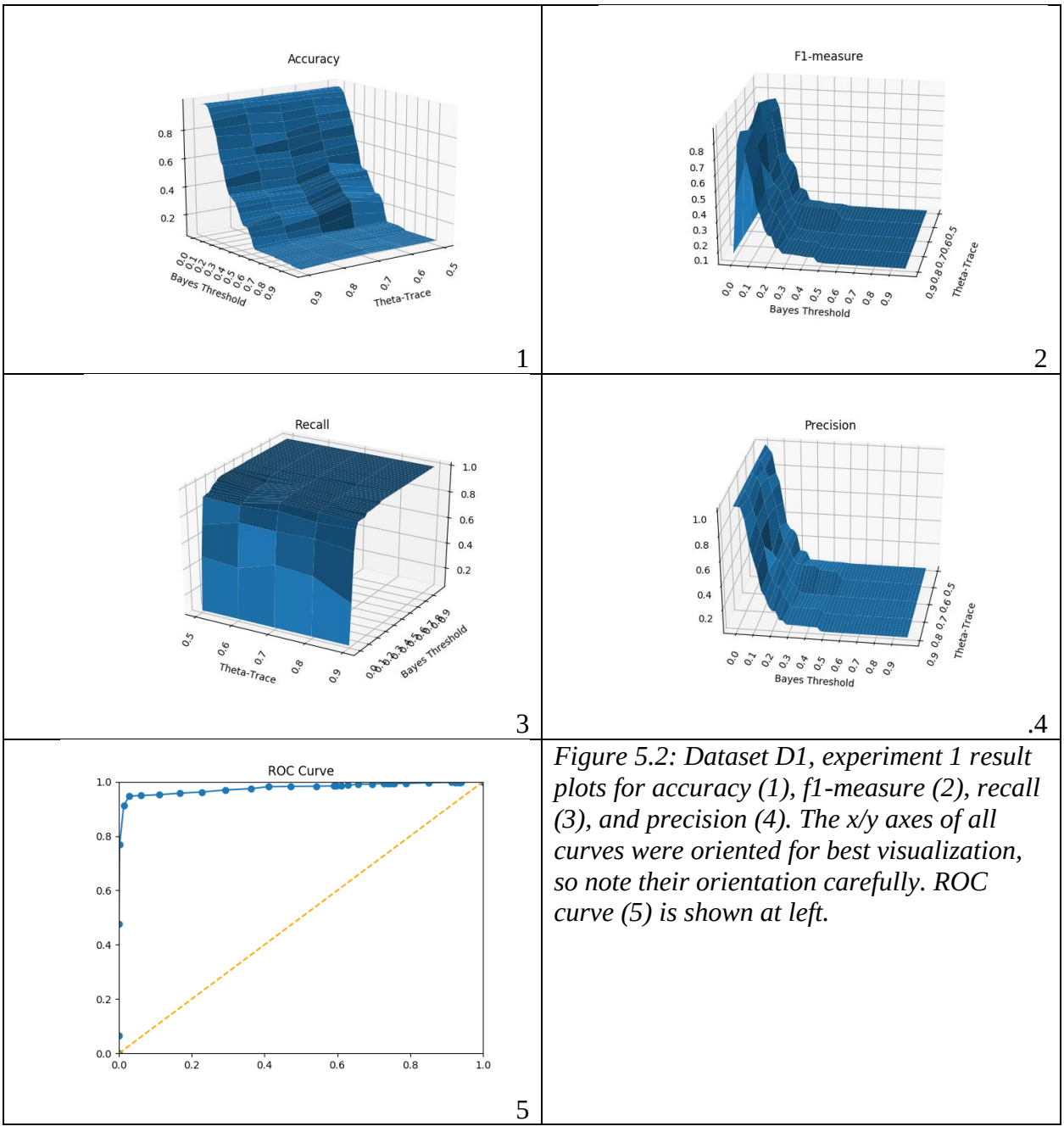


Figure 5.2: Dataset D1, experiment 1 result plots for accuracy (1), f1-measure (2), recall (3), and precision (4). The x/y axes of all curves were oriented for best visualization, so note their orientation carefully. ROC curve (5) is shown at left.

Notably, for all four performance curves, performance degraded only slightly along the $\boldsymbol{\theta_{trace}}$ axis for larger values, indicating the method worked well for very skewed trace

distributions. Clearly, $\boldsymbol{\alpha_{bayes}}$ impacted performance most strongly. From the top-left, accuracy maximized around $\boldsymbol{\alpha_{bayes}} = 0.04$, 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 $\boldsymbol{\alpha_{bayes}}$, as expected for a probabilistic anomaly detection threshold. 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 $\boldsymbol{\alpha_{bayes}}$ values and informs us where the greatest saturation of anomalies occurred. The f1-measure results are most informative in terms of $\boldsymbol{\alpha_{bayes}}$ selection, suggesting one choose an $\boldsymbol{\alpha_{bayes}}$ 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 $\boldsymbol{\alpha_{bayes}} \in$ [0.0,1.0] in 0.02 increments, averaged over all 60 models and all values of $\boldsymbol{\theta_{trace}}$. The area under the ROC curve is very near 1.0, indicating a high true-positive rate.

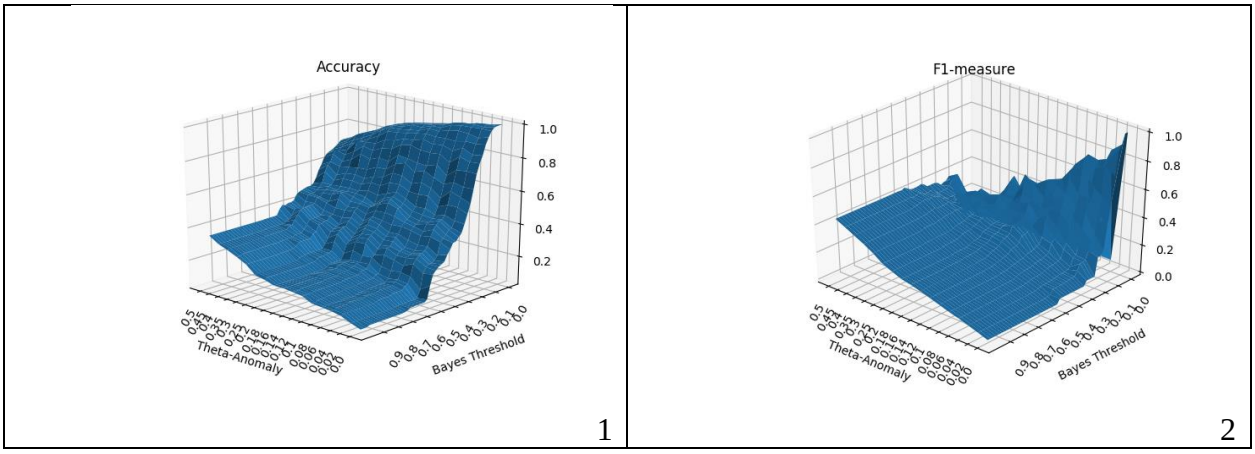
The results show lower values (0.04-0.10) of $\boldsymbol{\alpha_{bayes}}$ were preferable and can be tuned to suit recall vs. precision objectives. From a risk perspective, recall is the most important in terms of capturing all anomalies, at the expense of decreasing accuracy and precision. On the other 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 $\boldsymbol{\alpha_{bayes}}$ 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.

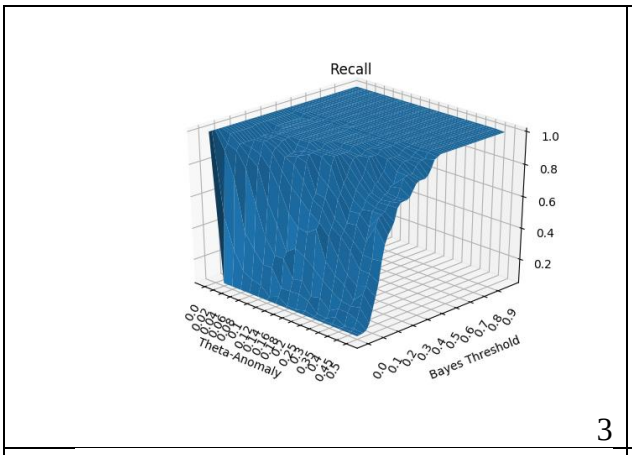
Finally, this experiment analyzed performance over a range of $\boldsymbol{\theta_{trace}}$ and $\boldsymbol{\alpha_{bayes}}$, but with $\boldsymbol{\theta_{anomaly}}$ fixed to 0.05. The results demonstrated that an algorithmic parameter, $\boldsymbol{\alpha_{bayes}}$,

determined the results far more than the data parameter, $\boldsymbol{\theta_{trace}}$. Thus, more experiments were required to stress properties of the data.

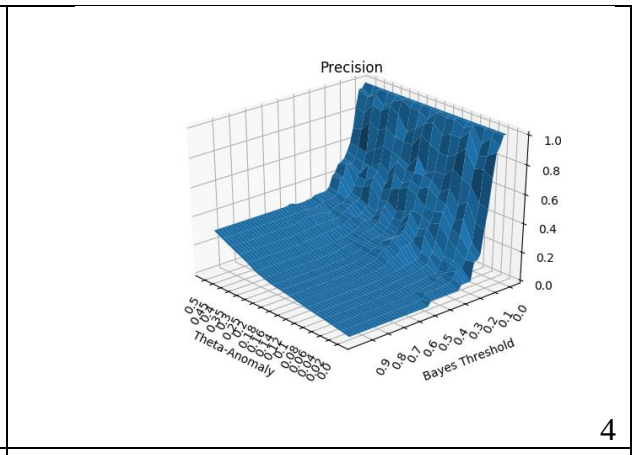
5.4 Experiment 2: $\boldsymbol{\theta_{anomaly}}$ Sensitivity

A second experiment was designed to analyze the sensitivity of the method with respect to $\boldsymbol{\theta_{anomaly}}$, and with $\boldsymbol{\theta_{trace}}$ fixed. This was to verify that the previously selected $\boldsymbol{\theta_{anomaly}}$ did not trivialize the task of anomaly detection. In this case, $\boldsymbol{\theta_{trace}} = 0.5$ and $\boldsymbol{\theta_{anomaly}}$ 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 $\boldsymbol{\theta_{trace}}$ was to yield a more uniform distribution of traces, and likewise recall from experiment 1 that results showed little impact across the parameter's range. The low range of $\boldsymbol{\theta_{anomaly}}$ was expected to trivialize the discovery of anomalies due to their low expected frequency, whereas the high range approached $\boldsymbol{\theta_{trace}}$, making anomalous substructures and normal substructures ambiguous. The same models as dataset D1 were used, but new traces were generated for the new parameter values, generating dataset D2.

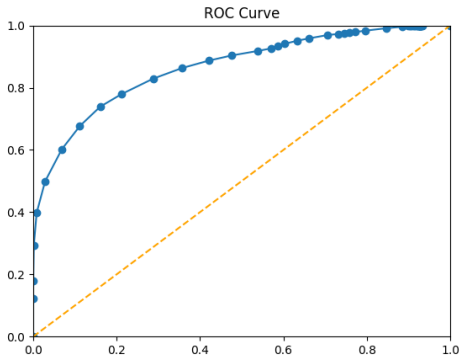




3



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 $\boldsymbol{\theta_{anomaly}}$ 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 $\boldsymbol{\theta_{anomaly}}$ and $\boldsymbol{\theta_{trace}}$ are not known in advance.

Experiments 1 and 2 were performed with a different anomaly characteristic. 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 $\boldsymbol{\theta_{trace}} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$, giving dataset D3. Dataset D4 was generated likewise but for various values of $\boldsymbol{\theta_{anomaly}}$, as per experiment 2.

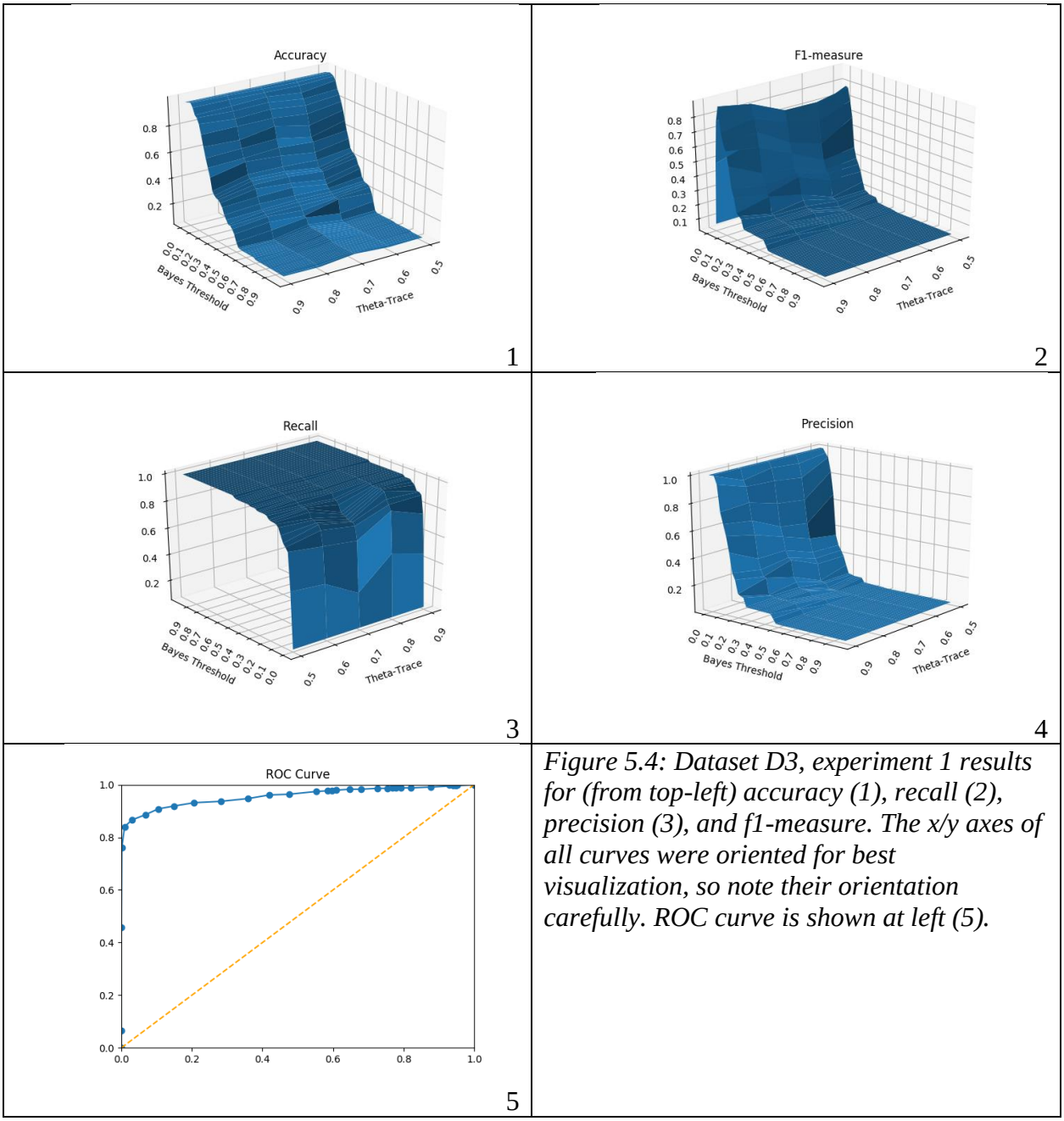


Figure 5.4: Dataset D3, experiment 1 results for (from top-left) accuracy (1), recall (2), precision (3), and f1-measure. The x/y axes of all curves were oriented for best visualization, so note their orientation carefully. ROC curve is shown at left (5).

