Bachelor Thesis

Analysis and Simplification of Business Graphs

Analyse und Vereinfachung von Unternehmensgraphen

Milan Gruner

milangruner@gmail.com

21.07.17

Information Systems Group

Supervisors
Prof. Dr. Felix Naumann
Michael Loster
Toni Grütze

Abstract

Constructing a graph made up of millions of businesses may be hard, but actually making sense of it is a lot harder. With huge amounts of data potentially being integrated into the data lake every day, automatic methods for finding interesting areas in the graph are needed. This paper discusses different approaches that can be taken to extract useful knowledge from such a graph.

First, the bachelor project "Ingestion" and the technologies used for implementing the algorithms in this paper will be introduced. Then, the data structures that are the basis of all data contained in the data lake will be detailed, followed by a introduction to the used graph processing framework and a description of the algorithm used to construct a graph from the previously discussed data structures.

Afterwards, an algorithm to extract company group structures from the graph is discussed, as well as a method to find structural patterns. These work using connected component analysis and motif search and are provided as examples for how to extract specific features from the graph.

Following, an automated process for simplifying the graph by applying PageRank and thresholding algorithms and methods customizing the extracted companies and for increasing the degree of interconnectedness are introduced.
Finally, the discussed algorithms are evaluated, conclusions are drawn and an outlook on further possible improvements is provided.

Zusammenfassung

Einen Graph aus Millionen von Unternehmen zu konstruieren mag einige Probleme bereiten, aber tatsächlich Erkenntnisse daraus zu extrahieren ist um einiges schwieriger. Während potentiell riesige Datenmengen jeden Tag in die Datenbasis integriert werden, sind automatisierte Methoden um interessante Bereiche des Graphen vonnöten. Diese Arbeit diskutiert verschiedene Ansätze, um hilfreiches Wissen aus einem solchen Graphen zu extrahieren.

Zuerst wird das Bachelorprojekt "Ingestion" und die Technologien zur Implementierung der Algorithmen in dieser Arbeit vorgestellt. Danach werden die Datenstrukturen, die die Basis der Datenbasis darstellen, detailliert, gefolgt von einer Einführung in das benutzte Graphprozessierungs-Framework und einem Algorithmus zur Konstruktion einer Graph-Datenstruktur aus den bereits diskutierten Datenstrukturen.

Im folgenden wird auf einen Algorithmus zur Extraktion von Konzernstrukturen aus dem Graph eingegangen, sowie auf eine Methode, um strukturelle Muster zu finden. Diese funktionieren mithilfe von Zusammenhangskomponenten-Analyse und Motif-Suche und sind als Beispiele zur Extraktion auf Basis von strukturellen Eigenschaften dargelegt.

Nachfolgend wird ein automatisierter Prozess zur Graphvereinfachung durch die Anwendung von PageRank- und Schwellwertbildungsalgorithmen vorgestellt, sowie Methoden, um die Einfluss auf die Extraktion der Unternehmen und um die Verbundenheit des resultierenden Graphen zu erhöhen.
Abschließend werden die diskutierten Algorithmen evaluiert und es wird ein Ausblick auf zukünftig mögliche Verbesserungen gegeben.

Contents

1 The German Corporate Graph Project

1.1 Why a German Corporate Graph?[1]

When it comes to economic decisions, uncertainty is a critical issue. Following the rational choice theory approach, every market player is constantly trying to maximize his utility and minimize his effort.

Uncertainty can be described as a lack of information of how a market - or herein the full German economic system - is constituted and about the future behavior of the market players. Presuming that every market player is acting on a rational basis, all information regarding his situation, resources, plans, and relations makes the results of his decisions more predictable. In this manner, we can state: The more relevant information a market player gathers about other players in the market or economy, the better the foundation of his decisions is. The broad range of that kind of information can lead to a significant competitive advantage. So it should be in a rational player's interest to collect as much relevant information as possible.

