LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT
FÜR DATENBANKSYSTEME

Bachelorarbeit
in Informatik

Efficient Event Classification through Constrained Subgraph Mining

Simon Lackerbauer

Aufgabensteller:	Prof. Dr. Peer Kröger
Betreuer:	Martin Ringsquandl
Abgabedatum:	16. Februar 2018

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 16. Februar 2018

.....
Simon Lackerbauer

Abstract

With this work, I consider the problem of discovering patterns in error log data predicting common failure modes within a near fully-automated assembly line. I present a novel approach of encoding error log events in a graph structure and leverage a constraint based mining method to efficiently discover and score sophisticated patterns in these data, using a self implemented version of the pattern-growth algorithm *gSpan*. As the algorithm as implemented does not scale quite as well as many traditional sequential pattern mining approaches, outside expert knowledge should be used to keep the input data to a manageable size and help with graph construction.

Contents

Chapter 1

Introduction

The ability to analyze log files is a crucial part in the work of systems administrators and software developers. Developers often enough outright provoke the generation of detailed logs while debugging a piece of software, while system and network operators are routinely pulled from their efforts to fine tune services by incessant alerts from their monitoring systems.

Indeed, transaction log files, a common subgroup of the more general event logs, have been used to diagnose errors in the workings of data processing systems nearly since their inception, and dedicated research efforts into the analysis of transaction log files can be traced back until at least the mid-1960s.[1]

With the decline of monolithic service architectures, and the recent rise of complex systems of interdependent microservices,[2] the accurate reading of log files and recognition of the patterns within has become more important than ever. Nowadays, a plethora of log file management and search tools exist as both open source and commercially licensed tooling.[3][4][5][6] More sophisticated log file analysis that goes beyond simple exploratory data analysis and simple monitoring and alerting systems is, however, not usually the focus of these tools.

But not only modern microservice architectures need powerful log file analysis tools to understand where bottlenecks and emergent properties of the architecture might stem from. That's why, with this bachelor's thesis, I want to leverage these techniques with a slightly more traditional, yet similarly highly modularized, architecture in mind: an automated production line.

Modern automated production lines usually consist of highly sophisticated robotic modules that assemble each produced unit with an efficiency and consistency that would be virtually impossible for human workers to achieve. However, at the same time, such automated systems are less tolerant of errors accumulating along the way and might stop working for relatively mundane

reasons like a part being slightly out of position on the assembly line. Unlike a human worker, a specialized robot cannot solve most of these problems by itself. Depending on the problems encountered, the amount of time and (human) effort needed to deal with them can drive up the cost of running the line enormously, maybe even up to the point of operating at a loss.

If problems like these crop up consistently, it is only natural to assume a common cause between propagated failures along the assembly line[7] and it can be assumed that, after identifying such causes early on and mitigating them, propagation of failures might be reduced or entirely averted. The production line analyzed in this thesis had the goal to optimize their efficiency as well.

Thus, the aim of this thesis will be to work on this problem using a graph mining based approach, starting with an overview of the literature (chapter 2) and the theoretical basis of graph mining (chapter 3), reflecting on how to extract a graph data structure from the data available as a log table (chapter 4.2), how best to mine the resulting graph or graphs for patterns (chapter 4.5), and eventually how to assess the resulting patterns (chapter 5.3.1).

Chapter 2

Related work

Leveraging a graph mining based approach for mining log data constitutes a relatively novel use for most of the algorithms implemented in this work. Graph mining as a concept, has, up to this point, been mostly used on data that naturally lend themselves to a graph or network based data structure, such as social networks, modeling human relationship networks in general, chemical component analysis or link networks.[8][9] Meanwhile, mostly sequential pattern mining approaches have been leveraged against access or error log based data, such as the *basket data* collected by most large retailers nowadays[10] or *web access logs*[11]. The *gSpan* algorithm was originally tested on synthetic graph data as well as for mining chemical compound data.[12]

Most of the above mentioned naturally occurring graph data don't include time as a feature having measurable impact on the depicted relations. Chemical compounds may change and degrade over time, but their graph representations usually portray their idealized form. Social networks have also been of interest as dynamic processes themselves,[13] but these analyses tend to focus on snapshots of the full network after longer periods of time, whereas the data set examined in this work is a classical time series, with each row explicitly time stamped down to the second.

2.1 Sequential pattern mining

Sequential pattern analysis is a staple approach in time series mining. The term was first introduced and defined by Agrawal and Srikant[14] as follows:

> [G]iven a sequence database where each sequence is a list of transactions ordered by transaction time and each transaction consists of a set of items, find all sequential patterns with a user-specified minimum support, where the support is the number of

data sequences that contain the pattern.

More formally, let $I = \{i_0, i_1, ..., i_n\}$ be a set of all items. Then a k-itemset I^*, which consists of k items from I, is said to be *frequent* if it occurs in a transaction database D no less than $\theta|D|$ times, where θ is a user-specified *minimum support threshold* (often abbreviated *min_sup*).

Mining sequential patterns can take the form of a-priori algorithms (see subsection 2.2.1), like Srikant and Agrawal's GSP[15] or pattern growth algorithms, like Yan and Han's *gSpan*[12] used in this work.

2.2 Classical graph mining approaches

The classical graph mining approach often focuses on obtaining general structural information about a network. As an example, the mining of social networks often reveals an overall structure of almost-leaves (e.g. friend- or kinship groups) being connected via so called "multiplicators" or "influencers" – nodes with a high degree centrality[16] that are also interconnected with each other, together acting as the central cluster in a basically star-shaped network, which offers an explanation for the small world problem[17].

Meanwhile, for this work, the overall structure of the graph (the dependency network between modules and inputs for the assembly line) is known beforehand, and even formally defined. Analysis of the design of the found substructures – viz. *what* the pattern symbolizes as opposed to *finding* it in the first place – is indeed only of interest after most of the work is already done.

2.2.1 A-priori based approach

The A-priori based paradigm to frequent pattern mining is a heuristic that generates a reduced set of patterns through each iteration.[11] The a-priori principle, on which these approaches are based on, states that *any super-pattern of an infrequent pattern cannot be frequent.*[18] Thus, these algorithms first generate a set of all frequent 1-element sequences. From that, they generate new candidate sequences in a step-wise way. For example, if the patterns A and B are each frequent according to a specific *min_sup*, then the pattern AB might be frequent as well. These generated patterns are then tested and discarded if and only if they don't reach the given *min_sup* (cf. algorithm 1, where T is the transaction database and C_k is the candidate set for sequence length k).

Well-studied a-priori-based algorithms include the already mentioned GSP[15], SPADE[19], or HitSet[20]. As a-priori-based approaches have to search through

Algorithm 1 APriori(T, min_sup)[14]

1: $L_1 \leftarrow \{\text{large 1-itemsets}\}$;
2: $k \leftarrow 2$
3: **while** $L_{k-1} \neq \emptyset$ **do**
4: $\quad C_k \leftarrow \{a \cup \{b\} \mid a \in L_{k-1} \wedge b \notin a\} - \{c \mid \{s \mid s \subseteq c \wedge |s| = k-1\} \nsubseteq L_{k-1}\}$
5: $\quad$ **for each** $t \in T$ **do**
6: $\quad\quad C_t \leftarrow \{c \mid c \in C_k \wedge c \subseteq t\}$
7: $\quad\quad$ **for each** $c \in C_t$ **do**
8: $\quad\quad\quad count[c] \leftarrow count[c] + 1$
9: $\quad\quad$ **end for**
10: $\quad$ **end for**
11: $\quad L_k \leftarrow \{c | c \in C_k \wedge count[c] \geq min\_sup\}$
12: $\quad k \leftarrow k + 1$
13: **end while**
14: **return** $\bigcup_k L_k$;

the input data at least once for every candidate pattern, the best a-priori algorithms achieve runtime efficiencies of $O(n^2)$.