Experiment 1 results for dataset D3 are shown in figure 5.4. Experiment 2 results for dataset D4 are shown in figure 5.5.

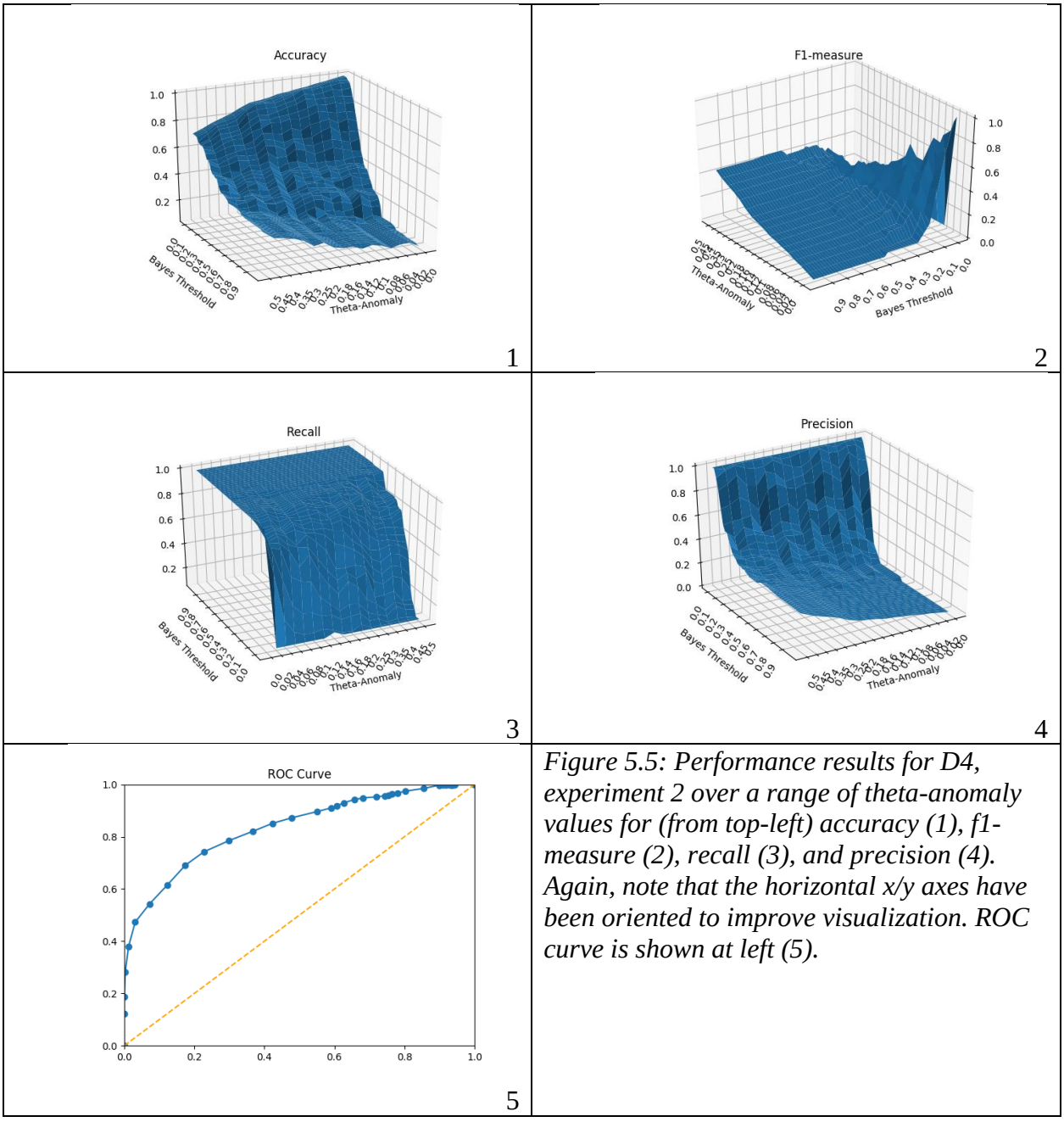


Figure 5.5: Performance results for D4, experiment 2 over a range of theta-anomaly values for (from top-left) accuracy (1), f1-measure (2), recall (3), and precision (4). Again, note that the horizontal x/y axes have been oriented to improve visualization. ROC curve is shown at left (5).

As the figures show, the curvature of the four performance metrics showed no significant changes, but overall performance decreased measurably, most clearly when comparing their

ROC curves. The decrease in performance was due to the less obvious nature of anomalies when replaced by existing activities.

5.5 Experiment 3: Multiple Anomalous Structures

The previous experiments analyzed performance sensitivity with respect to $\boldsymbol{\theta_{trace}}$ and $\boldsymbol{\theta_{anomaly}}$, but dealt with models for which the number of anomalous structures was distributed according to the model generation algorithm. As a result, most models contained between 0 and 2 anomalous structures by which to generate anomalous traces. These model classes are denoted as M_k, where *k* defines the number of anomalous structures in model *M*. Despite that larger values of $\boldsymbol{\theta_{anomaly}}$ in experiment 2 generated more anomalous traces, they were generated from models containing only a low number of anomalous structures. Therefore, a remaining task was to analyze performance sensitivity to a controlled range of M_k.

This required generating D5, a final model-oriented dataset consisting of models with *k* anomalous structures for $k \in \{0, 1, 2, 4, 8, 16, 32\}$. Thirty M_k models were generated for each *k*, each with a single log generated from fixed trace parameters: $\boldsymbol{\theta_{trace}} = 0.5$ and $\boldsymbol{\theta_{anomaly}} = 0.05$. 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 $\{\boldsymbol{\theta_{trace}}, \boldsymbol{\theta_{trace}}, \boldsymbol{\theta_{anomaly}}\} = \{0.5, 0.5, 0.05\}$, which do not sum to 1.0, so the edge probabilities at each vertex were simply normalized after anomaly generation. An example M_{16} model with 16 anomalies is shown in figure 5.6.

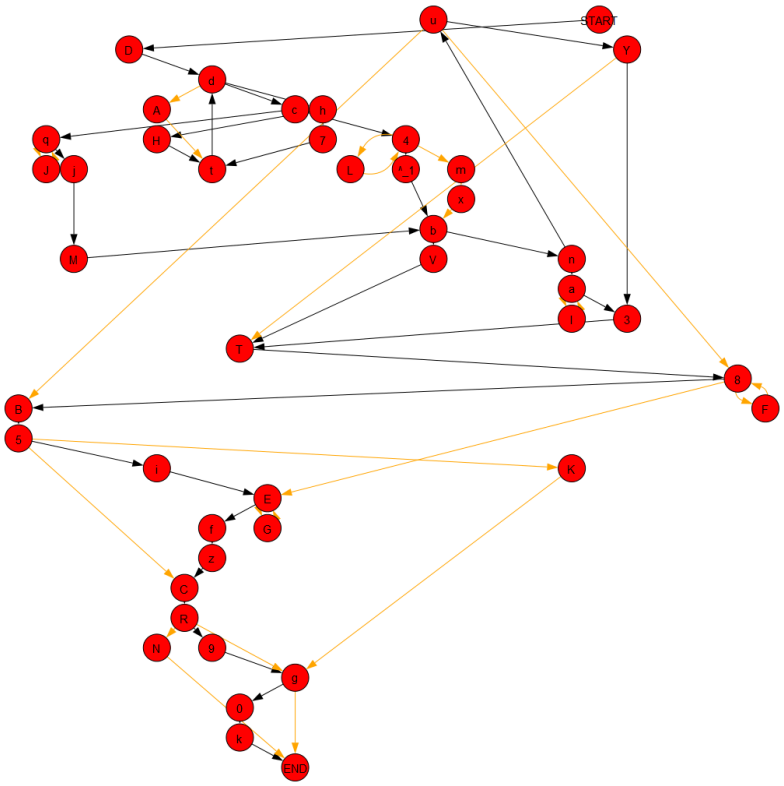


Figure 5.6: A synthetic model with 16 anomalies (yellow edges) and approximately 40 activities (vertices). Black edges represent paths for normal behavior. Vertices prefixed "^_" represent null activity transitions, where such a path immediately resolves to its successor. These only provide an implementation representation, and do not appear in the final trace data.

Models were generated using the same procedure as for previous experiment until one satisfied the target number k of anomalous structures. Models with $k > 2$ were very unlikely under the previous model generation parameters, so in this case anomalous structures were added until the target k was reached. To achieve a nearly uniform distribution of insertion, deletion, and substitution anomaly structures, this process used a stochastic method of its own, using the decision tree shown in figure 5.7.

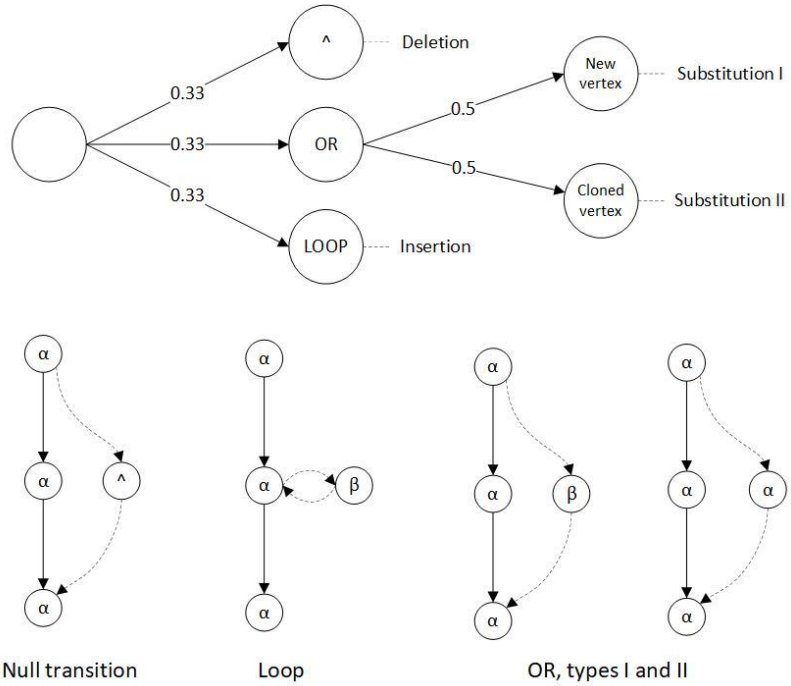
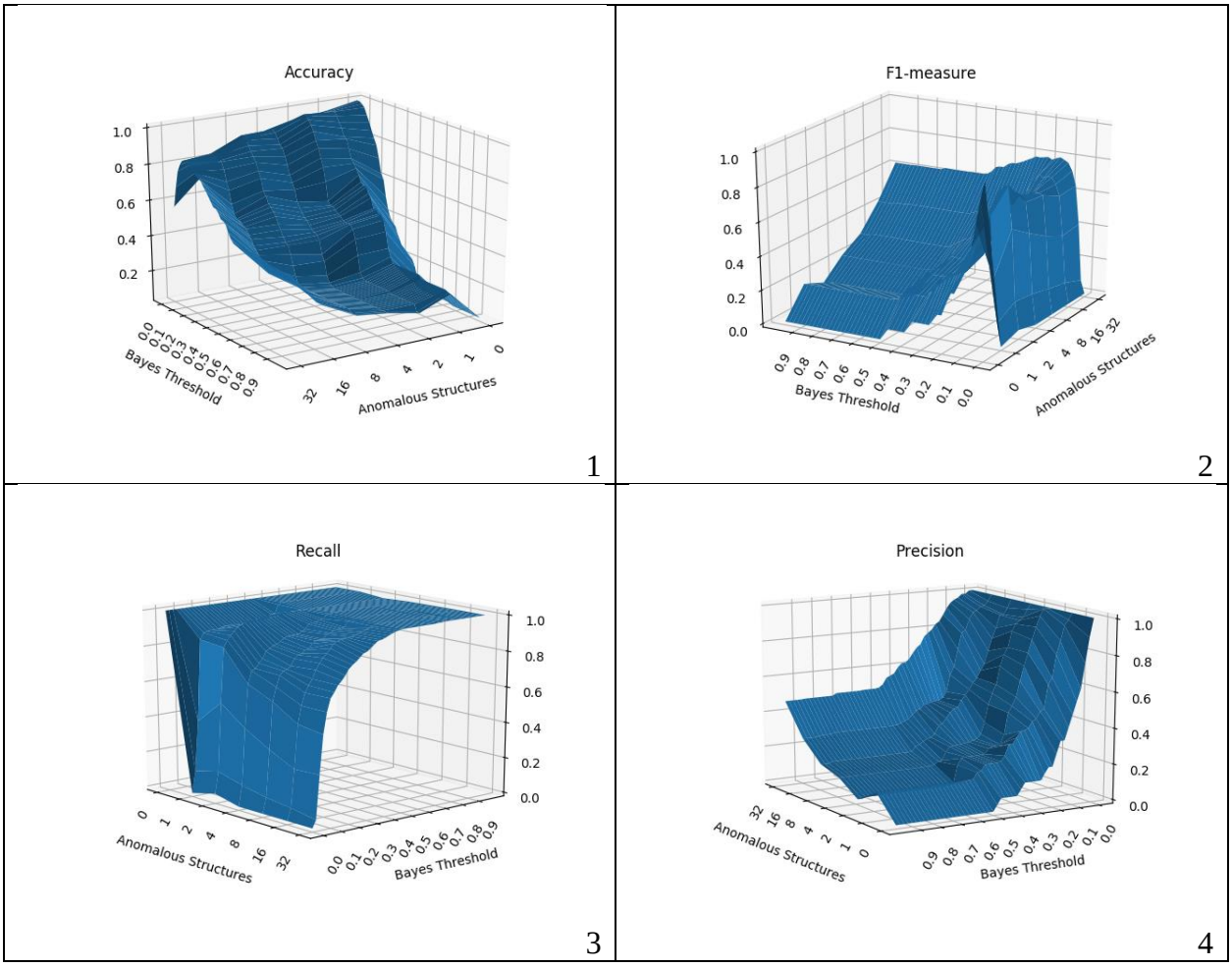


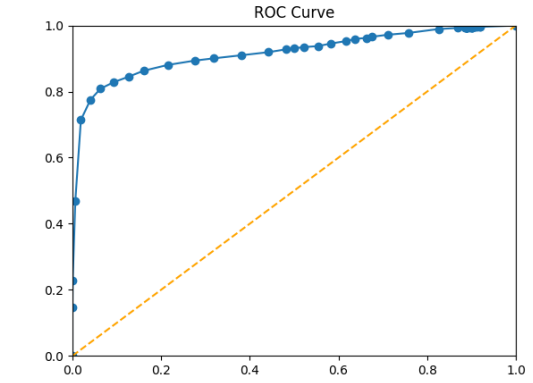
Figure 5.7: The anomalous structure generation decision tree and three anomaly types. Null transitions '^' represent execution paths that bypass (delete) normal behavior. LOOP structures transition and return from a newly inserted activity, creating an insertion. Lastly, OR branches decompose to two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

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 fixed

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 $k\ x\ \alpha_{bayes}$ values in terms of accuracy, recall, precision, and f1-measure, averaged over thirty models/logs. Here $k \in \{0, 1, 2, 4, 8, 16, 32\}$ and $\alpha_{bayes} \in [0, 1.0]$ in increments of 0.02. The results are shown in figure 5.8.