In a connected economy, a lot of those uncertainties lie in the relations between corporations[2]. This became evident in the so called *Abgas-Skandal* or *Dieselgate* of the Volkswagen AG in 2015, wherein a lot of external suppliers spun out of control, from a financial perspective [1, 4]. This happened although most of the suppliers did not take part in the scandal itself. Since there are a lot of other examples like the *Lehmann Brothers bankruptcy* or any other economic shock event, we can state that relations are a significant factor in the economic evaluation of corporations and their financial risks.

Because there are millions of corporations in the German economy[3] and each corporation can potentially holding relations to hundreds or thousands of other corporations, collecting and to overseeing all those relations becomes a complicated matter.

The *German Corporate Graph Project* is one approach to solve this problem. The project's purpose is to extract business entities from multiple structured knowledge bases (e.g. Wikidata and DBpedia), merge them, enrich them with relations extracted from unstructured documents and finally display the graph so that it can be visually explored.

The project consists of a pipeline, which starts with the import and normalization of structured knowledge bases. The next step is the Deduplication, which is the detection

---

[1]This section was authored by Matthias Radscheit [6].

[2]We define as *corporation* any juristic entity that takes part in the German economy. This includes especially businesses but also other entities like public corporations.

[3] The *Federal Bureau of Statistics* notes 3,469,039 businesses in Germany in 2015 [2]. Following our definition of corporations, this number has to be seen as a lower bound for the total number of corporations in Germany.

and fusion of occurrences of the same entity over multiple knowledge bases. These entities form a graph, whose nodes are businesses and whose edges are the relations between them. This graph is then enriched during the Information Extraction. In this step relations between entities are extracted from unstructured documents using Named Entity Recognition, Entity Linking and Relation Extraction.

The results of all these steps can be viewed and curated in the so-called Curation Interface. This is a web-interface, which can be used to control the pipeline itself, view statistical data generated by other pipeline steps and to view and curate the entities and relations of the graph itself. The final graph can be visually explored by using the Corporate Landscape Explorer, which is also a web-interface.

1.2 One project - seven contributions

This thesis is published as a part of a bachelor's project in 2016/2017 at Hasso-Plattner-Institute in Potsdam, Germany. The project's objective was to build the *German Corporate Graph*, like described above, for Germany's corporate landscape. The project lasted ten months and was accompanied by Commerzbank AG, Germany. As part of the process, the project participants published several theses.

See here a list of all published theses within the project's context:

- Pabst explores *Efficient Blocking Strategies on Business Data* [8].

- Löper and Radscheit evaluate duplicate detection in their thesis *Evaluation of Duplicate Detection in the Domain of German Businesses* [6].

- Schneider's thesis is entitled *Evaluation of Business Relation Extraction Methods from Text* [10].

- Janetzki investigates *Feature Extraction for Business Entity Linking in Newspaper Articles* [5].

- Ehmüller explores the *Evaluation of Entity Linking Models on Business Data* [3].

- *Analysis and Simplification of Business Graphs* is the title of Gruner's thesis (this paper).

- Strelow investigates the *Distributed Business Relations in Apache Cassandra* [11].

1.3 The Ingestion and Curation projects

This paper describes the basic approaches that were taken for analyzing the graphs generated as a part of the Ingestion bachelor project at HPI Potsdam. It deals with the algorithms and approaches that were employed to extract information from automatically generated graphs in the ball park of tens of millions of nodes in a large and mostly unconnected and sparse graph. This graph contains businesses, places and persons and can be augmented freely by adding additional data sources (or manual data input) of your own. The data is visualized and controllable from the Curation interface, which was a project developed alongside the Ingestion pipeline. Curation generates graphs on-the-fly that contain the statistics data that was written as a side effect of the numerous Spark jobs in the Ingestion pipeline and gives a lot of insight into what's going on in the "data lake".

1.4 Technologies used in this paper

All algorithms in this paper were executed on the Apache Spark platform which controls the parallelization and data flow of the functional steps described in the code samples. The Scala programming language is used because Spark is written in it and because it allows for a more terse representation of the algorithms than the respective Java or Python implementations, shortening the code listings and making them easier to understand. It is also the language the rest of the Ingestion project is written in, resulting in a better interoperability and facilitating code exchange and reuse with it.