2.2.2 Pattern-growth based approach

In contrast, the pattern-growth approach adopts a divide-and-conquer principle as follows: *sequence databases are recursively projected into a set of smaller projected databases based on the current sequential pattern(s), and sequential patterns are grown in each projected database by exploring only locally frequent fragments.*[18] The *gSpan*-algorithm used in this work is, like its predecessors *FreeSpan* and *PrefixSpan*, such a pattern-growth based approach.

Chapter 3

Theoretical basis

The proposed event classification method mines a graph set $\mathcal{D} = (G_0, ..., G_n)$ for substructures of interest, often called "patterns" in the literature. To establish a firm theoretical understanding of what the these structures are, some definitions of graph theory are in order.

3.1 Graph

Traditionally, a graph is represented as a set of vertices V and a set of edges E, with a mapping $f : E \to V \times V$ that defines each edge as a tuple of vertices. In directed graphs, this tuple is ordered. In undirected graphs, it is unordered, so that $\forall v_i, v_j : (v_i, v_j) \in E \to (v_j, v_i) \in E$. In this work, the vertex set V of a graph G may also be denoted $V(G)$. Likewise, the edge set may be denoted $E(G)$. To encode these structures in software, adjacency matrices and edge lists are commonly deployed. The following example adjacency matrix (table 3.1) and visual representation (figure 3.1) all encode the same undirected graph $G = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_2)\})$.

Table 3.1: Adjacency matrix for graph G

	v_1	v_2	v_3	v_4
v_1	0	1	0	0
v_2	1	0	1	1
v_3	0	1	0	1
v_4	0	1	1	0

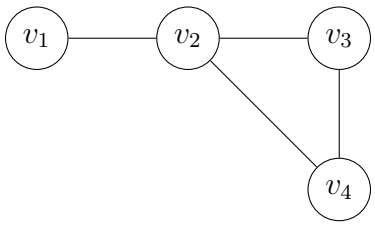

Figure 3.1: Visual representation of graph G

3.2 Subgraph

Let G_s and G be two graphs, where $G_s = (V_s, E_s)$ and $V_s \subset V(G), E_s \subset E(G)$. Then, if the following holds:

$$\forall (v_i, v_j) \in E_s \implies v_i, v_j \in V_s,$$

G_s is said to be a subgraph of G. Figure 3.2 illustrates an example subgraph of G with three of G's four vertices and two of its four edges. Note that the graphical representation need not be drawn in the same way to depict a subgraph.

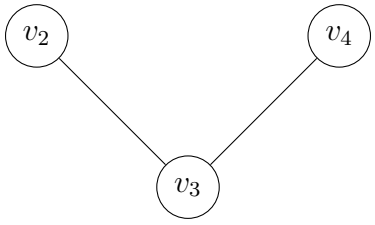

Figure 3.2: Example subgraph G_s of G

3.3 (Sub-)Graph isomorphism

Two graphs G and H are said to be isomorph when the following holds: let $f : V(G) \to V(H)$ be a bijection and let vertices u and v of G be adjacent in G. Then $G \simeq H$ if and only if $f(u)$ and $f(v)$ are adjacent in H. It is currently unknown if the graph isomorphism problem is P or NP.[21]

The subgraph isomorphism problem is the decision problem of whether, when given two graphs G and H, there exists a subgraph in G that is isomorphic to H. The subgraph isomorphism problem is known to be NP-complete.[22].

Chapter 4

Methodology

Especially considering time as both an explicit attribute of nodes and edges, as well as implicitly encoded into the graph structure and as such still present in subsequently mined subgraphs, constitutes an approach to exploring log data through graph mining that has not been examined extensively before.

4.1 Disregarded approaches

Before homing in on analyzing the given data set with *gSpan* and a *Support Vector Machine*, some approaches with lesser quality results were taken. Even though these approaches didn't prove fruitful in the long run, they helped me understand the data set and its underlying structure better and as such are, with their basic premises, included here to give a full account of all measures taken.

4.1.1 Shortest path algorithms on a single graph

As the manufacturing process consists of a circuit of interconnected modules, the first approach of translating the given data into a graph form was to build one single large graph with all information available around this first circuit of module nodes, with the intention of later on mining frequent patterns from this single large graph, using algorithms such as Elseidy et al's *GRAMI*[23] or Moussaoui et al's *POSGRAMI*[24].

To encode the log error data on top of this framework, the error messages were split into terms (essentially words) and each term made a node. Edges were drawn between all term-nodes in a given message, as well as between all term-nodes and the module-node they occurred in, as well as between all term-nodes and the production-unit-ID-node. This first naive approach of visualizing the available data produced a graph consisting of 523 nodes connected

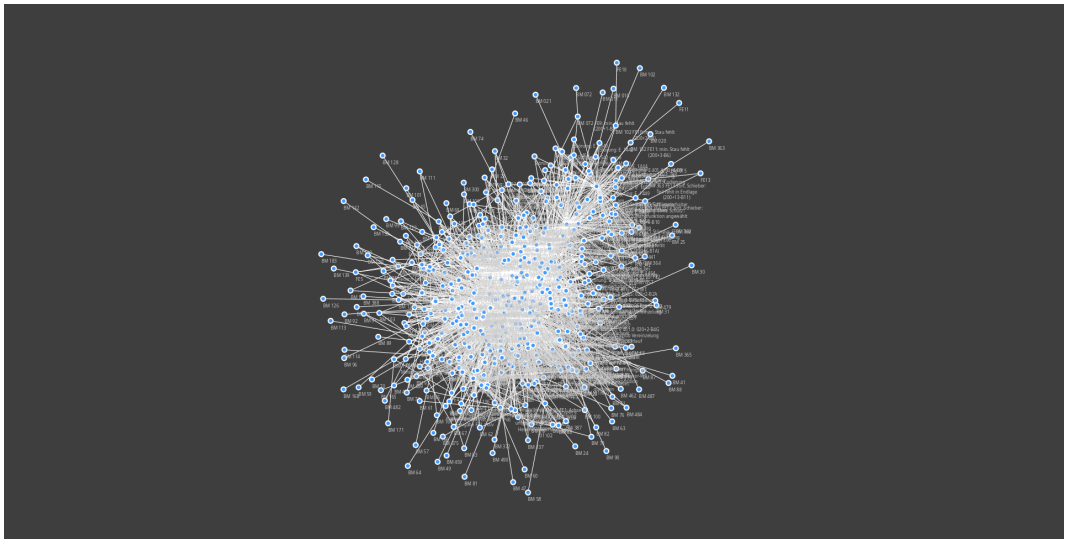

Figure 4.1: Naive single graph encoding all available information

by 2,182 edges (illustrated in figure 4.1). Edges were weighted simply by a count of how often they appeared throughout the whole data set, with some edges only appearing once and a maximum weight of over 10,000 appearances.

This first graph did not, to a first approximation, retain its expected circuit-like appearance, instead clustering heavily around a few modules and production unit IDs that produced the most errors, meaning there were parts in the system vastly more error prone than others.

In a second step, the Dijkstra shortest paths algorithm was used to find all paths between nodes that had a full path length of less than a specific constant, afterwards ordered by path length. For example, a short 4-edge path could connect two modules and the part ID via two error terms, indicating some kind of correlation, much like a sequential pattern analysis could find.