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. The k parameter is given by the axes labeled "Anomalous Structures." ROC curve is shown at left (5).

Overall, these results are most like experiment 1, since $\boldsymbol{\theta_{trace}}$ and $\boldsymbol{\theta_{anomaly}}$ were fixed, and experiment 1 showed little variance over the range of $\boldsymbol{\theta_{trace}}$. The overall curvature shows accuracy decaying in linear fashion for larger *k*, due to the larger numbers of anomalous traces in the logs. Again, recall quickly maximized since small $\boldsymbol{\alpha_{bayes}}$ 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, experiment 3 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 those anomalies arose from separate-but-rare events. In some ways this 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 $\boldsymbol{\theta_{anomaly}}$ one can expect worsening performance for larger k, as experiment 2 demonstrated.

This shows that $\boldsymbol{\theta_{anomaly}}$ 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.

5.6 Comparison with Existing Methods

To provide context for these results, the Sampling Algorithm from (Bezerra et al, 2013) was also evaluated, upon which the data generation method was based. 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

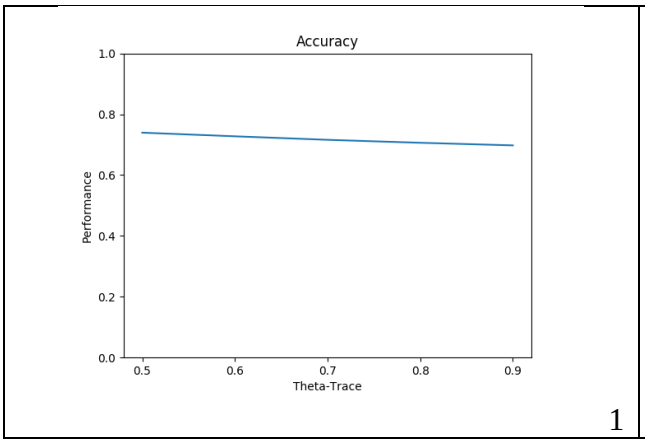
Input *L:* A process log
s: Sampling proportion, $s \in (0,1.0)$
mine: A process mining algorithm
Output T^A: The set of traces flagged as anomalous

```
T = set of all unique traces from the log L
T^C = { }     #used to contain anomalous candidate traces
T^A = { }     #used to contain traces flagged as anomalous
for t in T do:
    if freq_L(t) ≤ 0.02 then:
        T^C = T^C ∪ {t}
for t in T^C do:
    S = sample of s% of traces of L
    M = mine(S)
    if t is not replayable on M:
        T^A = T^A ∪ t
return T^A
```

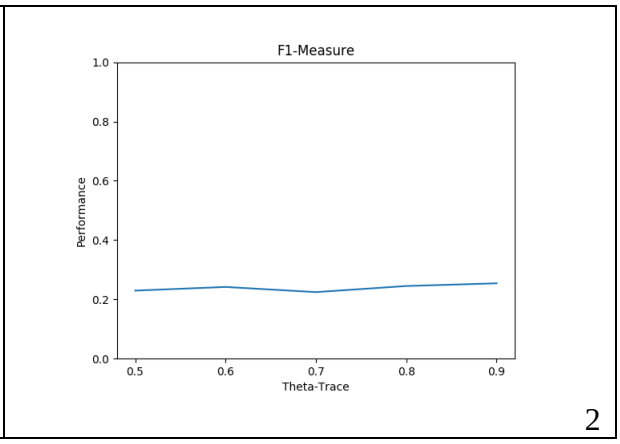
To detect anomalies, the Sampling Algorithm expects that an anomalous trace will deviate significantly from the expected process model derived from a randomly selected subset of the traces. Anomalous traces are defined as outliers that are also inconsistent with the expected behavior of a log. To exploit this property, the Sampling Algorithm 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 all the traces, and the trace is added to the anomalous trace set if it

is not replayable on the mined model. In contrast with the method presented in this work, the Sampling Algorithm flags anomalies without providing causal, structural context for the flag, although additional processing could provide it, but also does not provide the complete feature model which the dendrogram provides.

To test the Sampling Algorithm, it was implemented and evaluated on datasets D1 and D2, for $\boldsymbol{\theta_{trace}} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$ 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 $\boldsymbol{\theta_{anomaly}}$ values. Exhaustive testing was difficult because of implementation dependencies on ProM [2], 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

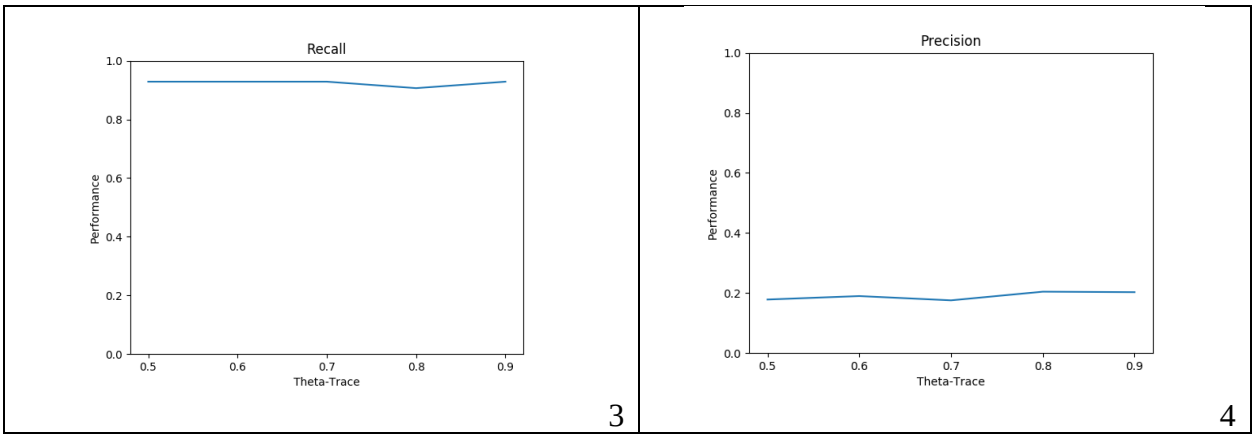


Figure 5.9: From top-left, dataset D1 Sample Algorithm results, (1) accuracy, (2) f1-measure, (3) recall, and (4) precision for $\theta_{trace} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$.

Table 5.1 compares performance of the method (Algorithm 2) with a generic value of $\boldsymbol{\theta_{bayes}} = 0.08$, beside the Sampling Algorithm (Bezerra et al., 2013), for dataset D1. The Δ column shows the difference between the two left-hand columns. Results were simply averaged over all $\boldsymbol{\theta_{trace}}$ values under test.

Table 5.1: Dataset D1 Performance Comparison

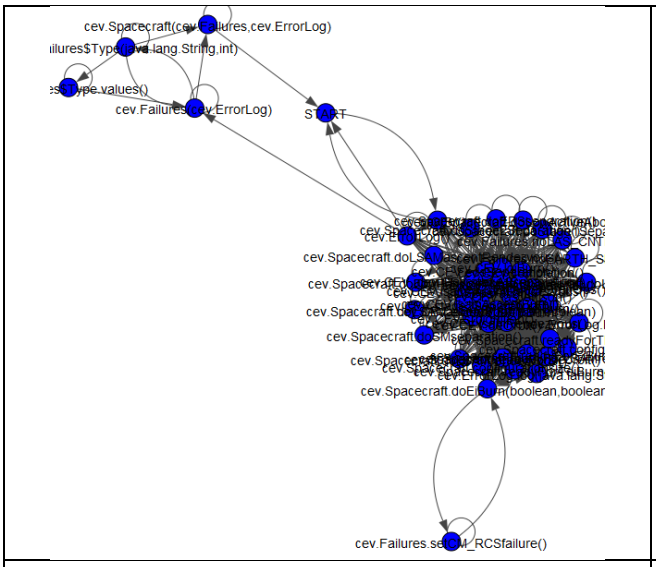
Algorithm	Algorithm 2	Sample	Δ
Accuracy	0.972	0.717	+0.255
Recall	0.892	0.924	-0.032
Precision	0.700	0.186	+0.514
F1-Measure	0.721	0.233	+0.488

As shown in table 5.1, Bezerra's Sampling Algorithm performed well, but below Algorithm 2 for all metrics except recall. The performance discrepancies are attributable to differences in their objectives: 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. The Sampling Algorithm's frequency threshold and sampling rate were the best values reported by the authors but could be optimized to improve performance on this data.

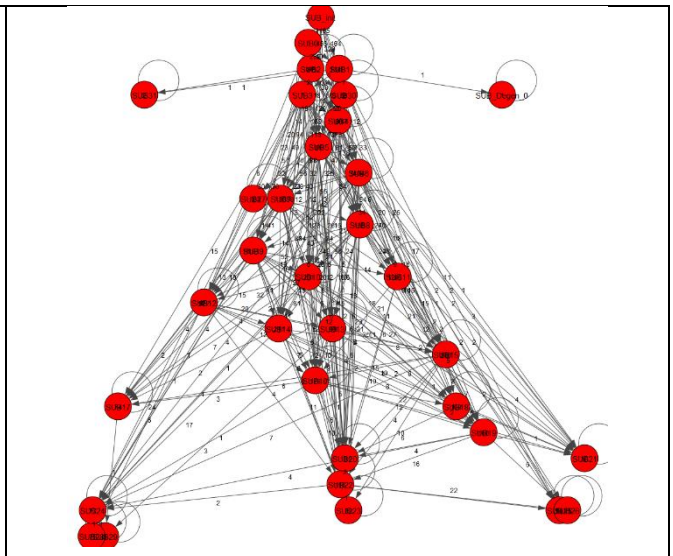
5.7 Real Data Evaluation

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 by software unit-tests of the NASA Crew Exploration Vehicle (CEV) (Leemans, 2017). The CEV system implements the complete UML-design of 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. The work, "Mining Specifications of Malicious Behavior" by (Christodorescu et al., 2007), used a similar graphical code execution formalism, "malspec", to detect and evaluate malware using a supervised classification approach, demonstrating the capacity of process-oriented, graphical representations of code to carry out security and other software evaluation tasks.

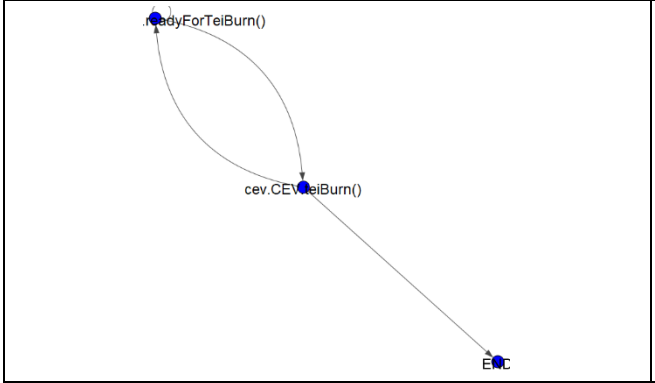
Unit-testing emphasizes code coverage, which entails repetitive software component calls and extreme value testing. Hence the distribution of these traces models the test design, whereas traces representing normal system operation are more desirable. Nonetheless, the dataset comprised a normative view 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. Instead 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.



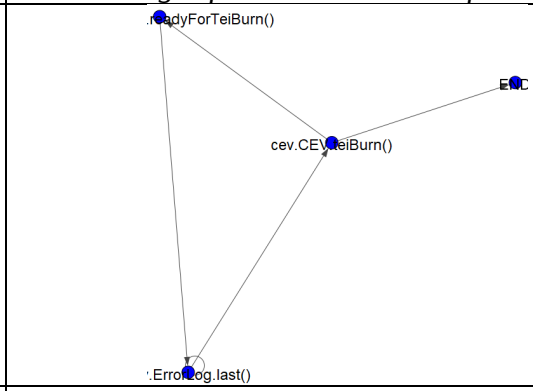
(1) *The mined process model of function calls, clearly very connected in terms of core functionality.*



(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 (Leemans, 2017).

Table 5.2: NASA CEV dataset test results		
Number of traces: 2,566		
Number of activities: 34		
α_{bayes}	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 $\boldsymbol{\alpha_{bayes}} = 0.07$ 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 $\boldsymbol{\alpha_{bayes}}$ 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). In summary, the results demonstrated how the method can be used as a grey-box verification aid, providing insight into system behavior that is overlooked by granular white- and black-box test perspectives.

5.8 Summary

Summarizing the results of this chapter, 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 $\boldsymbol{\theta_{trace}}$ parameter was shown impact performance far less than $\boldsymbol{\theta_{anomaly}}$, for which performance decreased as the value approached $\boldsymbol{\theta_{trace}}$. Additionally, the method showed comparable performance to a known anomaly detection algorithm, the Sampling Algorithm from (Bezerra et al., 2013). Finally, Algorithm 2 was tested on a real dataset which contained 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 of this chapter show strong performance on the task of anomaly detection, but also represent an algorithmic approach engineered for the specific objective of anomaly detection. Future approaches would surpass this property by incorporating less supervised engineering in their composition, or less foreknowledge of their intended internal representation and data sources, akin to how deep neural networks are capable of learning properties of data with less forward engineering.

CHAPTER SIX: CONCLUSIONS AND FUTURE WORK

Overall, the results indicate the process mining and anomaly detection method presented in this work succeeds for a range of model complexity, trace complexity, and anomaly characteristics, and demonstrates the value of graphical compression methods for process mining applications. The approach assumes that real-world data contains enough traces and sufficient regularity to discover regular graphical patterns. Additionally, the method is tunable via the $\boldsymbol{\alpha_{bayes}}$ 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 takes a non-trivial amount of time in a stream-oriented anomaly detection context. 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, a frequent goal of process mining is to create a more noise-tolerant balance between specificity and generality. Essentially, 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. Future work lies in making the approach more noise tolerant, much like the GBAD system determines acceptable deviations in the local context of a normative pattern using graph distance metrics.

A final detraction is that the method exemplifies a 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. In short, although the method is unsupervised, its design was heavily tailored to its intended purposes. However, emerging graphical deep learning models, such as auto-encoders, can encode normative patterns in real-valued parameters, softening the problems of discrete-search approaches (Goodfellow et al., 2016). 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, since they can potentially perform end-to-end normative pattern mining and anomaly detection with less manual engineering.