Most of the topics in this paper deal with graph-related topics, so the GraphFrames framework (in Scala) will be used to describe the algorithms that were employed, but it can be freely seen as functional pseudo code and adapted to other popular graph processing frameworks (such as GraphX, Giraph etc.).

Also one of the purposes of the Ingestion and Curation projects was to enable the execution of modern text mining, deduplication and parallel data analysis techniques to the german business landscape. This means that some of the examples will be in German (but they will be translated if needed for understanding the technique).

1.5 Usage Scenarios of Ingestion, Curation and GraphXplore

The process of classic risk analysis which is still majorly employed in today's banks is mostly a model-oriented process that relies on predictions and can be augmented by data-driven approaches. The techniques that Ingestion, Curation and this paper offer are universally applicable to analyze huge amounts of structured and unstructured data

in the form of WikiData, JSON, CSV or similar data as well as newspaper articles, email messages or similar data sources.

The results of the algorithms contained in these projects can either be viewed as a graph or a relational model using the conversion algorithms briefly discussed later. Because this project consists of automatic processes that control the flow, categorization and extraction of all the data contained in the data lake, it may be effectively deployed in a bank or similar institution as a data collection, curation and analysis software stack.

It is meant to be heavily extensible and allows the simple integration of new data sources as well as algorithms that allow for further data processing and enrichment. The pipeline itself is very modular and is composed of many stand-alone Spark jobs that transform the data in the Cassandra database.

2 Data Structures for Business Entities

2.1 The subject data structure

The central data structure that exists inside the data lake is called *subject*. It contains all the final businesses, industry sectors, countries or persons that are the result of the computations in the Ingestion project.

This is what it looks like in Scala:

Listing 1: Subject

```scala
case class Subject(
  var id: UUID = UUID.randomUUID(),
  var master: UUID,
  var datasource: String,
  var name: Option[String] = None,
  var aliases: List[String] = Nil,
  var category: Option[String] = None,
  var properties: Map[String, List[String]] = Map(),
  var relations: Map[UUID, Map[String, String]] = Map(),

  var master_history: List[Version] = Nil,
  var name_history: List[Version] = Nil,
  var aliases_history: List[Version] = Nil,
  var category_history: List[Version] = Nil,
  var properties_history: Map[String, List[Version]] = Map(),
  var relations_history: Map[UUID, Map[String, List[Version]]] = Map()
)
```

The first three attributes (*id*, *master*, *datasource*) constitute the primary key in the Cassandra databases. The partitioning key that is used to decide on which Cassandra instance to store a given *subject* entry is the *master* ID, which makes sure that a master node and all its data source slave nodes are stored on the same machine. *id* and *datasource* are used as clustering keys, which control the sorting inside the data partitions produced by database queries. They are also easily queryable when specific IDs or data source entries for a given master ID are needed.

When loading Cassandra data into a Spark job, simple fields (e.g. Strings) need to be wrapped in an Option in case that the field contains no value (otherwise there would be an error), which isn't necessary for the fields containing Maps or Lists, which would just contain an empty data structure in that case.

The *properties* field contains all the extracted data from the various data source extraction and normalization steps in the Ingestion pipeline. It is structured in a way to make storing arbitrary data possible, by allowing to store a List of Strings for each property key. This way, a list of eMail addresses can be stored as well as a single postal code and they can be treated the same way.

The *relations* field is structured in a two-dimensional map. The first level is addressed by the target *subject* ID, while the second level's keys are the specific relation type. The value that is returned when the field is queried by the target ID and relation type is the relation attribute, which is mostly used to store a confidence measure (a value from 0 to 1) or the count of the relation type between the current *subject* and the target.

2.2 A versioning scheme that stands the test of time

Versioning is needed for the automatic system that includes various pipelines that write to lots of different tables in Cassandra during their runtime, but at the end it is expected that all data will flow back into the central 'data lake' that is the *subject* table, described earlier in this chapter. This means that fully automated processes may and will write millions of data entries to the data lake at an hourly basis (or even more frequent), so if any breaking changes are introduced by these, they need to be rerolled quickly. This means that each version isn't saved as a diff like in version control systems (e.g. git), but rather every single version has a copy of all the attributes, a timestamp and the name and version of the program that modified it. As our whole pipeline is made up of individual small Spark jobs that write mostly into their own Cassandra tables, the error in the data can be quickly located and traced to the corresponding source code.