For Dijkstra to work, the naive edge weight for edge i ($w_{i_{naive}}$) had to be transformed from simple counts to a normalized form that would retain comparison operations between edges, but invert path lengths (as the Dijkstra algorithm is used, as its name suggests, to find the *shortest paths*). This was achieved by the following transformation being calculated after graph generation, with $w_{all} = \sum_{j=0}^{n} w_{j_{naive}}$:

$$w_{i_{normalized}} = -\ln\left(\frac{w_{i_{naive}}}{w_{all}}\right)$$

This weight was later penalized further by adding time constraints, so that

$$w_{i_{penalized}} = w_{i_{normalized}} + P(i)$$

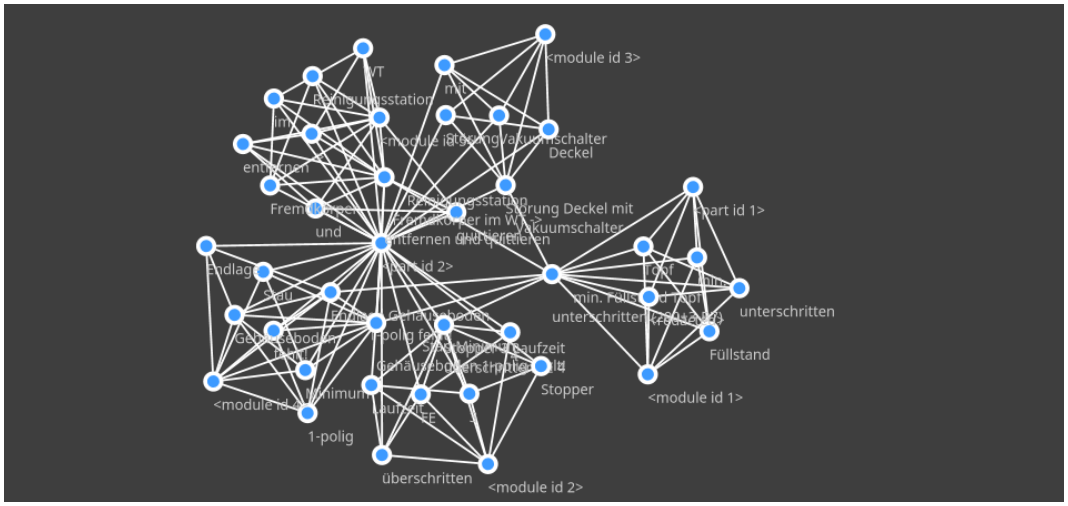

Figure 4.2: Large example pattern

where

$$P(i) = \begin{cases} \log_{\tilde{\Delta}t} \frac{\Delta t}{C(\Delta t) \cdot \tilde{\Delta} t}, & \text{if } C(\Delta t), \tilde{\Delta} t > 0 \\ 0, & \text{otherwise} \end{cases}$$

Here, Δt is the amount of time between rows being connected by the edge, $C(\Delta t)$ is the amount of connections between these nodes, and $\tilde{\Delta} t$ is the average time differential between rows overall. This ensured penalization for long times between events (indicating that the events weren't correlated), while similarly rewarding events that happened more often in conjunction.

4.1.2 Digraph

Using a directed graph was briefly considered, but did, in some first tests, not lead to noticeably different patterns from the undirected graph. This was almost expected, as graph connections were highly predictable, because of the construction of the graphs similar to dependency graphs.

If, for example, in one splice, specific terms had a directed edge *towards* a specific module, then this basic structure would obviously be repeated in other slices as well, by virtue of a construction that always pointed terms towards modules. The simple connectedness attribute that is an edge in an undirected graph already served this purpose implicitly when taking into account this background knowledge used during graph construction.

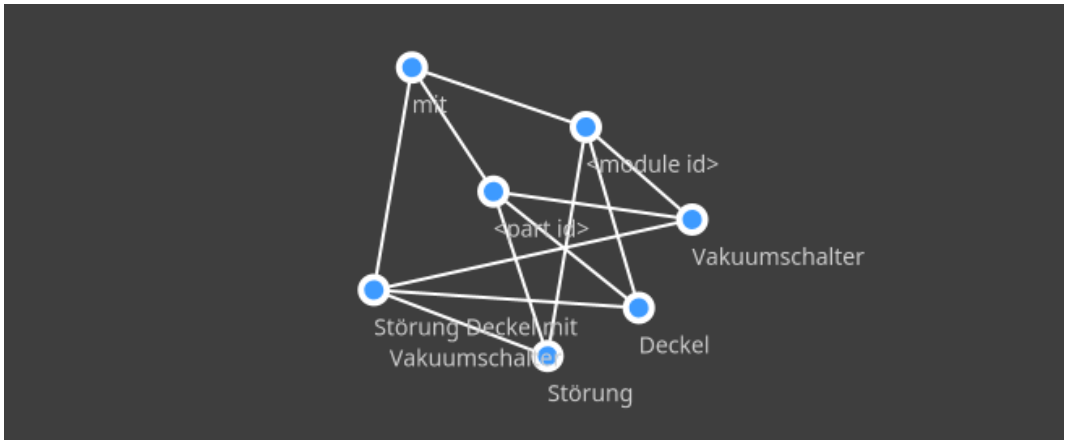

Figure 4.3: Smaller example pattern

4.1.3 Natural language processing

The idea of combining both techniques of graph mining and a natural language processing unit to make sense of the actual contents of the error logs was briefly entertained as well. Good, easily available NLP software for languages other than English is still quite hard to come by, however. In addition, the analyzed error logs were often very short and technical, and thus didn't lend themselves to an actual content based analysis. Even to native speakers, many of the messages would've been quite cryptic and so it was decided that the processing power that would've been needed for an NLP based approach would be better used elsewhere.

4.2 Data set splicing

The investigated data set consisted of about 57,000 messages logged over the course of five consecutive days in October 2016. Available columns were a time stamp (precision: 1s), an unstructured log message in German, the module ID where the message originated and the part ID of the produced item in that run. Messages sometimes included additional partly structured data, like a more detailed report of the location where the error occurred included in the German log message. See table 4.1 for an example of the data structure and a part of the synthetic data set used for the testing setup in section 5.2.

The data had to be cleaned up slightly prior to a first cursory visual analysis. An, at first glance, large amount of the total message count consisted of (without expert knowledge) seemingly meaningless general error messages consisting of only an error code and no further explanation. These specific

Table 4.1: Synthetic data set (excerpt)

time stamp	log message	module id	part id
2017-04-05 11:01:05	Laser überhitzt	Module 1	88495775TEST
2017-04-05 11:01:05	Laser überhitzt	Module 1	88495776TEST
2017-04-05 11:01:06	Teil verkantet	Module 2	88495776TEST
2017-04-05 11:01:06	Laser überhitzt	Module 1	88495776TEST
2017-04-05 11:01:10	Laser überhitzt	Module 1	88495776TEST
2017-04-05 11:01:12	Auffangbehälter leeren	Module 2	88495775TEST
2017-04-05 11:01:17	Unbekannter Ausnahmefehler	Module 0	88495775TEST
2017-04-05 11:01:17	Auffangbehälter leeren	Module 2	88495775TEST
2017-04-05 11:01:19	Unbekannter Ausnahmefehler	Module 0	88495775TEST
2017-04-05 11:05:22	Laser überhitzt	Module 1	88495775TEST
⋮	⋮	⋮	⋮

messages existed in 4 slightly different formats of about 30 messages with the same time stamp each, with the error code incremented by one with each message. Each instance of these 30 message bursts was replaced with a simple "general error" message with the same time stamp. After this preprocessing step, the amount of messages to be considered had roughly halved.