REFERENCES

1. http://www.xes-standard.org/
2. ProM: The Process Mining Toolkit. Version 6.6. Retrieved from http://www.promtools.org/doku.php.
3. Akoglu, L., Tong, H., & Koutra, D. (2015). Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, *29*(3), 626-688.
4. Bezerra, F., & Wainer, J. (2013). Algorithms for anomaly detection of traces in logs of process aware information systems. Information Systems, 38(1), 33-44.
5. 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.
6. Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, *41*(3), 15.
7. Christodorescu, M., Jha, S., & Kruegel, C. (2007). Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 5-14). ACM.
8. Control-Flow Patterns. Retrieved on September 13, 2017, from http://www.workflowpatterns.com/patterns/control/.
9. Cook, D. J., & Holder, L. B. (1994). Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, *1*, 231-255.

10. Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems, 1695*(5), 1-9.
11. Diamantini, C., Genga, L., & Potena, D. (2015). Esub: Exploration of subgraphs. Proceedings of the BPM Demo Session, 70-74.
12. 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.
13. Dietterich, T., Fern, A., Wong, W., Emmott, A., Das, S., Amran, M.A., & Zemicheal, T. Advances in Anomaly Detection. *Stanford Data Science Infoseminar*, February 27, 2015. Retrieved from http://web.engr.oregonstate.edu/~tgd/talks/stanford-feb-2015-v1.pdf.
14. 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.
15. 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.
16. Eberle, W., & Holder, L. (2007, October). Discovering structural anomalies in graph-based data. In Data Mining Workshops, 2007. ICDM Workshops 2007. Seventh IEEE International Conference on (pp. 393-398). IEEE.
17. Eberle, W. and Holder, L., Anomaly Detection in Data Represented as Graphs, *Intelligent Data Analysis, An International Journal,* Volume 11(6), 2007.
18. Eberle, W., & Holder, L. (2009, April). Graph-Based Approaches to Insider Threat Detection. In *Proceedings of the 5th annual workshop on cyber security and information*

intelligence research: cyber security and information intelligence challenges and strategies (p. 44). ACM.

19. Easley, D., & Kleinberg, J. (2010). *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press.
20. Esparza, J., & Nielsen, M. (1994). Decidability issues for Petri nets. *Petri nets newsletter*, *94*, 5-23.
21. Evermann, J., Rehse, J. R., & Fettke, P. (2017). Predicting process behaviour using deep learning. *Decision Support Systems*, *100*, 129-140.
22. Fawcett, T., & Provost, F. (1999, August). Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 53-62). ACM.
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. Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge: MIT press.
25. Grünwald, P. (2005). A tutorial introduction to the minimum description length principle. Accessed from https://arxiv.org/pdf/math/0406077.pdf, December 12, 2017.
26. Herbst, J. (2000, May). A machine learning approach to workflow management. In *european conference on machine learning* (pp. 183-194). Springer, Berlin, Heidelberg.
27. Hodge, V., & Austin, J. (2004). A survey of outlier detection methodologies. *Artificial intelligence review*, *22*(2), 85-126.

28. Holder, L. B. (1989). Empirical substructure discovery. In *Proceedings of the sixth international workshop on Machine learning* (pp. 133-136).

29. Jensen, K. (1987). Coloured Petri nets. In *Petri nets: central models and their properties* (pp. 248-299). Springer, Berlin, Heidelberg.

30. Joyner, I., Cook, D. J., & Holder, L. B. (2001). Discovery and Evaluation of Graph-Based Hierarchical Conceptual Clusters. J. Machine Learning Research, 19-43.

31. Jonyer, I., Holder, L. B., & Cook, D. J. (2001). Hierarchical conceptual structural clustering. International Journal on Artificial Intelligence Tools, 10(1-2), 107-136.

32. Jonyer, I., Cook, D. J., & Holder, L. B. (2001). Graph-based hierarchical conceptual clustering. Journal of Machine Learning Research, 2(Oct), 19-43.

33. Jonyer, I., Holder, L. B., & Cook, D. J. (2000, May). Graph-Based Hierarchical Conceptual Clustering. In FLAIRS Conference (pp. 91-95).

34. Karp, R. M., & Miller, R. E. (1969). Parallel program schemata. *Journal of Computer and system Sciences*, *3*(2), 147-195.

35. Kiepuszewski, B., ter Hofstede, A. H., & van der Aalst, W. M. (2003). Fundamentals of control flow in workflows. *Acta Informatica*, *39*(3), 143-209.

36. Korte B., Vygen J. (2000) Bin-Packing. In: Combinatorial Optimization. Algorithms and Combinatorics, vol 21. Springer, Berlin, Heidelberg.

37. Leemans, Maikal. (2017). NASA Crew Exploration Vehicle (CEV) software event log. http://doi.org/10.4121/uuid: 60383406-ffcd-441f-aa5e-4ec763426b76, 2017.

38. 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.

39. Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541-580.

40. Omar, S., Ngadi, A., & Jebur, H. H. (2013). Machine learning techniques for anomaly detection: an overview. *International Journal of Computer Applications*, *79*(2).

41. Peterson, J. L. (1981). Petri net theory and the modeling of systems.

42. Phadke, A. G. (1993). Synchronized phasor measurements in power systems. *IEEE Computer Applications in power*, *6*(2), 10-15.

43. 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.

44. Petri, C. A., & Reisig, W. (2008). Petri net. *Scholarpedia*, *3*(4), 6477.

45. 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.

46. Senator, T. E., Goldberg, H. G., Memory, A., Young, W. T., Rees, B., Pierce, R., et al. (2013, August). Detecting insider threats in a real corporate database of computer usage activity. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 1393-1401). ACM.

47. Shetty, J., & Adibi, J. (2004). Enron email dataset. USC Information Sciences Institute, Tech. Rep.

48. 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.

49. Tax, N., Verenich, I., La Rosa, M., & Dumas, M. (2017, June). Predictive business process monitoring with LSTM neural networks. In *International Conference on Advanced Information Systems Engineering* (pp. 477-492). Springer, Cham.
50. 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.
51. Van der Aalst, W. M. (2005). Business alignment: using process mining as a tool for Delta analysis and conformance testing. *Requirements Engineering*, *10*(3), 198-211.
52. Van der Aalst, W.M.P. *Process Mining Discovery, Conformance and Enhancement of Business Processes.* Heidelburg, Germany: Springer, 2011.
53. 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.
54. Yamamoto, M., Sekine, S., & Matsumoto, S. (2017, January). Formalization of Karp-Miller tree construction on Petri nets. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (pp. 66-78). ACM.

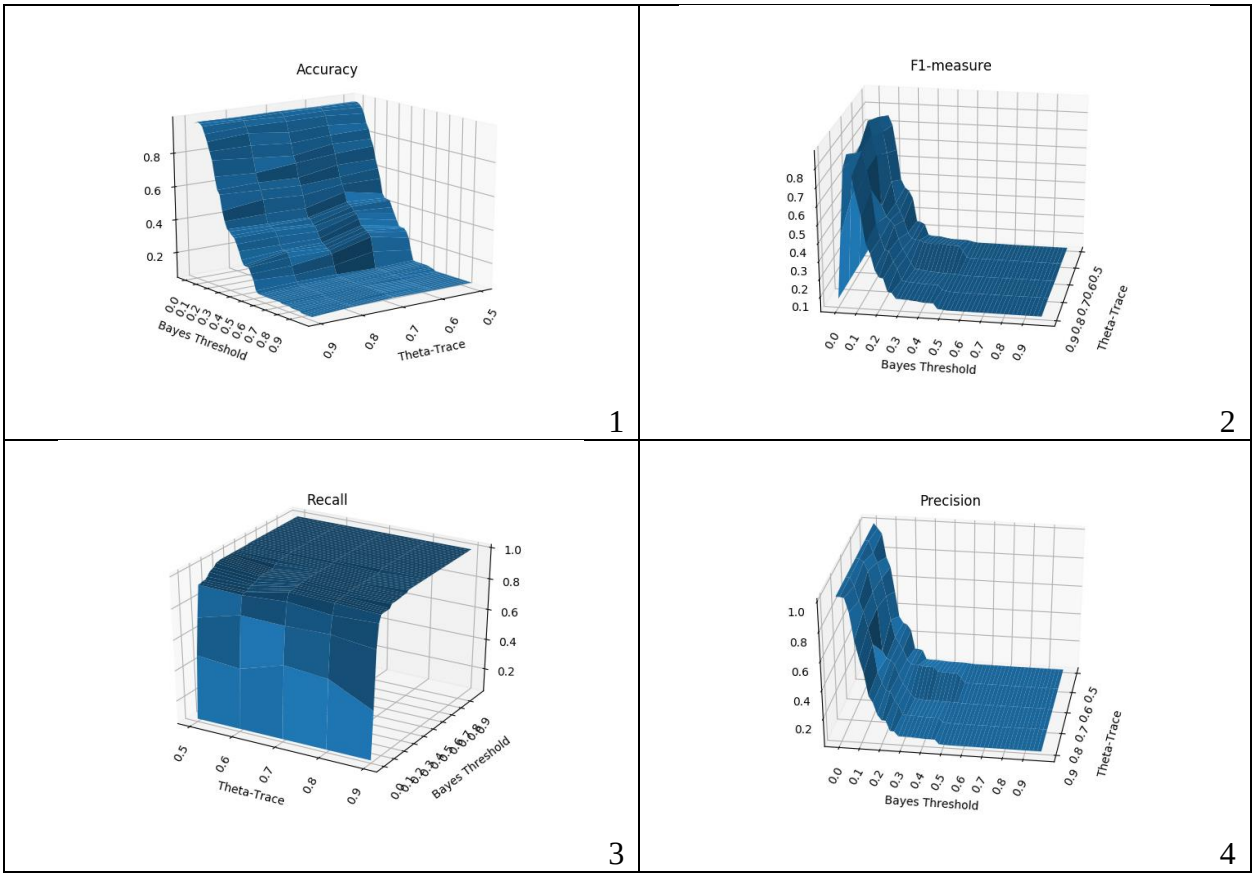
APPENDIX

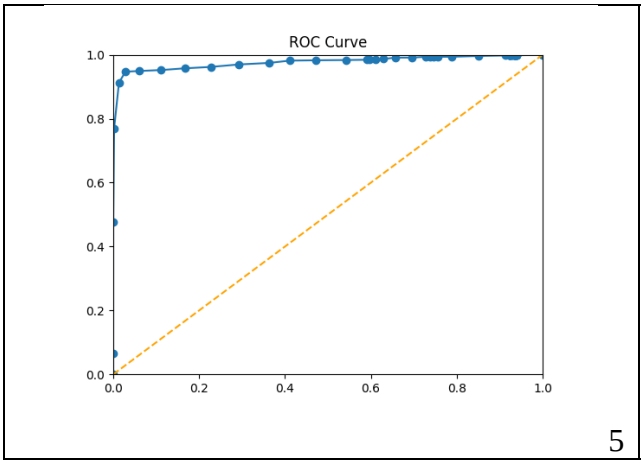
A1. Additional Experimental Results

This section contains the full results and visuals for experiments 1 thru 4. Some of the results are reproduced from the previous discussion, but additional results are also included for each experiment. For each experiment, the following are included:

- Curves for accuracy, recall, precision, and f1-measure
- Curves for raw true positive, true negative, false positive, and false negative counts
- Variance for accuracy, recall, precision, and f1-measures.

a. Experiment One Full Results





5

Figure A1. 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.

Performance metrics for accuracy, recall, precision, and f1-measure are shown in figure A1.

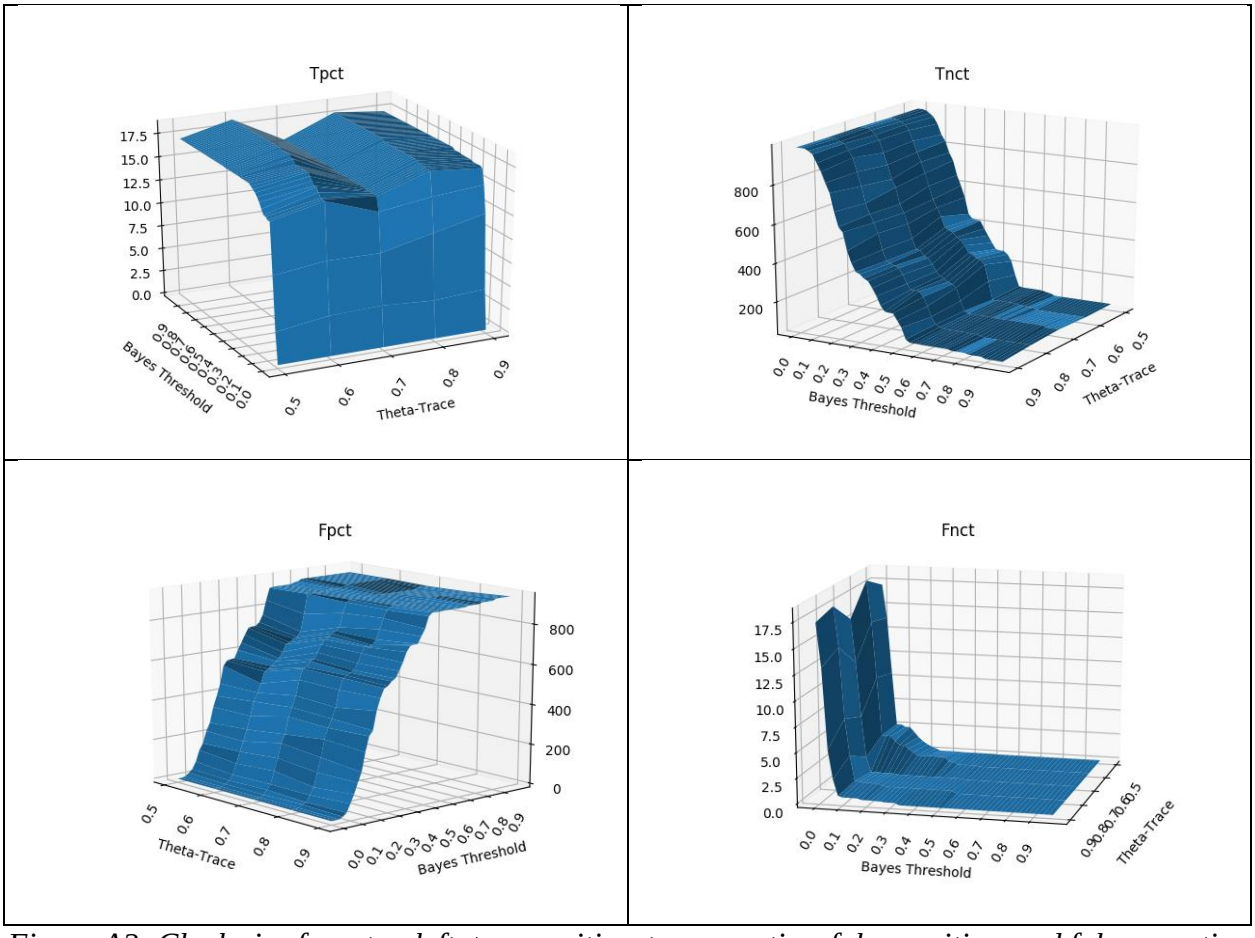


Figure A2: Clockwise from top left, true positive, true negative, false positive, and false negative count.