Because our data domain included events, attributes and relations that had temporal validity parameters attached to them, we had to realize a versioning scheme that was compatible with the column-family oriented storage of Cassandra. As everything is

distributed across the cluster here, difference-based versioning would be hard to keep up, so our team decided on a deeply structured deep copy backup system for our versions.

The version data structures are stores for each attribute and each individual key of the *subject* entries, which makes restoring individual columns or single data entries efficient and easily parallelizable.

The version data structure that was used is structured like this:

Listing 2: Version

```scala
1    case class Version(
2      version: UUID = UUIDs.timeBased(),
3      var program: String,
4      var value: List[String] = Nil,
5      var validity: Map[String, String] = Map(),
6      var datasources: List[String] = Nil,
7      var timestamp: Date = new Date()
8    )
```

It is stored inside the collections of the *history* fields of the *subject* data structure and contains useful meta information for restoring old versions, modeling temporal validity (attributes only being valid for a certain timespan) or filtering out data from certain data sources. They also contain the Spark program's name that created the relevant changes and the time they were made.

Additionally, the *version* table contains a list of all changes to the data lake, making it traceable which programs were ran at what time, which table they wrote to and what data sources they processed. Using this, the Curation interface can display the annotated version list, and run Spark jobs that restore the whole data lake to a previous version or compute the changes that a version made in the *subject* table.

More detail on the design of the *subject* data structure and the versioning scheme is discussed in Strelow [11].

3 Using Spark and GraphFrames for Graph Analysis

This chapter will briefly discuss the reasons for using the Apache Spark[4] and DataStax GraphFrames[5] frameworks alongside the Apache Cassandra database[6]. Then the transformations that are necessary to extract GraphFrames-compatible data from the subject data structure will be detailed. The following chapters will build on the techniques described in this one, so that only the relevant graph processing sections need to be shown later on.

3.1 Why Spark, Cassandra and GraphFrames?

The subject table contains millions of entries. In order to efficiently extract a graph and calculate the results of complex graph algorithms, a parallel processing framework like Apache Spark is a really important measure to take into consideration when runtimes of multiple hours (or even days) aren't desired, which normally would be the case for data of this magnitude. Cassandra is a database that is up to the challenge of delivering the input for and storing the results of these computations without much complicated serialization, like storing them in CSV or binary files (like e.g. the Parquet format commonly used for storing Spark's data frames) would involve. It is also well integrated into the Spark API and can take care of complex nested data structures like e.g. the properties and relations attributes of the *subject* data structure discussed, which was discussed in chapter 2.1.

GraphFrames is a really useful tool for interacting with huge amounts of graph data. The advantages in comparison to the GraphX framework (included in Spark) are that arbitrary data types can be used for IDs, instead of just *long* values, and that the whole data frames are always usable, instead of just a single vertex attributes that are overwritten by algorithms executed on the graph. This is necessary, when the algorithms need to evaluate certain attributes like the size, industry sector or geographic location of the companies. GraphFrames also comes with handy implementations of common graph algorithms like breadth-first search, PageRank and motif search that will be heavily used in the following chapters.

3.2 Constructing a GraphFrame from subject data

The core class of the GraphFrame API is the class GraphFrame itself, which is basically a combination of a vertex and relation data frame as well as a lot of operations which can

---

[4]`https://spark.apache.org`
[5]`https://graphframes.github.io`
[6]`https://cassandra.apache.org`

be performed on them. It is a useful tool for easily handling the graph data structure and interacting with it in a functional manner whilst still being able to interact with the relational API of Spark's data frames. The case class entries that are queried from Cassandra are transformed into vertex and relation data using the Spark RDD API. The results are then translated into data frames, which GraphFrames expects when creating a new graph instance.