Further, if the module ID and manufacturing unit ID were included in the message in one of several standardized ways, they were extracted and given their own column in the data set to prohibit random integers cropping up in the messages to be mistaken for e.g. a module id.

To generate the graph set to be mined, a week's log data was spliced along 5 minute time frames, producing graphs of the size like exemplified in figure 4.4. Two different modules throwing errors in this specific time window can be easily distinguished. A 3 minute window was briefly considered, but the resulting graphs were, almost surprisingly, much smaller overall, so that the mined patterns weren't open to any meaningful interpretation and were fewer in number as well. Windows smaller than 5 minutes in general mostly led to many more small graphs, so that no long pattern could conceivably ever reach any useful *min_sup* threshold.

4.3 Basic graph construction

Extracting a basic graph structure from the input is both one of the least computationally intensive steps in the proposed methodology and the most important one. The analyzed data sets don't lend themselves to a natural scheme; background knowledge about the facility had to play an important part in determining the basic structure of the graphs.

As previously mentioned, the constructed graphs heavily relied on known

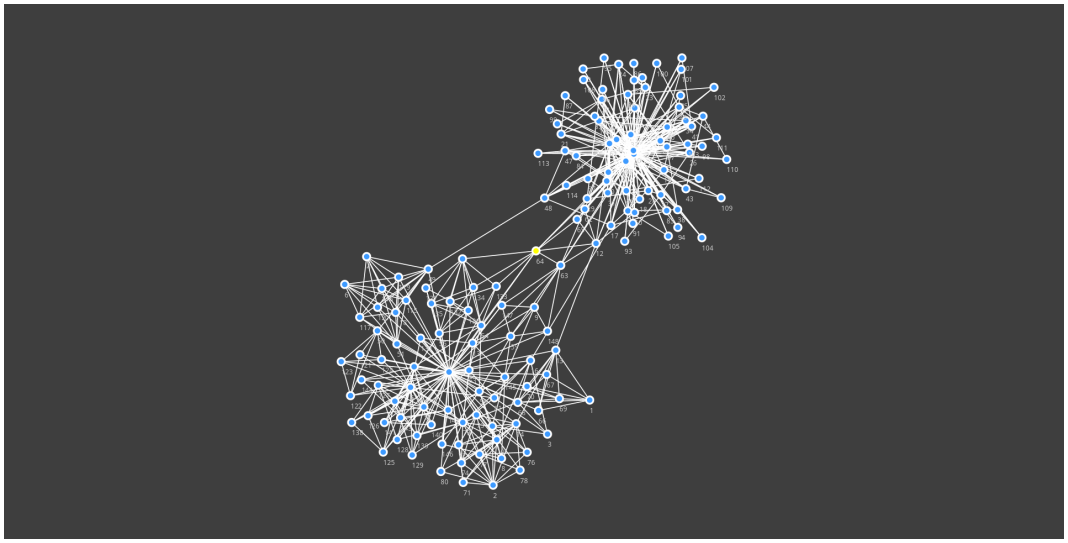

Figure 4.4: Graph of a 5-minute log data window

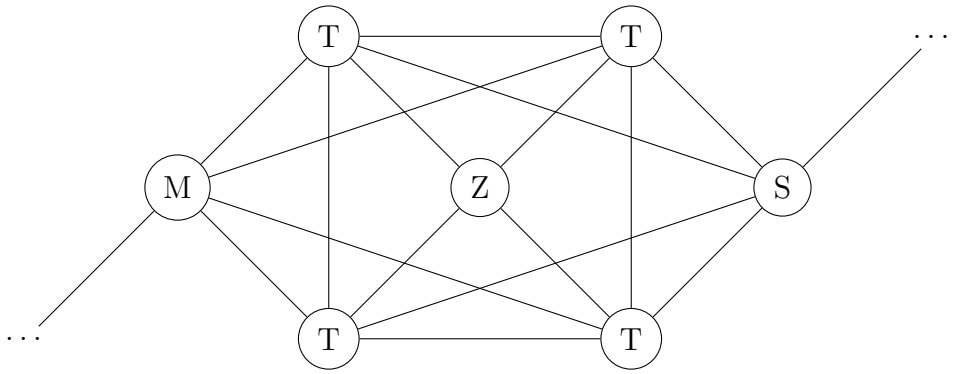

Figure 4.5: Example slice

dependencies between error messages, modules and part IDs. Figure 4.5 illustrates the common connections between the full error message Z, its terms T, the error logging module M and the specific part ID S. Error messages could be connected by using common words between them, through, e.g., a further narrowed down standardized localization term, while modules and part ids would be connected to all error messages and their terms produced with their involvement.

4.3.1 Using background knowledge during model construction

In the case of production facilities, the basic structure of the facility itself, even more so in a modularized system like the one on hand, has to be integrated into the basic graph scheme. Considering inherent parallelisms in the system (like two modules working in parallel), already places a few constraints on the resulting input graphs. Constraints were mostly input manually into the system prior to graph construction.

4.4 Features of interest

As mentioned before, the production facility doesn't run at peak performance most of the time. As such, of interest to this analysis was mostly if specific events or sequence of events would be able to predict a decrease in the OOE figure. A sequence analysis performed by my advisor some time before this thesis had already yielded some preliminary results in this direction which were considered known problems by the experts. Stumbling upon a pattern which would be considered a novel mechanical problem to solve would be the prime result of this work.

4.5 Modified gSpan

gSpan was first introduced by Yan and Han in 2002.[12][25] *gSpan* leverages depth-first search (DFS) to map graphs to minimum DFS codes, which are a canonical lexicographic graph labeling method. As all isomorphic graphs have the same canonical label, once computation of the labels is completed, it's trivially easy to solve the isomorphism question for any two graphs by comparing their canonical labels. If those labels are available in a lexicographic format, their comparison in itself is also trivially achievable by simple string comparison. The modification of *gSpan* in this work is based on adding a hash function to make the lexicographic comparison faster.

Introducing a hashing algorithm obviously introduces the risk of collisions between hashes, rendering the formerly unambiguous canonical label to graph mapping no longer bijective, but instead surjective. In the case of structured or semi-structured operator controlled inputs, however, the theoretical possibility of collisions because of the hashing shouldn't be a huge concern.

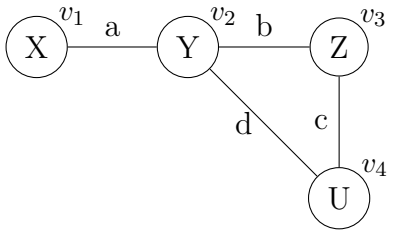

Figure 4.6: Graph G from chapter 3 with labels

4.5.1 DFS Lexicographic Order

To demonstrate the construction of a minimum DFS code, we're again using the example graph from chapter 3, this time with additional labels for nodes and edges as visualized in figure 4.6.

Mapping this graph to a minimum DFS code using algorithm 2 yields the minimum DFS code in table 4.2.