The raw true positive, true negative, false positive, and false negative counts are shown in figure A2. These curves elucidate the previous performance metrics, since the performance metrics are directly based on these values. As shown, the small values of $\boldsymbol{\alpha_{bayes}} \in [0.0, \sim 0.1]$ had the strongest influence in terms of either finding or failing to find anomalous traces. A slight bump along the theta-trace axis of the false-negative count curve, in the span of $\boldsymbol{\theta_{trace}} \in [0.5,0.6]$ and $\boldsymbol{\alpha_{bayes}} \in [0.0,0.4]$ demonstrates the algorithm suffering slightly due to greater trace diversity at these values.

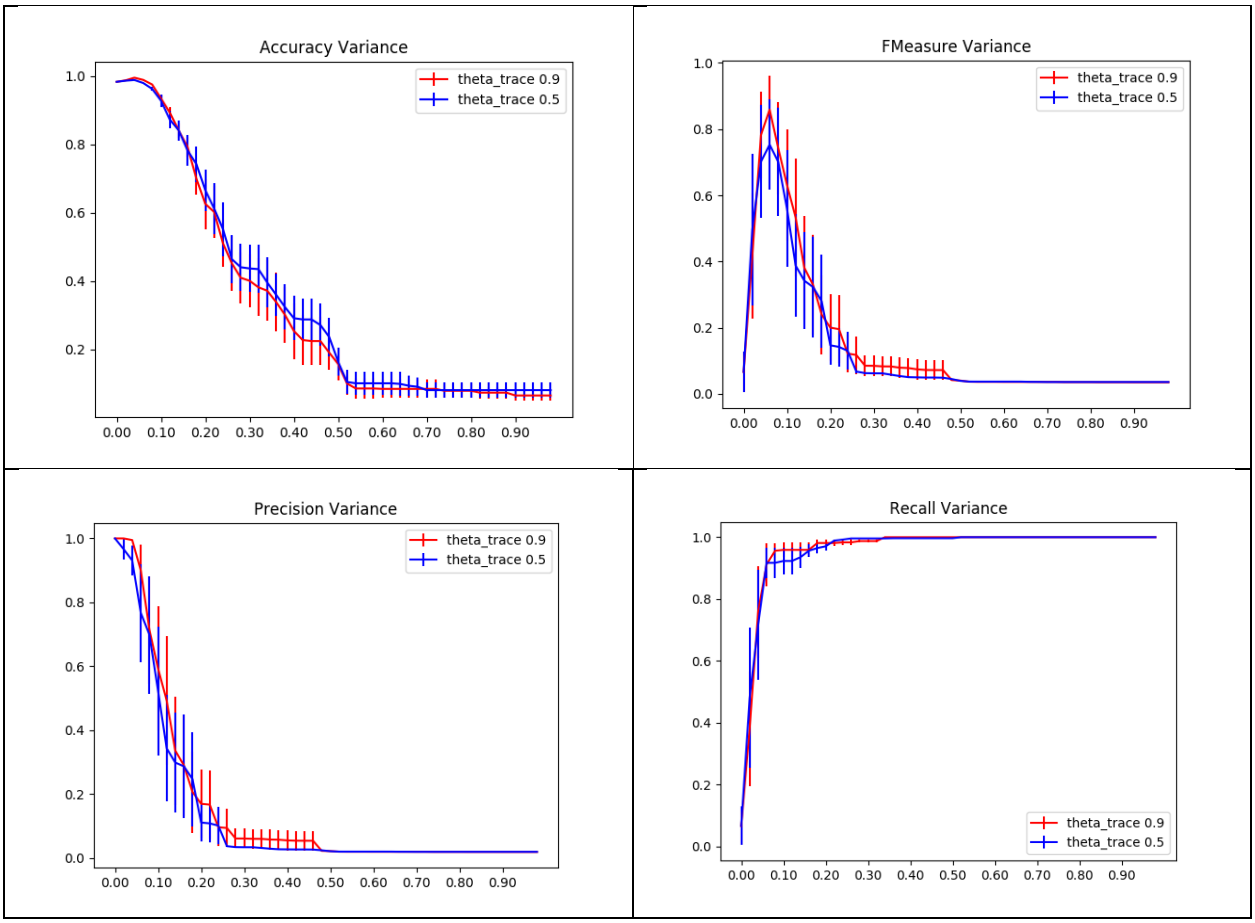
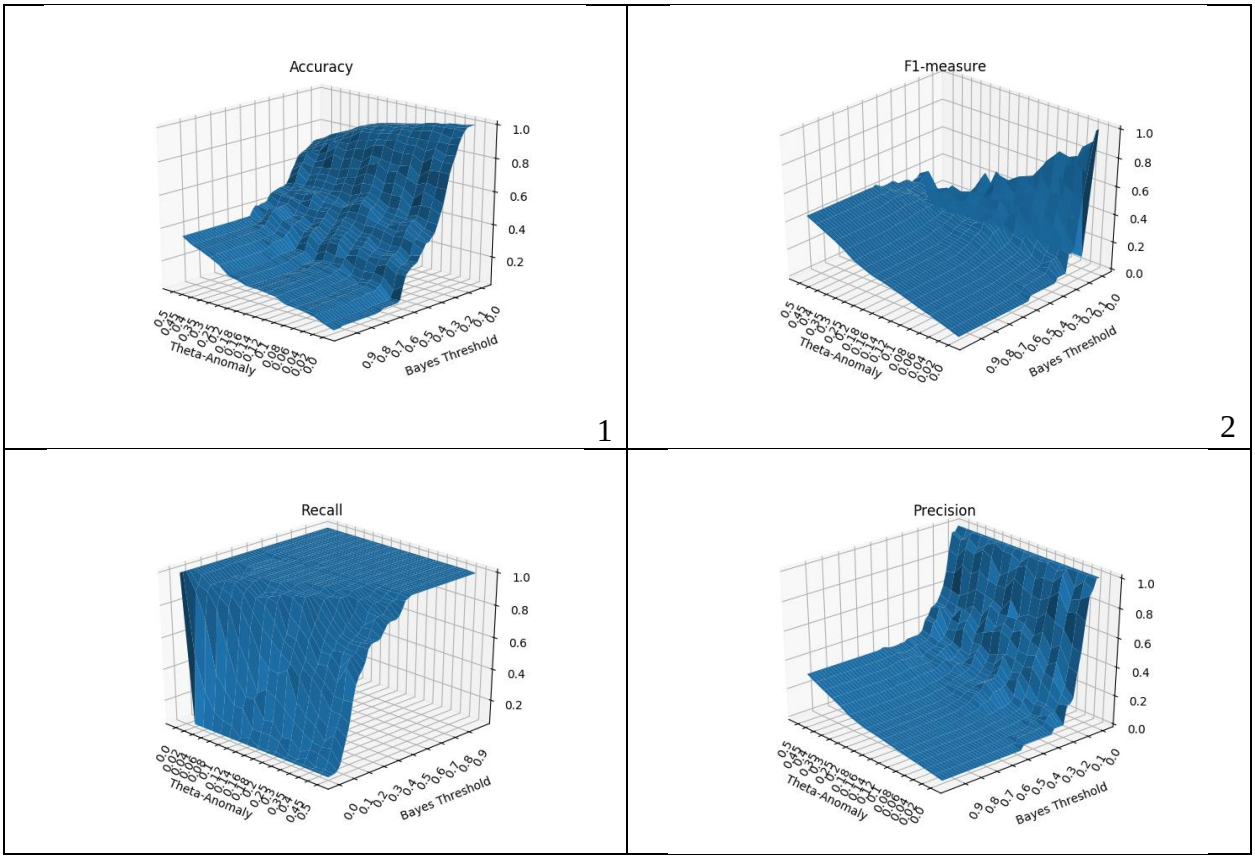


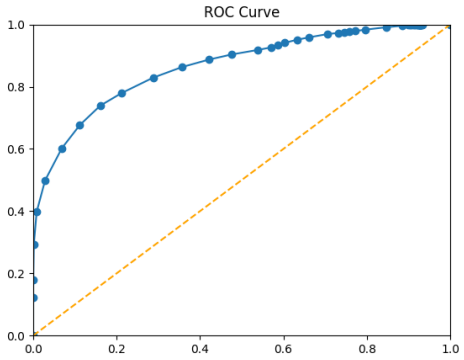
Figure A3: From top-left, accuracy variance, f1-measure variance, precision variance, and recall variance, plotted over the range of $\boldsymbol{\alpha_{bayes}}$ under test, for $\boldsymbol{\theta_{trace}} \in \{0.5, 0.9\}$.

The variances for different performance metrics are shown in figure A3. As shown, accuracy variance is quite low for reasonable values in the approximate range [0.05,0.15], when

averaged over the 60 models tested for each $\boldsymbol{\theta_{trace}}$ value. The importance of including these results is given by the recall and precision charts (and to the f1-measure to a lesser extent, since the f1-measure is calculated using recall and precision as input). The variance of recall and precision is quite significant, above 0.20, in the range $\boldsymbol{\alpha_{bayes}} \in [0.0,0.20]$. The cause of this variance is the nature of the data: if the algorithm fails to find a certain *type* of anomaly, that anomaly is usually shared by several or more traces. Thus, by failing to find any specific instance of an anomaly, a whole set of graphical traces is missed, resulting in a large punishment and significant variance. This effect is expected given the way that the data was generated, but it is worth noting given how it helps to characterize the performance of the algorithm in the context of the structural data with which it was evaluated.

b. Experiment Two Full Results



3	4
 5	*Figure A4: Dataset D1, 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).*

Performance metrics for accuracy, recall, precision, and f1-measure are shown in figure A4.

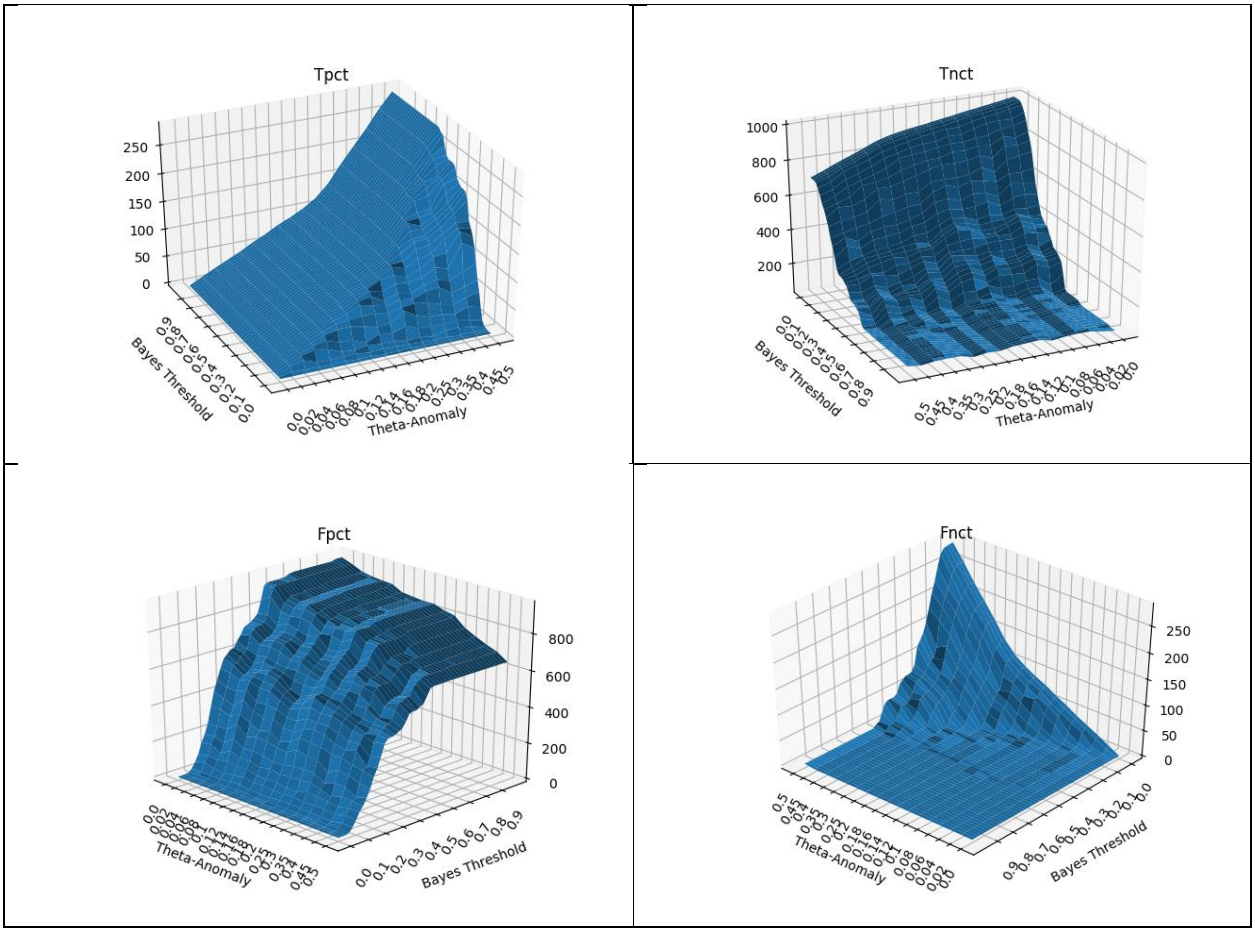
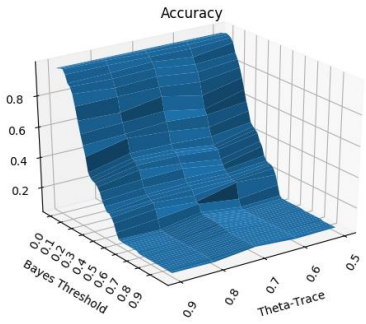


Figure A5: Clockwise from top left, true positive, true negative, false positive, and false negative counts plotted over a range of $\theta_{trace}, \alpha_{bayes}$ values.

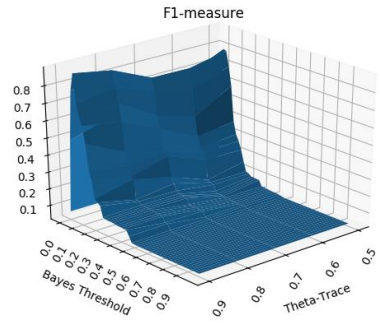
The raw true positive, true negative, false positive, and false negative counts are shown in figure A5. These curves explain the previous performance metrics, since the performance metrics are directly based on these counts. As shown, the small values of $\boldsymbol{\alpha_{bayes}} \in [0.0, \sim 0.1]$ had the strongest influence in terms of either finding or failing to find anomalous traces. The true positive/negative counts show steady performance even for large values of $\boldsymbol{\theta_{anomaly}}$, indicating the method continued to find anomalies even when they became very dense in the data. Perhaps the most important curve here is the false-negative count (figure A5, bottom right), showing a linear increase along the $\boldsymbol{\theta_{anomaly}}$ axis. The linearity along this axis implies the method's performance decayed in terms of a constant, however it also shows the potential for the method to provide unsatisfactory performance on certain datasets. Values of $\boldsymbol{\theta_{anomaly}}$ are a bit unreasonable since they imply data for which the distribution of traces and anomalies is very nearly the same, but they provide context for expected algorithmic performance in such extreme cases. In cases where a false-negative (type II error) is very high cost, and where the data exhibited an unusually high anomaly probability, one would need to use a high $\boldsymbol{\alpha_{bayes}}$ value to defend against false-negatives.

c. Experiment 3 Full Results

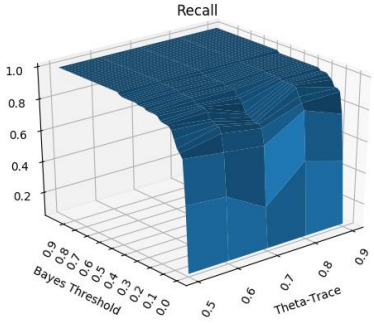
The full results for experiment 3 are nearly identical to those of experiment 1, since they tested the same model parameter, $\boldsymbol{\theta_{trace}}$, but for models with the modification of reusing existing activities for anomalies. The results are presented in figure A6 with minimal discussion, since the comments in (a) above apply in the same manner.