The *extractGraph* function from *GraphExtractor* implements this process:

Listing 3: extractGraph in *GraphExtractor*

```
1    val subjectVertices = subjects
2      .map(subject => (subject.id.toString, subject.name))
3      .toDF("id", "name")
4    val subjectRelations = subjects.flatMap(subject => {
5      subject.masterRelations.flatMap { case (id, relTypes) =>
6        relTypes.map { case (relType, value) =>
7          (subject.id.toString, id.toString, relType, value)
8        }
9      }
10   }).toDF("src", "dst", "relationship", "value")
11
12   GraphFrame(subjectVertices, subjectRelations)
```

If entries from the *properties* map are needed (e.g. for calculating the custom weights in chapter 5.2), they need to be manually extracted and added to the column names like shown in listing 4. They can then be used as a vertex attribute.

Listing 4: Custom weight extraction

```
1    val subjectVertices = subjects
2      .map(subject => {
3        val weight = subject.properties("graph_weight")
4          .headOption.getOrElse("1.0").toDouble
5        (subject.id.toString, subject.name, weight)
6      }).toDF("id", "name", "weight")
```

The three *flatMap* and *map* statements flatten the nested structure of the *relations* attribute, transforming it into multiple tuples of source and target node, relation type, and relation value. It then names them in the scheme that the *GraphFrame* constructor expects.

The result is a GraphFrame instance containing the company and relation data of all *subject* entries in the database that will be used to execute the algorithms in the following chapters.

4 Approaches for Subgraph Extraction

This chapter deals with various graph processing algorithms developed using the Graph-Frames framework that extract useful subsets of the full graph in the data lake. These are some examples of simple graph processing that lay the cornerstone for the remaining graph evaluation techniques discussed in later chapters.

4.1 Finding company groups in the business graph

As a first experiment with real-world applications, company group extraction was a good starting algorithm that could be implemented using the connected component tagging implementation in the GraphFrames API[7]. Here, all businesses that are connected via partial or full ownership relations are considered as a company group.

Before the ConnectedComponets algorithm is run, all ownership-related company relations are extracted from the *subject* graph. This is done by whitelisting all relations (in *ownershipRelations*) that concern this particular subgraph, e.g. "owns" and "ownedBy". Then, after executing the search, all entries are grouped by the ID of the associated component and their names and UUIDs are extracted and saved to Cassandra as an entry of the *graphs* table. The following code snippet describes how this process is expressed using GraphFrames:

Listing 5: Company Group Extraction

```scala
1  val ownershipEdges = graph.edges.filter((edge) =>
2    ownershipRelations.contains(edge.getAs[String]("relationship")))
3  val ownershipGraph = GraphFrame(graph.vertices, ownershipEdges)
4
5  ownershipGraph.connectedComponents.run()
6    .select("id", "component", "name")
7    .rdd
8    .map(row => (
9      row.getAs[Long]("component"),
10     UUID.fromString(row.getAs[String]("id")),
11     row.getAs[String]("name")))
12   .groupBy(_._1) // component
13   .map(_._2.map(t => (t._2, t._3)).toList) // extract id and name
14   .filter(_.length >= minComponentSize)
15   .map(idNameTuples => ResultGraph(
16     outputGraphType,
17     idNameTuples.map(_._1),
18     idNameTuples.filter(_._2 != null).map(_._2)))
```

---

[7]`https://graphframes.github.io/api/scala/index.html#org.graphframes.lib.ConnectedComponents`

As a result we get a list of businesses like the following:

- Oracle
- Oracle America
- Sun Microsystems
- Hyperion Solutions
- Micros Systems
- Oracle Financial Services Software
- PeopleSoft
- Acme Packet
- Art Technology Group

The corresponding IDs are also stored alongside the names and the graph type (here: "CompanyGroup"). This makes it easier to analyze the results without joining all subject names and IDs to this dataset. It also facilitates displaying a graph list in the Curation interface, because only the content of one table needs to be fetched for it.

4.2 Using motif finding to find structural patterns

Motif finding is an approach to search for specific relation structures between vertices in the graph. On a GraphFrame, it is expressed using strings in a domain-specific format, not unlike the Cypher query language that is used for querying the Neo4j database, and executed using the $find$ method of the GraphFrame[8]. Vertices are selected using arbitrary names inside of parentheses, while edges are expressed using square brackets, which have to be preceded by a dash and followed by an arrow. Vertices and edges don't have to be named if they aren't needed for further processing. A single definition consists of an edge between two nodes, and these constructions are separated from each other by semicolons.