Table 4.2: Minimum DFS code of graph G

edge no.	DFS code
0	$(0, 2, U, d, Y)$
1	$(1, 2, X, a, Y)$
2	$(0, 3, U, c, Z)$
3	$(2, 3, Y, b, Z)$

Algorithm 2 MinDFSCode(G)

1: initiate $S \leftarrow \emptyset$
2: **for each** vertex $v \in V(G)$ **do**
3: perform a depth-first search with v as a starting point;
4: transform the resulting DFS tree t into a DFS code tuple;
5: **end for**
6: sort DFS code tuples by comparing their length according to DFS lexicographic order and choose the smallest one as the canonical label

The DFS lexicographic order is a linear order defined by the less or equal function in algorithm 3. For the neighborhood restrictions and the comparison function between two DFS code tuples $a = (i_a, j_a, l_{i_a}, l_{(i_a,j_a)}, l_{j_a})$ and $b = (i_b, j_b, l_{i_b}, l_{(i_b,j_b)}, l_{j_b})$, please see algorithm 4. In both algorithms, the following definitions apply: $\alpha = (a_0, a_1, ..., a_m)$ and $\beta = (b_0, b_1, ..., b_n)$, where each a_t, b_t is a DFS code tuple of the form $x_t = (i_x, j_x, l_{i_x}, l_{(i_x,j_x)}, l_{j_x})$. i_x, j_x are vertices, l_{i_x}, l_{j_x} are their labels, and $l_{(i_x,j_x)}$ is the edge label.

Algorithm 3 DFSLexicographicLE(α, β)

if $n \geq m$ and $a_m = b_m$ **then**
    return True, ie $\alpha \leq \beta$
else
    $a_{\text{forward}} \leftarrow \text{Bool}(j_a > i_a)$
    $b_{\text{forward}} \leftarrow \text{Bool}(j_b > i_b)$
    $a_{\text{backward}} = \neg a_{\text{forward}}$
    $b_{\text{backward}} = \neg b_{\text{forward}}$
    if $a_{\text{forward}} \wedge b_{\text{forward}}$ **then**
        return True, ie $\alpha \leq \beta$
    end if
    if $a_{\text{backward}} \wedge b_{\text{backward}} \wedge j_a < j_b$ **then**
        return True, ie $\alpha \leq \beta$
    end if
    if $a_{\text{backward}} \wedge b_{\text{backward}} \wedge j_a = j_b \wedge l_{(i_a,j_a)} < l_{(i_b,j_b)}$ **then**
        return True, ie $\alpha \leq \beta$
    end if
    if $a_{\text{forward}} \wedge b_{\text{forward}} \wedge i_b < i_a$ **then**
        return True, ie $\alpha \leq \beta$
    end if
    if $a_{\text{forward}} \wedge b_{\text{forward}} \wedge i_b = i_a \wedge l_{i_a} < l_{i_b}$ **then**
        return True, ie $\alpha \leq \beta$
    end if
    if $a_{\text{forward}} \wedge b_{\text{forward}} \wedge i_b = i_a \wedge l_{i_a} = l_{i_b} \wedge l_{(i_a,j_a)} < l_{(i_b,j_b)}$ **then**
        return True, ie $\alpha \leq \beta$
    end if
    if $a_{\text{forward}} \wedge b_{\text{forward}} \wedge i_b = i_a \wedge l_{i_a} = l_{i_b} \wedge l_{(i_a,j_a)} = l_{(i_b,j_b)} \wedge l_{j_a} < l_{j_b}$ **then**
        return True, ie $\alpha \leq \beta$
    end if
    return False, ie $\alpha > \beta$
end if

Algorithm 4 DFSTuplesLexicographicLE(a, b)

$a_{\text{forward}} \leftarrow \text{Bool}(j_a > i_a)$
$b_{\text{forward}} \leftarrow \text{Bool}(j_b > i_b)$
$a_{\text{backward}} = \neg a_{\text{forward}}$
$b_{\text{backward}} = \neg b_{\text{forward}}$
if $a_{\text{forward}} \wedge b_{\text{forward}} \wedge a_j < b_j$ **then**
    return True, ie $a \leq b$
end if
if $a_{\text{backward}} \wedge b_{\text{backward}} \wedge (a_i < b_i \vee (a_i = b_i \wedge a_j < b_j))$ **then**
    return True, ie $a \leq b$
end if
if $a_{\text{backward}} \wedge b_{\text{forward}} \wedge a_i < b_j$ **then**
    return True, ie $a \leq b$
end if
if $a_{\text{forward}} \wedge b_{\text{backward}} \wedge b_j \leq a_i$ **then**
    return True, ie $a \leq b$
end if
if a_{backward} **then**
    if $b_{\text{forward}} \wedge b_i \leq a_i \wedge b_j = a_i + 1$ **then**
        return True, ie $a \leq b$
    end if
    if $b_{\text{backward}} \wedge b_i = a_i \wedge a_j < b_j$ **then**
        return True, ie $a \leq b$
    end if
end if
if a_{forward} **then**
    if $b_{\text{forward}} \wedge b_i \leq a_j \wedge b_j = a_j + 1$ **then**
        return True, ie $a \leq b$
    end if
    if $b_{\text{backward}} \wedge b_i = a_j \wedge b_j < a_i$ **then**
        return True, ie $a \leq b$
    end if
end if
return False, ie $a > b$

With these comparison algorithms in place, a *DFS Code Tree* can be constructed. In a *DFS Code Tree*, each node represents one graph via its DFS code. Obviously, in such a tree, the DFS code for a graph can turn up more than once, depending on node addition order. Thus, the first code that turns up on a pre-order depth-first search of the *DFS Code Tree* is what we previously called the minimum DFS code.

This results in the more formal definition given by [25]:

> Given a graph G, $Z(G) = \{code(G,T)|\forall T, T \text{ is a DFS code}\}$, based on DFS lexicographic order, the minimum one, $\min(Z(G))$, is called **Minimum DFS Code** of G. It is also the canonical label of G.

4.5.2 Graphset projection and subgraph mining

The pattern growth approach now becomes clearer when we're trying to construct a new pattern from an already found one: to construct a valid DFS code for the new pattern, the new edge cannot be added at an arbitrary position, but can only be added to vertices on the "rightmost path." This is further limited, as only forward edges can grow from all vertices on the rightmost path, whereas backward edges can only be grown from the rightmost vertex.

With these definitions in place, the *gSpan* algorithm works as follows (see algorithm 5 for the pseudocode):

Algorithm 5 GraphSet_Projection($\mathcal{D}, \mathcal{S}$)[25]

```
1: sort labels of the vertices and edges in D by their frequency;
2: remove infrequent vertices and edges;
3: relabel the remaining vertices and edges in descending frequency;
4: S^1 ← all frequent 1-edge graphs in D;
5: sort S^1 in DFS lexicographic order;
6: S ← S^1;
7: for each edge e ∈ S^1 do
8:     initialize s with e, set s.GS = {g|∀g ∈ D, e ∈ E(g)};
9:     Subgraph_Mining(D, S, s);
10:    D ← D − e;
11:    if |D| < min_sup then
12:        break;
13:    end if
14: end for
```

In a first step, infrequent single nodes and edges are removed from the search space, as there can be no longer patterns with infrequent substructures

in them. The frequent one-edge subgraphs are stored in $\mathcal{S}^1$ and will be used as the seeds from which longer patterns are grown by calling algorithm 6 on all such one-edge patterns. Along the way, the graph set $\mathcal{D}$ is consecutively shrunk during each iteration, as previously searched patterns cannot turn up again later on. After finding all one-edge patterns and all their decedents, the algorithm terminates. For a definition of Enumerate(), see [25].