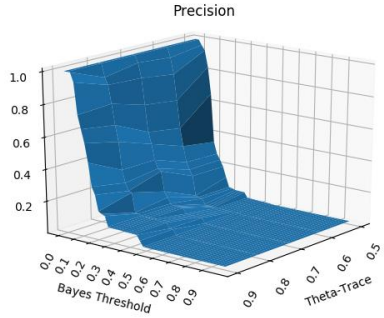
1



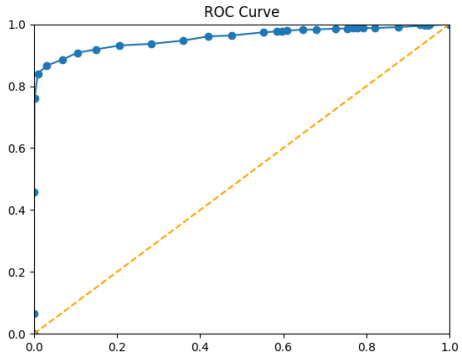
2



3



4



5

Figure A6: Dataset D2, experiment 1 results for (from top-left) accuracy (1), recall (2), precision (3), and f1-measure. The x/y axes of all curves were oriented for best visualization, so note their orientation carefully. ROC curve is shown at left (5).

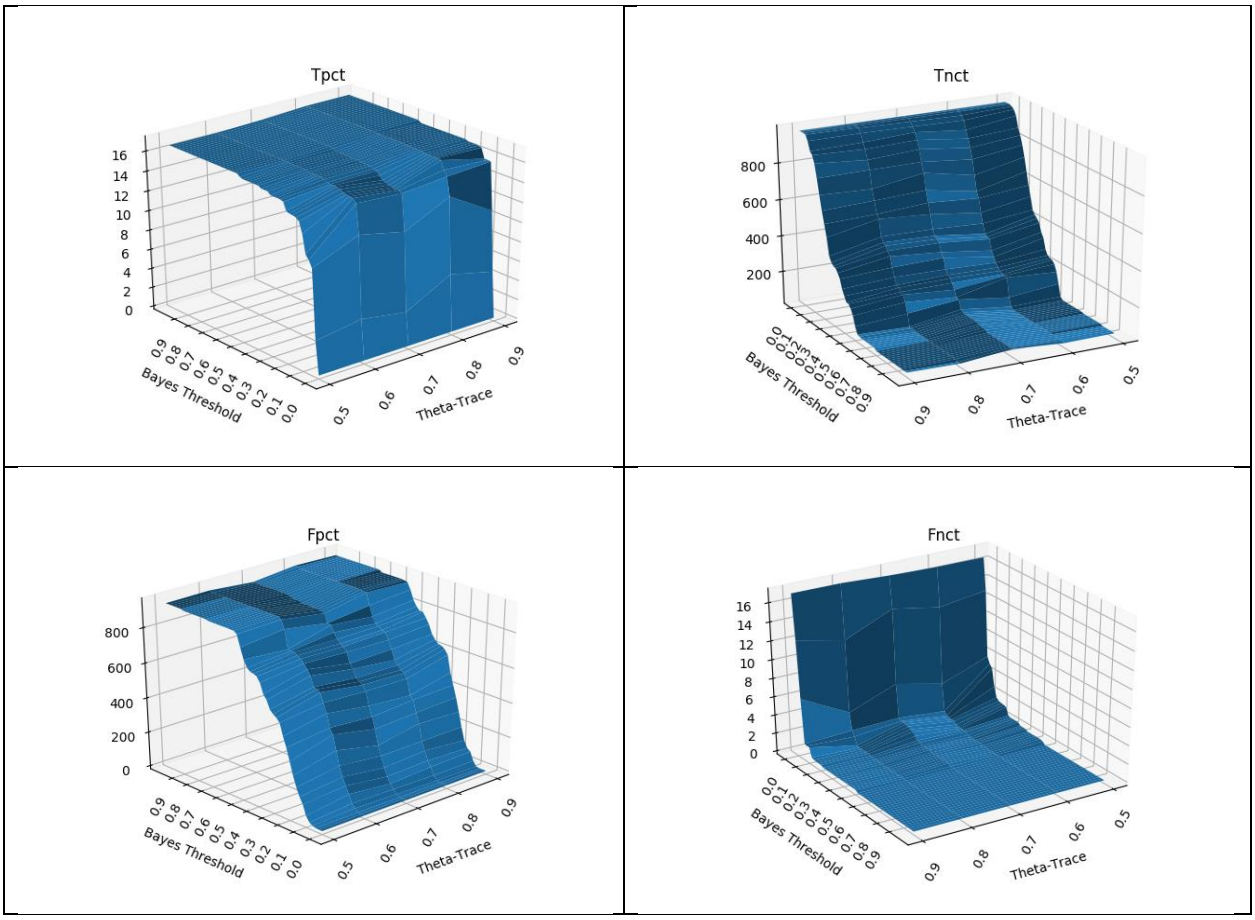


Figure A7: Clockwise from true positive, true negative, false positive, and false negative counts for experiment 3.

True positive, true negative, false positive, and false negative curves are shown in figure A7, averaged over the 60 logs tested each $\boldsymbol{\theta_{anomaly}}$ value, and show only slightly altered performance compared with experiment 1 results listed in Appendix 1.a.

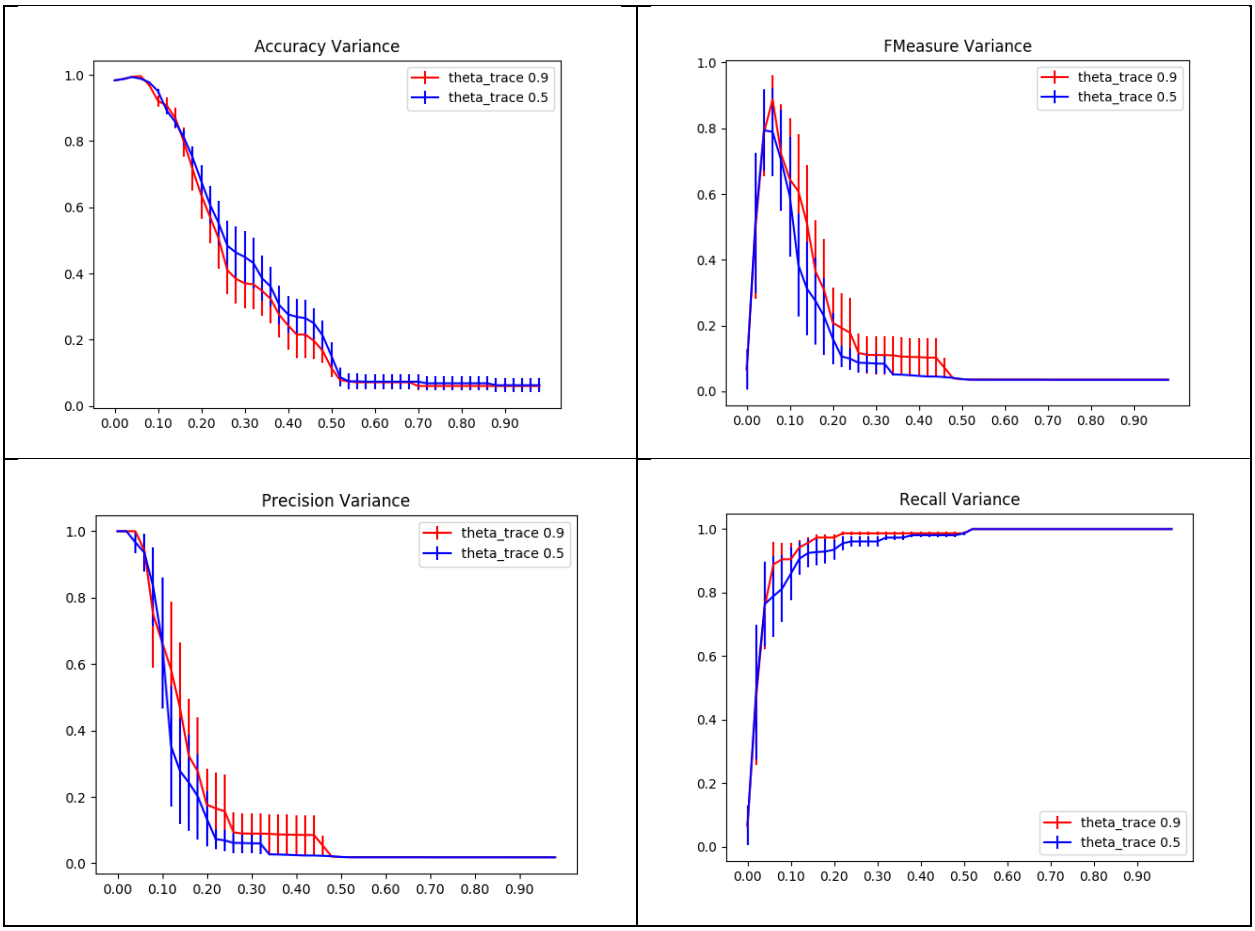
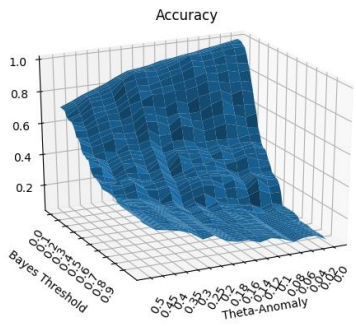


Figure A8: Accuracy, f1-measure, recall, and precision variance for experiment 3.

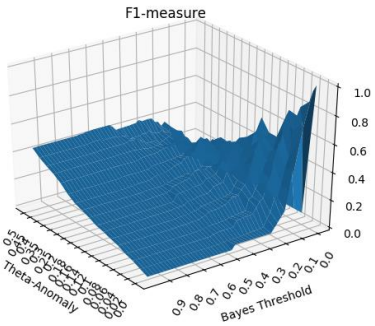
Test variance for accuracy, f1-measure, recall, and precision are shown in figure A8, and again differ little from experiment 1 results listed in Appendix 1.a.

d. Experiment Four Full Results

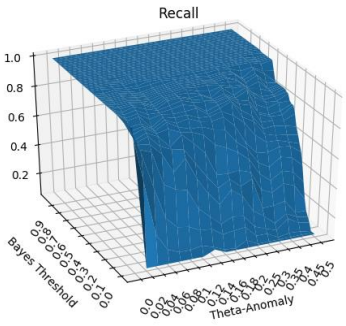
The results for experiment 4 differ very little from those of experiment 1 listed in section 1.b., and do not require additional comment. See section 1.b. for details.



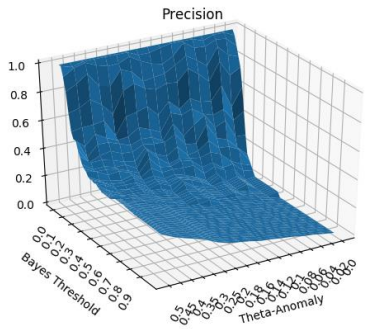
1



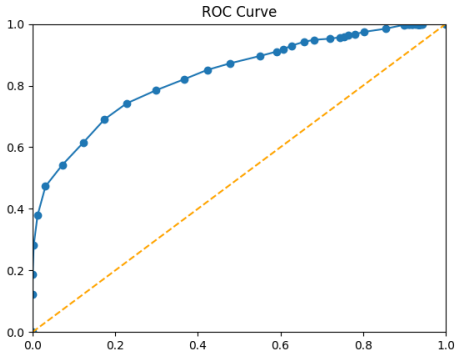
2



3



4



5

Figure A9. Performance results for D4, experiment 2 over a range of theta-anomaly values for (from top-left) accuracy (1), f1-measure (2), recall (3), and precision (4). Again, note that the horizontal x/y axes have been oriented to improve visualization. ROC curve is shown at left (5).

The raw true positive, true negative, false positive, and false negative curves differ very little from those found in section 1.c., as shown in figure A10.