The resulting data frame that is returned by the $find$ method contains a column for each of the assigned names, which contain *StructType* instances that have the original data of the corresponding vertices or relations inside of them (in the same format as the data frames in chapter 3.2). As an example, the following expression selects chains of three vertices x, y, z:
```
"(x)-[e1]->(y); (y)->[e2]->(z)"
```

Because the queries are expressed using strings, they can be automatically constructed. This is necessary, because each query can only search for a constant number of vertices and relations, but most relevant graph structures are indifferent to their count.
Listing 6 details how to automatically construct queries for stars of varying sizes:

---

[8]`https://graphframes.github.io/api/scala/index.html#org.graphframes.GraphFrame`

Listing 6: Star Motif Query Creation

```
1  ('a' to 'z')
2    .take(starSize)
3    .map(char => "("+char+")-[]->(center)")
4    .mkString("; ")
```

This approach is very flexible and can be used to generate many types of patterns, like chains or cycles of varying lengths. It is executed (with the associated $find$ call) inside a counting loop (incrementing $starSize$). After this motif search iteration has been run, custom conditions can be evaluated on the resulting data frame to limit the number of returned patterns. For instance, only patterns containing companies with certain attribute values, only certain relation types or at least one relation of a specific type can be filtered, resulting in a more fine-grained control over the extracted patterns. In the end, the final vertices and relations are stored in the $graphs$ table in the same fashion as the company groups in the previous section.

5 Graph Simplification using PageRank and Thresholding

The PageRank algorithm by Page et al. [9] was originally intended to compute the likelihood that a user might arrive at a web page in a network of interconnected documents, to objectively quantify their "importance" in this network. It can, however, be adapted to most graph structures and also finds an application when dealing with business graphs, as the navigation between businesses over their outgoing relations is quite alike to the navigation between web pages over their hyper links. Therefore, the vertex output values ($pagerank$) of the algorithm can be interpreted as the probability that a user might arrive at a certain business when freely navigating the graph, starting at any connected node. The relation output values ($weight$) can be seen as the likelihood that the user travels along this relation to the next company.

5.1 Using PageRank to find relevant companies

The PageRank implementation found in the GraphFrames library[9] is used for parallel execution of this algorithm. There are two different possibilities to call it on a given graph: it can either be executed for a given number of iterations, or it can be run until convergence is reached within a certain tolerance interval to keep the algorithm from running endlessly. The former is a lot faster, because it does not repeat until the iterations produce a small enough change in each node, but the number of iterations has to be manually adapted to the graph size and its degree of interconnectedness, otherwise

---

[9]`https://graphframes.github.io/api/scala/index.html#org.graphframes.lib.PageRank`

the results differ too much from the values when convergence is reached, possibly making for a high chance of errors. Here, a trade-off between execution speed and resulting graph quality has to be made. By running until convergence, the only parameter that has to be controlled is the tolerance threshold, which is independent of the input graph because it only depends on the accuracy that is desired, as the algorithm continues executing until all *pagerank* value changes are under this threshold.

After the PageRank algorithm is executed, the results still have to be interpreted, meaning that the nodes have to be separated into "important" and "unimportant" groups. This can be done using thresholding techniques which can be parallelized over the connected components of the business graph.

First, the individual *pagerank* values assigned to every node have to be normalized over the whole connected component to limit them to the interval $[0, 1]$, so that the following tresholding step can work with a constant maximum value. This is done by finding the maximum of this value in the component and dividing every node's value by it. Then, a threshold value has to be picked for the component. This can either be done using an absolute threshold that has to be determined for the whole graph, or using an adaptive thresholding technique like Otsu's method [7], which is normally used for greyscale image thresholding, but is equally applicable here as it works on all one-dimensional histograms. It is particularly useful in this case because it removes the reliance on the degree of interconnectedness in the components and their size, and finds a good threshold for all connected component sizes that separates two value classes ("important" and "unimportant") from each other.