Algorithm 6 Subgraph_Mining($\mathcal{D}, \mathcal{S}, s$)[25]

```
1: if s ≠ min(s) then
2:     return;
3: end if
4: S ← S ∪ {s}
5: generate all s' potential children with one edge growth;
6: Enumerate(s);
7: for each c, c is s' child do
8:     if support(c) ≥ min_sup then
9:         s ← c;
10:        Subgraph_Mining(D, S, s);
11:    end if
12: end for
```

4.6 Support Vector Machine

An SVM[26] is a supervised machine learning model that can classify a data set into distinct groups by constructing a hyperplane between their feature vectors that maximizes the distance between the nearest data points and the hyperplane for any class (the functional margin). Figure 4.7 illustrates this for a simple, two dimensional example with two intuitively distinct classes. The samples on the margin are called the support vectors. The SVM module from the Python package scikit-learn[27] was used.

The original problem formulation for support vector classification is as follows:[29]

Let $\boldsymbol{x}_i \in \mathbb{R}^p$ with $i = 1, ..., n$ and let $\boldsymbol{y} \in \mathbb{R}^l$ be an indicator vector, such that $\boldsymbol{y}_i \in \{1, -1\}$. Then SVC solves the following primal optimization problem:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

subject to

$$y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i, \zeta_i \geq 0, i = 1, ..., n$$

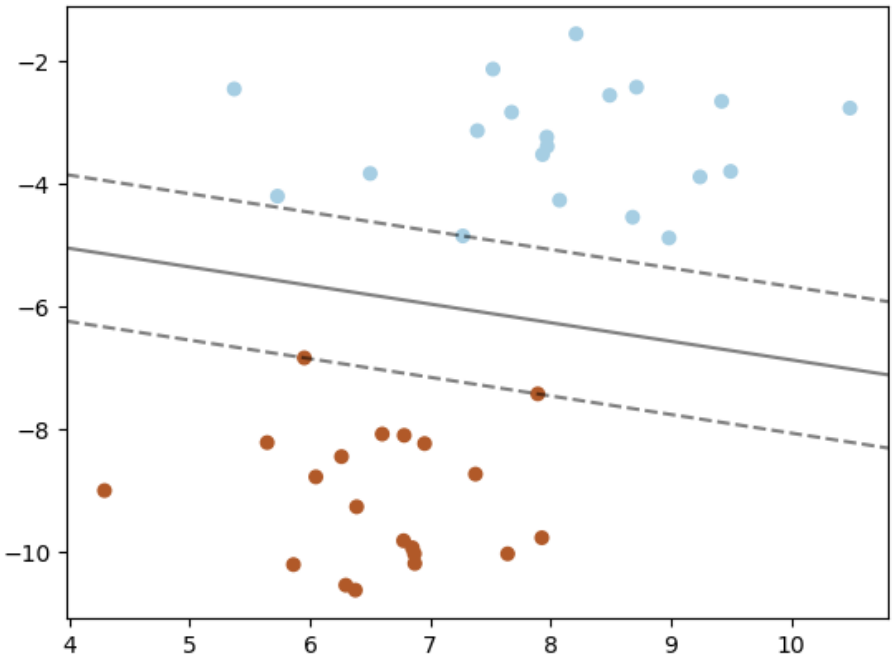

Figure 4.7: Functional margin between two classes of data points[28]

with its dual being

$$\min_{\alpha} \frac{1}{2}\alpha^T Q\alpha - e^T\alpha$$

subject to

$$y^T\alpha = 0, 0 \leq \alpha_i \leq C, i = 1, ..., n$$

where $\phi(\boldsymbol{x}_i)$ maps $\boldsymbol{x}_i$ into a higher-dimensional space, e is the vector of all ones, Q is an $x \times x$ positive semidefinite matrix with $Q_{ij} = y_i y_j K(x_i, x_j)$, with $K(x_i, x_j) = \phi(x_i)^T\phi(x_j)$ being the kernel, and $C > 0$ is the regularization parameter (upper bound). The decision function is given by:

$$\operatorname{sgn}\left(\sum_{i=1}^{n} y_i\alpha_i K(x_i, x) + \rho\right).$$

4.7 OEE anomaly detection

Site performance is measured through OEE (Overall Equipment Effectiveness) scoring. OEE calculation yields a scoring between 0 and 1 according to the following formula:

$$OEE = \frac{POK \cdot CT}{OT},$$

where POK is the number of **p**arts that came out of quality control **OK**, CT is the **c**ycle **t**ime in seconds per part and OT is the **o**perational **t**ime of the assembly line in seconds. All values are reset during shift changes, resulting in a short period of 0% OEE before the first part of a new shift is produced. As an example, if the line ran for 3600 seconds, needed 10 seconds to produce a part and produced 300, the resulting OEE score would be

$$OEE = \frac{300 \cdot 10}{3600} \approx .83.$$

This result would indicate an assembly line running with about 83% effectiveness. It should've produced 60 parts more in the given time, and, thus, was held up for some reason or another about 17% of the time.

OEE scores were available as a time series for the same time frame as the factory data set. As OEE scores were calculated every second, the resulting data set was considerably larger than the error logs, consisting of more than 800,000 rows of 26 columns, most of which weren't used. Anomaly detection consisted mainly of identifying more or less sudden drops in OEE scoring, indicating times when no parts were produced. The first few anomaly detection systems proved very capable of detecting shift changes and not much else, while later iterations did indeed pick up on most of the intuitively obvious drops.

Algorithm 7 provides an efficient anomaly detection algorithm with $O(n)$ complexity, with S being a set of slope indicators, $\tilde{S}$ the mean slope indication and c a manually set parameter for how many standard deviations from the mean slope a anomaly should be assumed.

Algorithm 7 OEEDetectAnomalies(OEE_data)

$S \leftarrow \emptyset$;
$R \leftarrow \emptyset$;
for each 5 minute slice $\boldsymbol{s}$; **do**
    if $\min_{OEE} \boldsymbol{s} \neq 0$; **then**
        $x \leftarrow \frac{\max_t \boldsymbol{s}}{\min_t \boldsymbol{s}} - 1$;
        $S \leftarrow S \cup x$;
    else
        $S \leftarrow S \cup 0$;
    end if
end for
$l \leftarrow \tilde{S} - c \cdot SD(S)$;
for each $\hat{\boldsymbol{s}} = (x, y, z)$ in $S, z < l$; **do**
    $R \leftarrow R \cup \hat{\boldsymbol{s}}$;
end for
return R;

Chapter 5

Experiments and performance study

5.1 Test setup

All tests were performed on a 2015 Lenovo Thinkpad T450s, with 12GB of RAM and an Intel Core i7-5600U clocked at 2.6 GHz, running a NixOS 17.09 with Python 3.6.4 built with GCC 6.4.0.

5.2 Results for a synthetic data set

The synthetic data set was generated by first simulating a random walk of OEE values from 11 am until about 9 pm, mimicking about one shift (figure 5.1). Later, equivalent error logs were created, with some messages more likely to turn up at times when the generated random walk resulted in an OEE drop as recognized by the OEE anomalies detection.

These generated data look similar to the real facility set, with various drops and and ascents. Resulting patterns like figure 5.2 also look very similar to real patterns like figure 4.3 above.

The synthetic error log consisted of 1000 rows of errors, the OEE set of 35,919 rows (roughly 9 hours of second by second logs). Run times were very manageable for higher min_sup values, but soon reached exponential (and thus, unsustainable) growth for min_sup values much lower than .2 (cf. table 5.1).