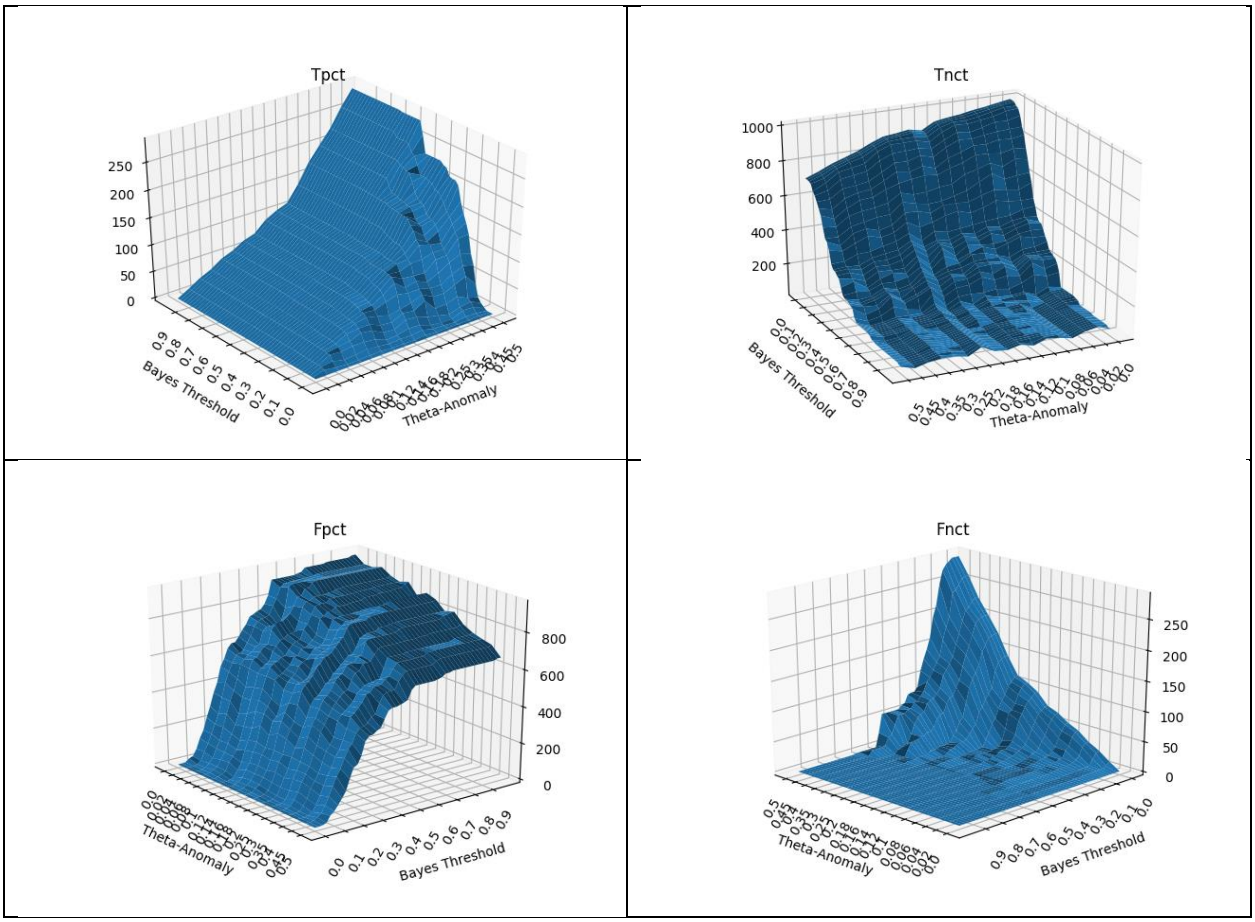
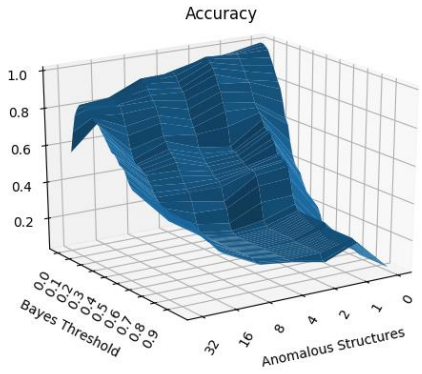


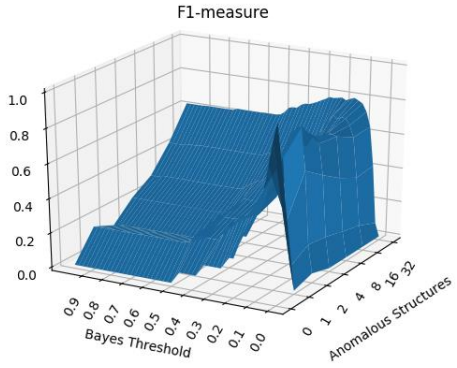
Figure A10: Clockwise from top left, true positive, true negative, false positive, and false negative curves for experiment 4.

e. Experiment Five Full Results

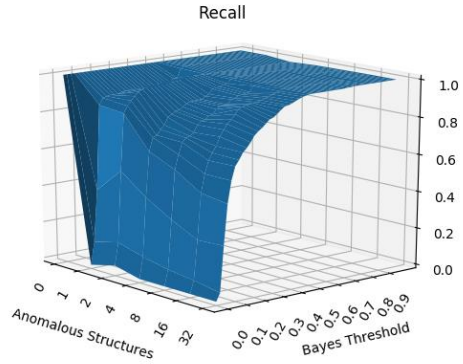
Performance curves for accuracy, recall, precision, and f1-measure are displayed in figure A11 for experiment five over a range of model anomaly quantities and $\boldsymbol{\alpha_{bayes}}$. The results were averaged over the 30 models generated and tested for each number of anomaly structures in $\{0, 1, 2, 4, 8, 16, 32\}$.



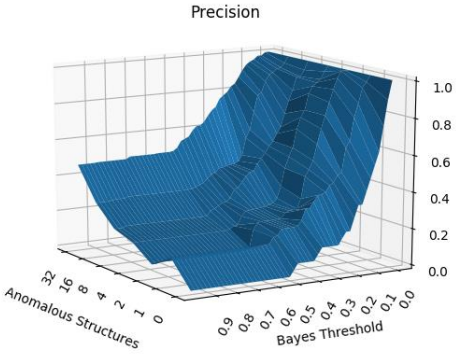
1



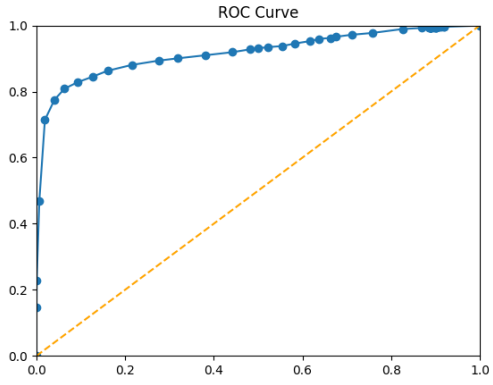
2



3



4



5

Figure A11. Performance results for D3, 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. The k parameter is given by the axes labeled "Anomalous Structures." ROC curve is shown at left (5).

Figure A12 shows curves derived by averaging the true positive, true negative, false positive, and false negative counts over the 30 models tested for each number of anomalous structures.

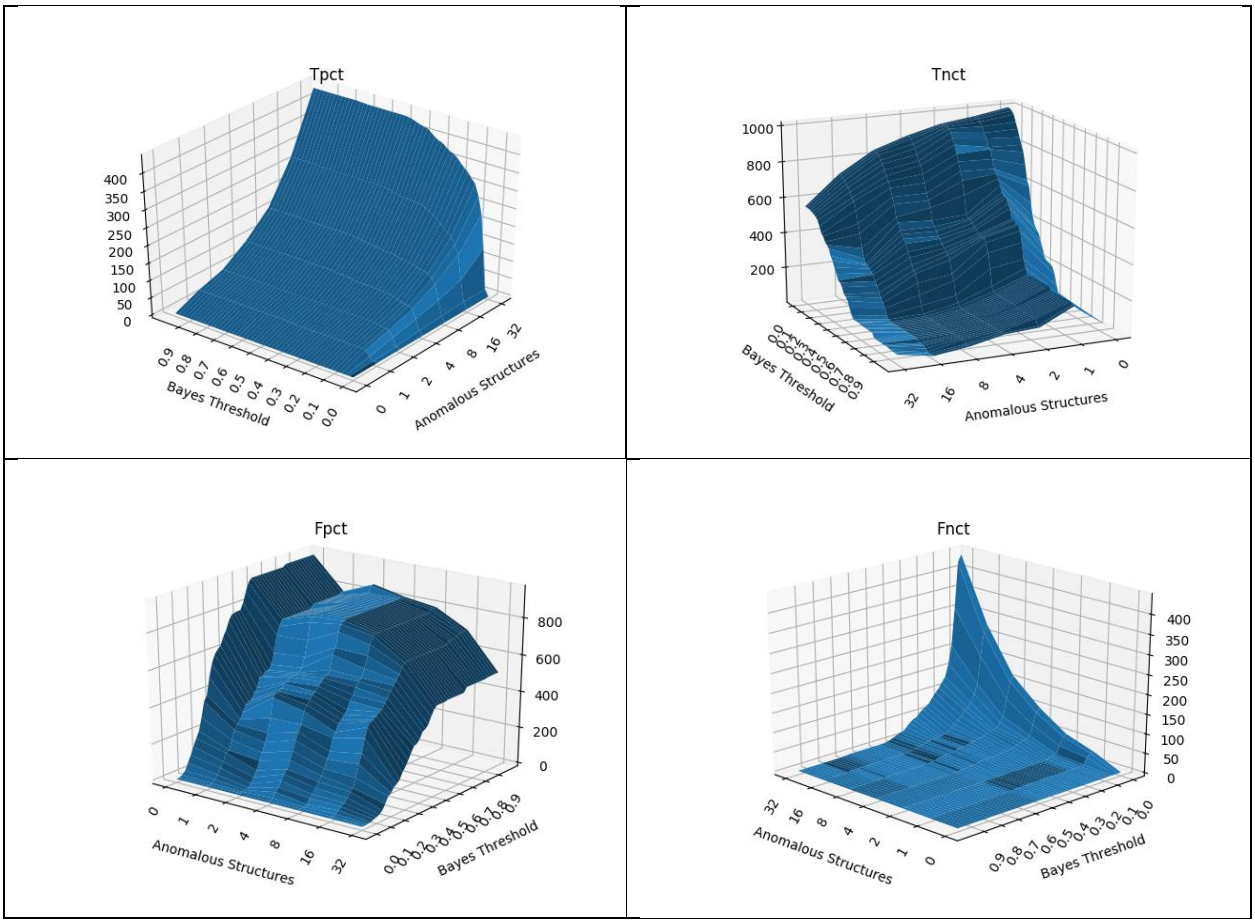


Figure A12: Clockwise from top left, true positive, true negative, false positive, and false negative counts per number of anomalous structures and α_{bayes}.

The true positive, true negative, false positive, and false negative rates are particularly important for this experiment, since the previous performance metrics are derived directly from these values. The dynamics shown in these curves give direct insight about the characteristic advantages and weaknesses of the method in experiment five, particularly for the reasonable parameter span defined by $\boldsymbol{\alpha_{bayes}} \in [0,0.1]$ and between 0 and perhaps 8 anomalous structures. Values outside this range were exhaustive for the sake of evaluation but are clearly extreme. The false-negative curve again shows a characteristic increase along the anomalous-structures axis in the direction of more numerous anomalous structures, but the curve here is shown in powers of

two. In linear space, the curves are linear, as shown in figure A13 for a slice across the anomalous structure axis at a reasonable $\boldsymbol{\alpha_{bayes}}$ value of 0.06.

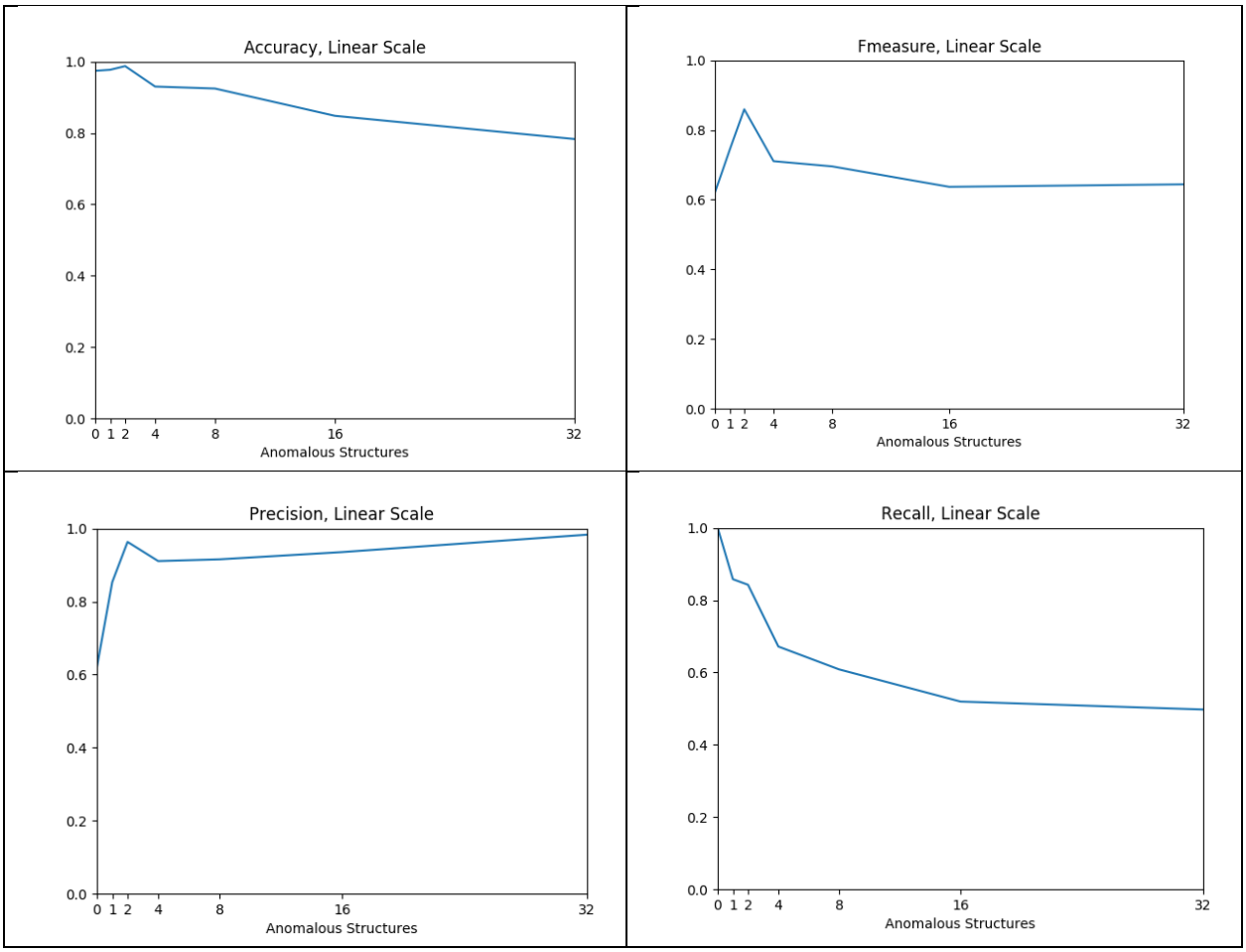


Figure A13: Performance curves in linear space for a selected α_{bayes} value of 0.06. Clockwise from top-left, accuracy, f1-measure, precision, and recall.The plots are merely a slice of the performance curves from prior for a fixed α_{bayes} value.

As shown in figure A13, the performance of the method maximizes for lower numbers of anomalous structures, less than 8 or so. The results demonstrate the level of effectiveness of this method for a modest density of anomalous structures; 16 or more anomalous structures are extreme values for the sake of exhaustive evaluation of the method.

A2. Sampling Algorithm Full Results

The full results for the Sampling Algorithm are included here for datasets D1 and D2. The results for dataset D1 were fully explained in a previous section, and the results for D2 differ very little, and don't require further commentary.