It works by creating a histogram of the absolute frequency of each *pagerank* value present in the component, which are grouped into bins. Then, the values are iteratively separated into two classes while trying to minimize the variance inside these classes.
In the end, a threshold is picked that can be used to decide whether or not to include a node in the important group based on its *pagerank* value.

The same can be done for the relations in the current connected component. By only showing the edges that the user is likely to use, the graph can be greatly simplified, making use of the weights the PageRank algorithm assigns to the relations present in the graph. This approach is useful when the input graph contains a great number of edges that are not desired in the output, but can, depending on the chosen threshold, lead to a very segmented and unconnected result graph.

5.2 Adding custom weights to influence company extraction

In many cases, a fully automated extraction process can be undesirable because it leads to a loss of control over which companies are chosen to be part of the output. To introduce custom weights into the process, more data can be joined and multiplied into the results of the PageRank algorithm before applying the thresholding process.

For all companies that are to be modified, its UUID (*id*) and a custom floating point value (*weight*) need to be present in the new data frame (*weightFrame*). This data can be acquired by manually adding attributes (e.g. *graph_weight*) into the concerned companies' properties in the Curation interface, or by logging and counting every manual change in the Curation interface and using these values as an offset for the PageRank values (these would have to be added instead of multiplied in then). The weight values can also be calculated by a function that measures the existence of relevant attributes in the *properties* structure of the companies, or by using another dataset like a blacklist of companies that should never be included.

Then, the vertex data frame containing the PageRank output and the new data frame containing the custom weights can be joined and multiplied like this:

Listing 7: Custom Weight Joining

```
1  val weightedVertices = rankGraph.vertices
2    .join(weightFrame, List("id"))
3    .select("id",
4      (rankGraph.vertices.pagerank * weightFrame.weight).alias("pagerank"))
```

After applying this step and the tresholding pass, companies that had a weight of 0.0 assigned won't show up in the resulting graph, while companies with a weight greater than 1.0 are more likely to be part of it.

5.3 Finding paths between relevant companies

The companies extracted in the previous sections are a small subset of the actual graph. This means that most of them won't share any relations, resulting in a mostly unconnected graph. This of course isn't desirable, so the graph needs to be enriched by additional edges.

These edges however won't exist in the original graph, so they have to be generated by searching for paths between the extracted nodes. This can be done by using Graph-Frames' breadth-first search (BFS) implementation[10], which is able to use arbitrary at-

---

[10]`https://graphframes.github.io/api/scala/index.html#org.graphframes.lib.BFS`

tribute selectors for the source and target nodes. For this algorithm, either all vertices are selected or only ones with few relations, meaning small degrees, which can be determined by joining *graph.degrees* into the vertex data frame beforehand. It can be limited by a maximal path length that should be set to three or four to not include too complicated paths which wouldn't make for useful relations in a simplified graph.

After executing the BFS algorithm, the output needs to be processed. For instance, the resulting data frame can be filtered by only containing a single relation type or by summing up the weights all contained edges and thresholding this value as previously discussed. Also, all paths of length zero and one have to be filtered out because they don't provide any meaningful information. Then, a description has to be chosen for the resulting relations. A simple approach would be to concatenate the company names along the path (without the source and target), or using the relation types with ellipses (...) to make it evident that the concerned relationship is not present in the source graph.

Afterwards, the *to* and *from* columns (which contain the source and target vertices) as well as the constructed relation description are extracted and concatenated with the thresholded relations from the PageRank output, and the final vertices and relations are written to the database.

6 Conclusion and Evaluation

Using the subgraph extraction algorithms from chapter 4, interesting areas in the graph can be highlighted and extracted for manual inspection or further analysis. The company group graphs are a helpful tool to understand the intricate ownership network of the global business landscape and may also be used a means to evaluate what the parent companies of customers, competitors or business partners are. With the motif finding algorithm, long chains of specific relation types like trade relations may be found, which are important to know about in case one of the involved businesses is unable to deliver its goods. It can also find star structures or cycles in the graph, which are normally hard to detect by manual search.