5.2.1 Evaluation with OEE data set

OEE anomalies were split into a training data set and an validation data set, with 80% of the data being used for training and the remaining 20% used

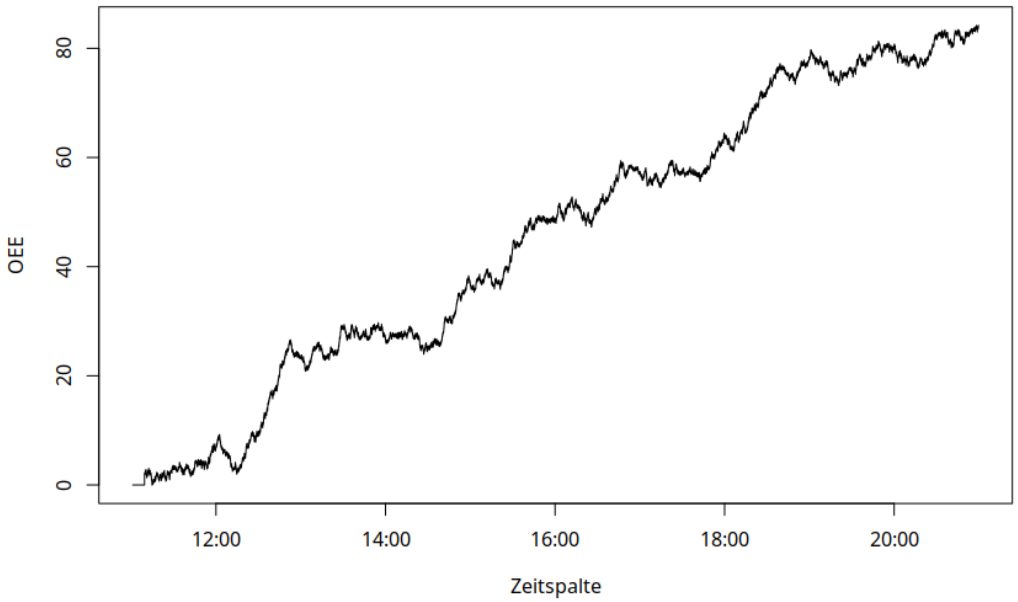

Figure 5.1: Synthetic OEE values

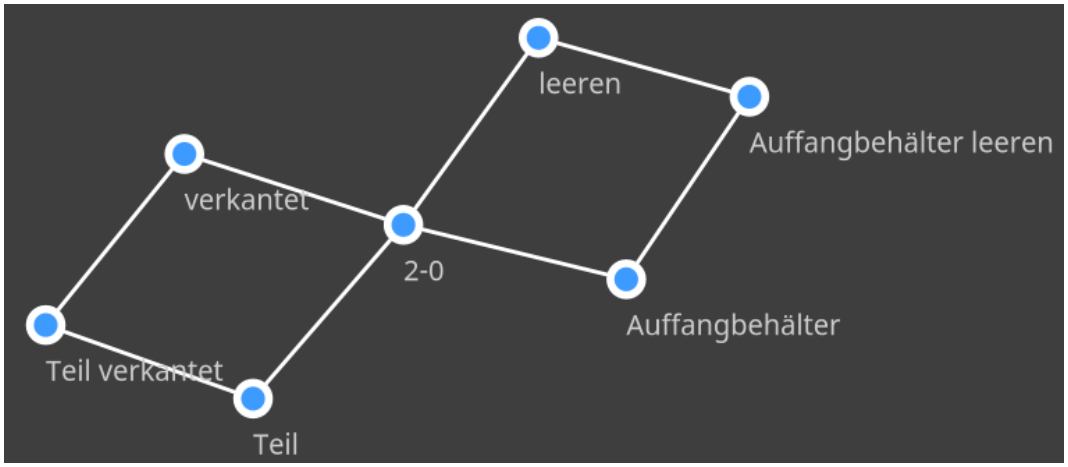

Figure 5.2: 8-edge pattern from the synthetic data set, min_sup = .4

Table 5.1: Run times and patterns found (synthetic data set)

data set	t	patterns
import errors and graph generation	1s	
import and anomalies detection on OEE	8s	
gSpan (min_sup = .7)	2s	40
gSpan (min_sup = .6)	8s	106
gSpan (min_sup = .5)	19s	241
gSpan (min_sup = .4)	74s	1056
SVM training and validation (min_sup = .7)	4s	
SVM training and validation (min_sup = .6)	8s	
SVM training and validation (min_sup = .5)	35s	
SVM training and validation (min_sup = .4)	13m 14s	

for validation. For the min_sup = .5 run, experimental mean slope $\tilde{S}$ (cf. algorithm 7) was .5 with a standard deviation of .21 and a c-value of .1. The validation data set consisted of 49 time windows, 33 of which were deemed as a noticeable drop by the OEE evaluation algorithm. Of these 33, the SVM correctly identified 28 as drops, for a sensitivity score of 85%. Of the remaining 19 non-drops, 5 were falsely identified as positives, for a specificity score of 74%.

For the min_sup = .4 run, experimental mean slope $\tilde{S}$ was .5 with a standard deviation of .21 and a c-value of .1. The validation data set consisted of 49 time windows, 33 of which were deemed as a noticeable drop by the OEE evaluation algorithm. Of these 33, the SVM correctly identified 28 as drops, for a sensitivity score of 84.85%. Of the remaining 19 non-drops, 5 were falsely identified as positives, for a specificity score of 73.68%.

5.3 Results for facility data set

The facility data set consisted of 57,171 rows of error logs and 802,800 rows of OEE evaluation data. Results with min_sup values of less than .5 could not be achieved. The algorithm consistently used up so much memory that it was OOM killed by the operating system after about a day of run time.

5.3.1 Evaluation with OEE data set

OEE anomalies were split as above. Experimental mean slope $\tilde{S}$ (cf. algorithm 7) was 1.1 with a standard deviation of .16 and a c-value of .1. The validation data set consisted of 486 time windows, 64 of which were deemed as a noticeable drop by the OEE evaluation algorithm. Of these 64, the SVM trained on

Table 5.2: Run times and patterns found (facility data set)

data set	t	patterns
import errors and graph generation	50s	
import and anomalies detection on OEE	2m 27s	
gSpan (min_sup = .9)	2m 20s	12
gSpan (min_sup = .7)	6h 27m 12s	846
gSpan (min_sup = .5)	*OOM killed*	–
SVM training and validation (min_sup = .7)	27s	

patterns with a min_sup of .7 correctly identified 60 as drops, for a sensitivity score of 93.75%. Of the remaining 422 non-drops, 18 were identified as false positives, for a specificity score of 95.73%.

Chapter 6

Summary and Discussion

To conclude this bachelor's thesis, the following will summarize my findings with the acknowledgment that, although the essential research in this work was of my own design and execution, a project such as this is virtually impossible without guidance by an advisor and support by friends and family.

With this work, I've introduced a method and provided a Python program to mine error log data for useful patterns, using a graph representation to take advantage of structural information and incorporate outside expert knowledge. I touched upon the most important concepts on which my model assumptions rest and expounded on some approaches that did not yield usable results.

The proposed algorithm has been shown to produce patterns with adequate experimental time complexity, with synthetic data and proprietary Siemens facility data. The found patterns, to a first approximation, seem to provide real informational value and seem able to predict facility downtimes, as measured by a drop in OEE, all to a reasonable degree. A possible next step would be to show the patterns and the thoughts that went into the OEE anomaly detection to an expert with domain knowledge and then refine the proposed approaches through a few more iterations.

The proposed approach has been shown to be somewhat fragile, in that at least some implementation details of *gSpan*, the overall data structure used in this work, and maybe even the included libraries should be reevaluated at a later time, to hammer out possible errors and improve on the interaction between parts.