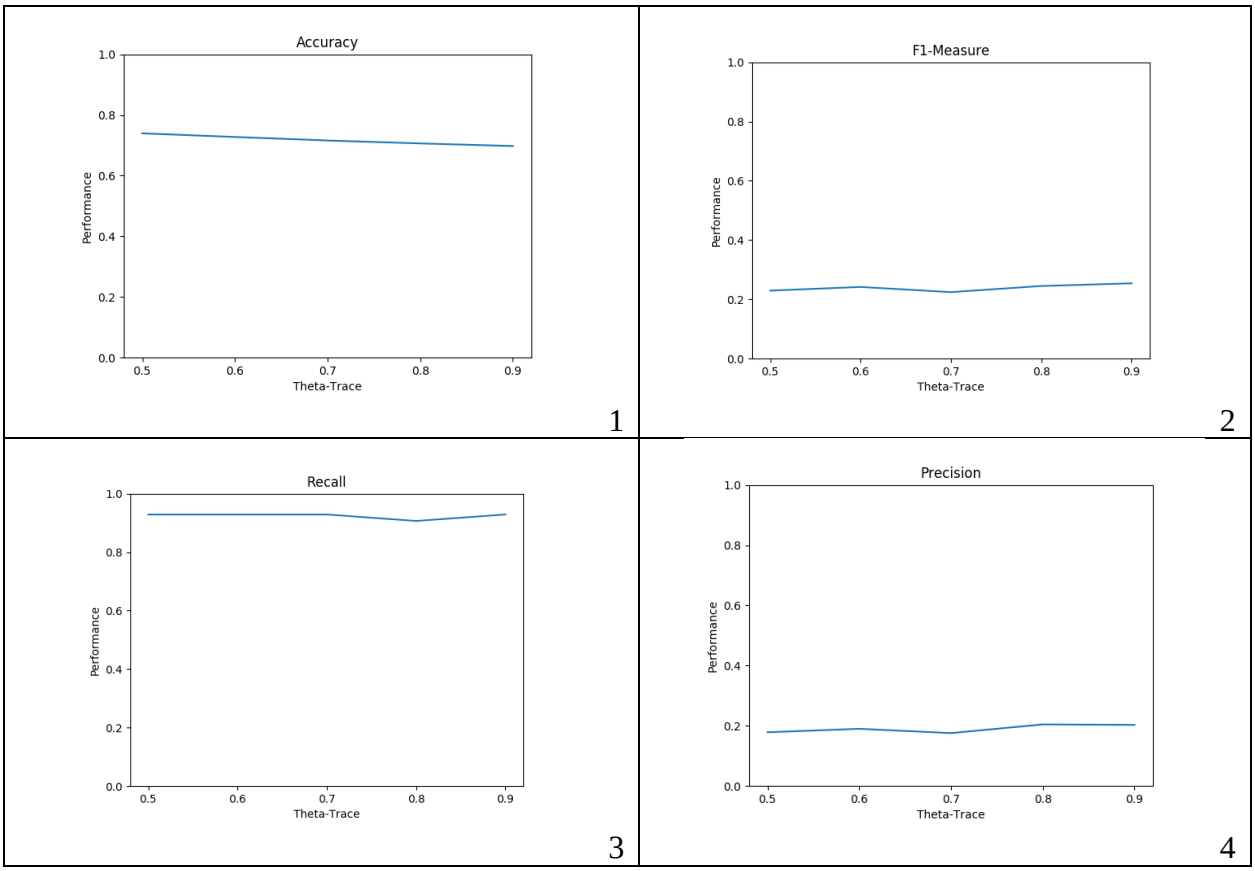


Figure A14: From top-left, dataset D1 Sample Algorithm results, accuracy (1), f1-measure (2), recall (3), and precision (4) for $\theta_{trace} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$.

The results for dataset D2 differed very little from D1, and are shown in figure A15 for documentation.

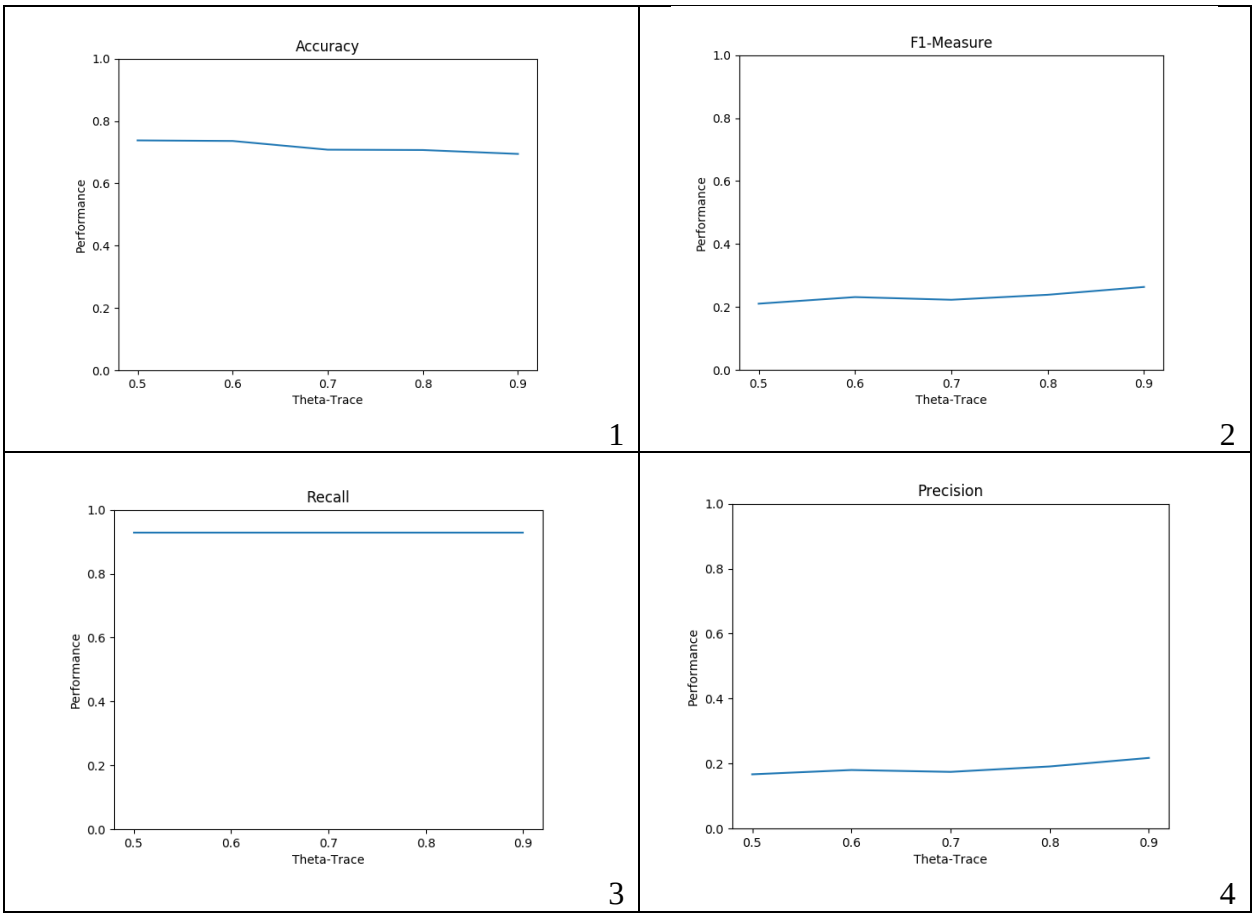


Figure A15: From top-left, dataset D1 Sample Algorithm results, accuracy (1), f1-measure (2), recall (3), and precision (4) for $\theta_{trace} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$.

A3. NASA CEV Software Test Log Full Results and Visuals

Since anomalies were not known in the NASA CEV software log dataset (Leemans, 2017), the output consisted of multiple visuals and graphs. A visual of the dendrogram, larger than the one used in a previous chapter, is shown in figure A16. As depicted by the graph layout method, the higher up components are more compressing, based on the order in which they were compressed. The directed edges indicate parent-child relationships between substructures, where the direction also indicates that the "child" was compressed later.

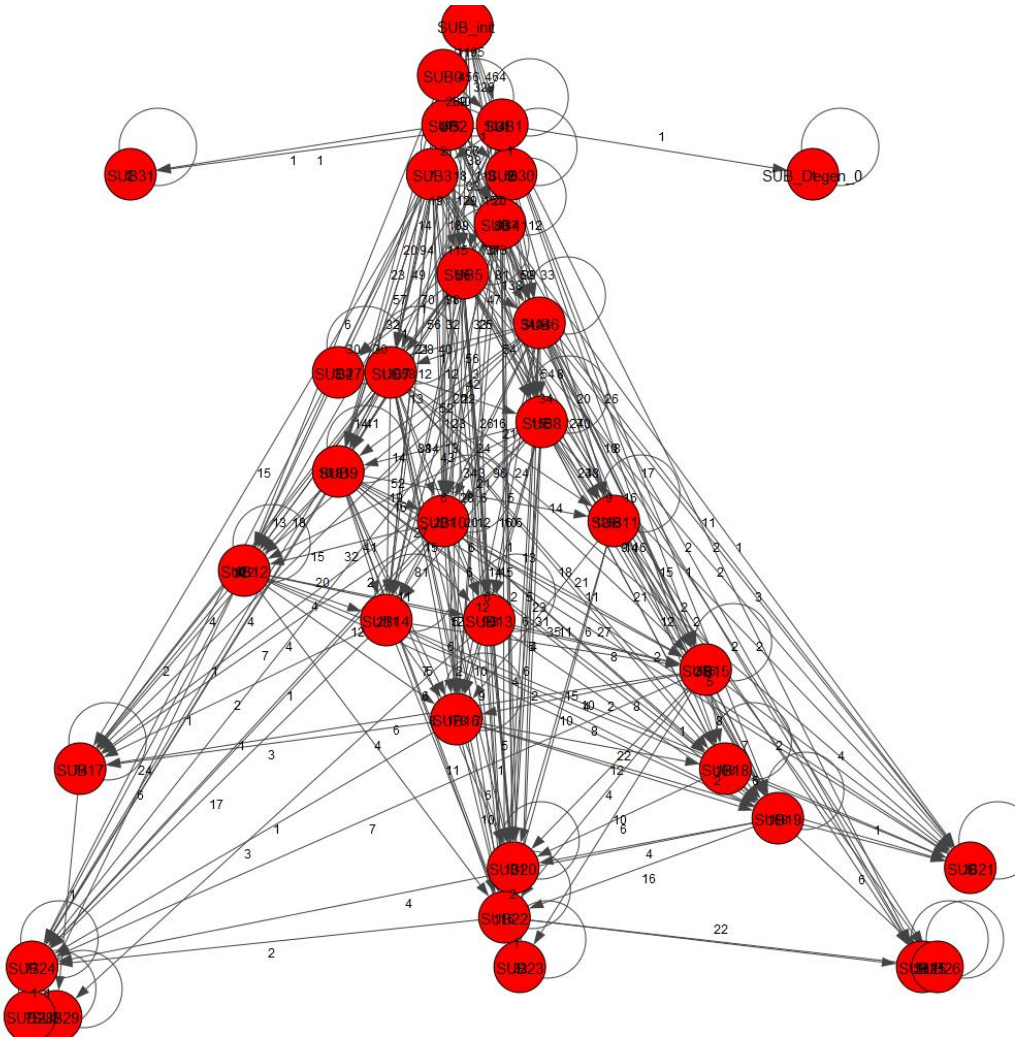


Figure A16: The dendrogram of substructures for the NASA CEV Software Test Log dataset.

The model, mined from the input log, is also shown in figure A17. The larger view and graphical layout of the model is to help demonstrate the internal structure of the software log, where each vertex represents a function, and is labeled as such.

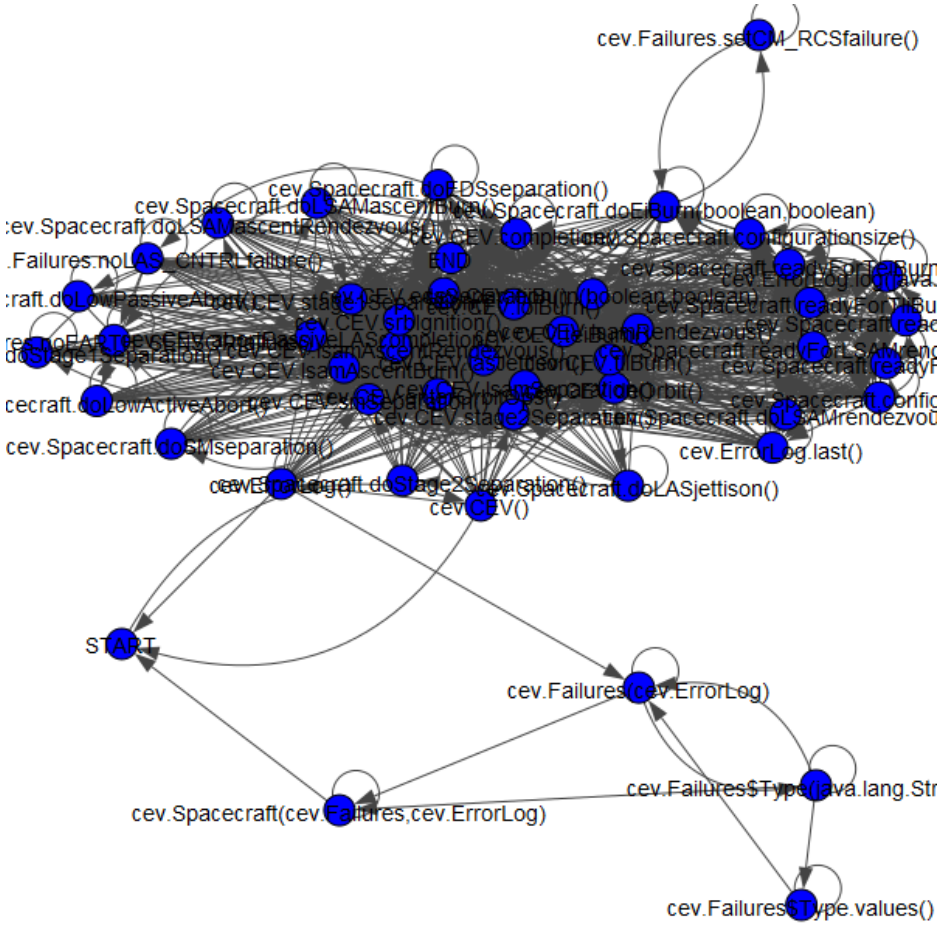


Figure A17: A larger view of the model mined from the NASA CEV software test log.

Each red vertex of the dendrogram represented a substructure, where each substructure is a component of the mined model above. An example substructure is shown in figure A18, and represents an anomalous component, which the call to an error function implies was due to the satisfaction of some error condition.

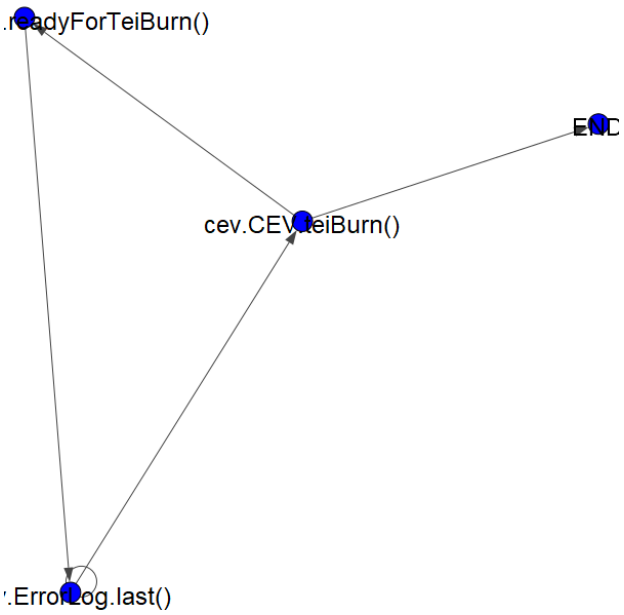


Figure A18: A compressing substructure of the previous dendrogram, representing a common code component.