The graph simplification algorithm describe in chapter 5 is useful for getting a better understanding of the structures present in the subject graph. It can also be used to provide users with a basic navigation structure when the graph gains more relations so its connectedness increases. This graph might be preloaded in the Corporate Landscape Explorer developed during the previous bachelor's project to show a base skeleton of companies that include its most important connections.

6.1 Technical challenges

The graph processing algorithms in this paper are to be seen as examples of how the specific graph features could be extracted. They are in no way the most efficient implementation (particularly because GraphFrames is quite slow in processing the huge amounts of data contained in the data lake), but are already quite useful for extracting different views of the subject graph that represent specific features like contained company groups or patterns that provide a different way to navigate the graph, instead of just traversing it using outgoing relations.

The inclusion of automatically extracted relations holds many challenges of its own, because most algorithms in this paper are heavily dependent on the quality of the relations. For instance the company group extraction algorithm is extremely vulnerable to the introduction of wrong ownership relations into the subject graph, because it is based on finding connected components in it, which leads to the two company groups being considered as one, leading to a lot of wrong data.

6.2 Outlook

When more relation types become available in the data lake using the relationship extraction methods contained in the text mining subproject in Ingestion, many more extraction algorithms can be implemented, like the extraction of production chains of multiple companies using supply and trade relations (utilizing motif search like in chapter 4.2) or the extraction of competitor graphs (using connected components like in chapter 4.1).

Another issue is the performance of the discussed algorithms. To increase it, they can be partially ported over to the GraphX framework integrated into Spark. This is easily integrated into the current algorithms, because GraphFrames offers conversion methods that translate its GraphFrame structure into GraphX graphs and back. Because it works on a much smaller data structure that is tightly packed into a binary format, the data exchange over the local network is faster and less data needs to be stored between stages. Using this method, particularly slow sections of the algorithms can my be sped up by a lot, depending on how much data is removed from the vertices before execution.

References

[1] Abgas-Skandal: VW-Zulieferer kämpfen mit Kurzarbeit. In: *Automobilwoche* (2016), October. `http://www.automobilwoche.de/article/20161018/AGENTURMELDUNGEN/310189944/abgas-skandal-vw-zulieferer-kaempfen-mit-kurzarbeit`

[2] Bundesrepublik Deutschland, Statisches B.: *Unternehmensregister.* `https://www.destatis.de/DE/ZahlenFakten/GesamtwirtschaftUmwelt/UnternehmenHandwerk/Unternehmensregister/Tabellen/UnternehmenBeschaeftigteUmsatzWZ08.html`, Abruf: 2017-07-18

[3] Ehmüller, Jan: *Evaluation of Entity Linking Models on Business Data.* July 2017

[4] Flaig, Imelda: VW-Abgasaffäre belastet Zulieferer im Land. In: *Stuttgarter Zeitung* (2016), October. `http://www.stuttgarter-zeitung.de/inhalt.baden-wuerttemberg-vw-abgasaffaere-belastet-zulieferer-im-land.06ea4637-d060-4109-ad3c-c341a04bf1bd.html`, Abruf: 2017-07-18

[5] Janetzki, Jonathan: *Feature Extraction for Business Entity Linking in Newspaper Articles.* July 2017

[6] Löper, Lando E. N. ; Radscheit, Matthias: *Evaluation of Duplicate Detection in the Domain of German Businesses.* July 2017

[7] Otsu, Nobuyuki: A threshold selection method from gray-level histograms. In: *IEEE transactions on systems, man, and cybernetics* 9 (1979), Nr. 1, S. 62–66

[8] Pabst, Leonard: *Efficient Blocking Strategies on Business Data.* July 2017

[9] Page, Lawrence ; Brin, Sergey ; Motwani, Rajeev ; Winograd, Terry: The PageRank Citation Ranking: Bringing Order to the Web. Stanford InfoLab, November 1999. – Technical Report

[10] Schneider, Alec: *Evaluation of Business Relation Extraction Methods from Text.* July 2017

[11] Strelow, Nils: *Distributed Business Relationships in Apache Cassandra.* July 2017

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used any other than the declared resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Potsdam, July 21, 2017

....
Milan Gruner