Further improvements to the algorithm, especially to improve on average-case time and memory performance, and allow it to directly process data streams instead of stale data would be much appreciated, but are sadly out of scope for this bachelor's thesis.

The results that *could* be reached, however, point in a promising direction. The overall approach – leveraging a graph-based mining algorithm against a

time series of event logs – seems to have merit, not least of all because the resulting patterns can be visualized in a way that immediately makes a lot of sense to both the casual observer as well as the expert with intimate domain knowledge.

This remains true even if event logs don't immediately spring to mind as being structurally similar to networks and as such means this approach needs further research and should at least be tried again with similarly non-obvious graph data in the future.

Acknowledgments

I want to thank first and foremost my advisor, Martin Ringsquandl, as without his ever intelligent and on-point criticisms and ideas this bachelor's thesis wouldn't have been possible. Second, Prof. Dr. Krger, for allowing this thesis as an external bachelor's thesis at the Munich Siemens AG headquarters. Third, Siemens Corporate Technology, and all the intelligent and lovely folks at the Research, Development and Automation/Business Analytics and Monitoring unit, who provided valuable input not only during lunch hours. I also wish to thank all my friends and family, who constantly bugged me about my progress especially during the later stages, especially Irina, Christina and my mother. Last, but certainly not least, I also want to thank my cat Tigris, who bugged me as well while I was writing, although he was mostly out for food.

Furthermore, I am very thankful to live in a time with tools such as CytoScape[30],TeXStudio[31], TeXLive[32], IntelliJ PyCharm, and PaperPile for making the development of my analytics software and the later write-up much easier and more efficient.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Thomas A Peters. *The history and development of transaction log analysis. Library Hi Tech* **11** no. 2, (1993) 41–66.

[2] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: yesterday, today, and tomorrow.* `arXiv:1606.04036 [cs.SE]`.

[3] "Apache Hadoop - Open source software for reliable, scalable, distributed computing." `http://hadoop.apache.org/`. Accessed: 2018-2-2.

[4] "Elasticsearch - Open Source Search & Analytics." `https://www.elastic.co/`. Accessed: 2018-2-2.

[5] "Grafana - The open platform for analytics and monitoring." `https://grafana.com/`. Accessed: 2018-2-2.

[6] "Graylog - Open Source Log Management." `https://www.graylog.org/`. Accessed: 2018-2-2.

[7] Martin Ringsquandl, Steffen Lamparter, Ingo Thon, Raffaello Lepratti, and Peer Kroger. *Knowledge Graph Constraints for Multi-label Graph Classification.* In: 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW), pp. 121–127. IEEE, 2016.

[8] Takashi Washio and Hiroshi Motoda. *State of the Art of Graph-based Data Mining. SIGKDD Explor. Newsl.* **5** no. 1, (July, 2003) 59–68.

[9] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. *Frequent pattern mining: current status and future directions.* In: Proceedings of the fifteenth ACM SIGKDD international conference on Knowledge discovery and data mining, vol. 15, pp. 55–86. Kluwer Academic Publishers-Plenum Publishers, Aug., 2007.

[10] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. *Mining Association Rules Between Sets of Items in Large Databases. SIGMOD Rec.* **22** no. 2, (June, 1993) 207–216.

[11] Jian Pei, Jiawei Han, Behzad Mortazavi-asl, and Hua Zhu. *Mining Access Patterns Efficiently from Web Logs.* In: Knowledge Discovery and Data Mining. Current Issues and New Applications, pp. 396–407. Springer, Berlin, Heidelberg, Apr., 2000.

[12] Xifeng Yan and Jiawei Han. *gSpan: graph-based substructure pattern mining.* In: 2002 IEEE International Conference on Data Mining, 2002. Proceedings., pp. 721–724. IEEE Comput. Soc, 2002.

[13] Gueorgi Kossinets and Duncan J Watts. *Empirical analysis of an evolving social network. Science* **311** no. 5757, (Jan., 2006) 88–90.

[14] Rakesh Agrawal, Ramakrishnan Srikant, and Others. *Fast algorithms for mining association rules.* In: Proc. 20th int. conf. very large data bases, VLDB, vol. 1215, pp. 487–499. 1994.

[15] Ramakrishnan Srikant and Rakesh Agrawal. *Mining Quantitative Association Rules in Large Relational Tables.* In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD '96, pp. 1–12. ACM, New York, NY, USA, 1996.

[16] Mark Newman. , *Networks: An Introduction.* Oxford University Press, Mar., 2010.

[17] Jeffrey Travers and Stanley Milgram. *The small world problem. Phychology Today* **1** no. 1, (1967) 61–67.

[18] Jia-Wei Han, Jian Pei, and Xi-Feng Yan. *From sequential pattern mining to structured pattern mining: A pattern-growth approach. J. Comput. Sci. & Technol.* **19** no. 3, (May, 2004) 257–279.

[19] Mohammed J Zaki. *SPADE: An Efficient Algorithm for Mining Frequent Sequences. Mach. Learn.* **42** no. 1-2, (Jan., 2001) 31–60.

[20] Jiawei Han, Guozhu Dong, and Yiwen Yin. *Efficient mining of partial periodic patterns in time series database.* In: Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337), pp. 106–115. Mar., 1999.

[21] Scott Fortin. *The graph isomorphism problem.* Tech. Rep. 96-20, University of Alberta, 1996.

[22] Stephen A Cook. *The complexity of theorem-proving procedures.* In: Proceedings of the third annual ACM symposium on Theory of computing, pp. 151–158. ACM, May, 1971.

[23] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. *GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph.* In: Proceedings of the VLDB Endowment. Mar., 2014.

[24] Mohamed Moussaoui, Montaceur Zaghdoud, and Jalel Akaichi. *POSGRAMI: Possibilistic Frequent Subgraph Mining in a Single Large Graph.* In: Information Processing and Management of Uncertainty in Knowledge-Based Systems, Joao Paulo Carvalho, Marie-Jeanne Lesot, Uzay Kaymak, Susana Vieira, Bernadette Bouchon-Meunier, and Ronald R Yager, eds., vol. 610 of *Communications in Computer and Information Science*, pp. 549–561. Springer International Publishing, Cham, 2016.

[25] Xifeng Yan and Jiawei Han. *gSpan: Graph-Based Substructure Pattern Mining, Expanded Version.* Tech. Rep. UIUCDCS-R-2002-2296, UIUC Technical Report, 2002.

[26] Corinna Cortes and Vladimir Vapnik. *Support-vector networks. Mach. Learn.* **20** no. 3, (Sept., 1995) 273–297.

[27] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. *Scikit-learn: Machine Learning in Python. J. Mach. Learn. Res.* **12** no. Oct, (2011) 2825–2830.

[28] "1.4. Support Vector Machines — scikit-learn 0.19.1 documentation." `http://scikit-learn.org/stable/modules/svm.html`. Accessed: 2018-2-4.

[29] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: A Library for Support Vector Machines. ACM Trans. Intell. Syst. Technol.* **2** no. 3, (May, 2011) 27:1–27:27.

[30] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. *Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome Res.* **13** no. 11, (Nov., 2003) 2498–2504.

[31] Benito van der Zander. , "TeXstudio." `http://www.texstudio.org/`. Accessed: 2018-2-2.

[32] Sebastian Rahtz, Akira Kakuto, Karl Berry, Luigi Scarso, Mojka Miklavec, Norbert Preining, Reinhard Kotucha, Siep Kroonenberg, and Staszek Wawrykiewicz. , "TeX Live - TeX Users Group." `https://www.tug.org/texlive/`. Accessed: 2018-2-2.