TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Kristiina Konno 142877

EXAMINING WORDNET TYPE SEMANTIC HIERARCHIES WITH GRAPH DATABASE NEO4J

Bachelor's thesis

Supervisor: Ahti Lohk
PhD

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kristiina Konno 142877

WORDNET-TÜÜPI SEMANTILISTE HIERARHIATE UURIMINE GRAAFI ANDMEBAASIGA NEO4J

Bakalaureusetöö

Juhendaja: Ahti Lohk
PhD

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Kristiina Konno

10.01.2018

Abstract

This work focused on wordnet type semantic hierarchies and graph databases. Several aspects go into creating wordnet and during that process, many different mistakes might occur. All wordnets need to be validated, to correct said mistakes. There are differet types of errors in a wordnet and also several different ways to validate a wordnet. This work used different test patterns that help find different errors on wordnets.

The problem is that there is no universal tool for validating wordnet and most research teams do not have the resources to create a tool on their own. This work aims to create a universal program, which can detect errors in all the wordnets with the help of said patterns.

Graph databases were used as the basis of this program. All different types of databases are good in their own unique way. However, since graph databases are made with large amount of connected data in mind, then graph databases were used as the basis of our program. More specifically, Neo4j was chosen as our graph database.

The created program fullfilled our main goal and can be used to verify all wordnets. It was possible to create a graph database program without any previous knowledge in said area. Only two patterns were not realized and there are a few shortcoming that will be implemented in the next version of the program. On the other hand, our program creates a visual representation of the pattern results. Moreover, since all nodes are represented only once there, then there is a possibility to discover new patterns.

This thesis is written in English and is 46 pages long, including 6 chapters, 29 figures and 1 tables.

Annotatsioon

Wordnet tüüpi semantiliste hierarhiate uurimine graafi andmebaasiga NEO4j

Antud lõputöö keskendub wordneti tüüpi semantilistele hierarhiatele ja graafi andmebaasile. Wordneti loomise ja täiendamise käigus võib tekkida mitmeid semantiliste hierarhiates esinevaid kõrvalkaldeid. Taoliste vigade parandamiseks on tarvis wordnette valideerida. Veoolukordi, mis wordnetis kontrollida on erinevaid. Samuti on ka erinevaid meetodeid wordneti valideerimiseks. Käesolev lõputöö kasutab wordneti valideerimiseks testmustreid.

Probleem on selles, et ei ole olemas universaalselt vahendit wordnetide valideerimiseks ja mitmetel uurimisrühmadel ei ole kas oskusi või rahalisi vahendeid enda programmi loomiseks. Selle lõputöö eesmärk on luua universaalne graafiandmebassipõhine rakendus, mis suudab tuvastada vigu igas wordnetis, kasutades mainitud mustreid.

Loodud rakenduse aluseks kasutati andmebaasi. Kõik erinevad andmebaasi tüübid on head ja vajalikud omal unikaalsel viisil. Kuid, kuna graafi andmebaasid on loodud suurte ja ühendatud andmebaaside tarbeks, siis just graafi andmebaasid on ma selle lõputöö raames loodud programmi aluseks. Täpsemalt on kasutatud graafi andmebaasi Neo4j.

Loodud rakendus täitis oma eesmärgi ja seda saab kasutada wordnetide valideerimiseks. Oli võimalik luua graafi andmebaasil põhineva programmi ilma eelnevate teadmisteta graafi andmebaasidest. Ainult 2 mustrit jäid realiseerimata ja olid veel mõned puudujäägid mis implementeeritakse loodud programmi järgmises versioonis. Teisest küljest loob meie programm visuaalse graafi tulemustest. Ning, kuna kõik synsetid lisatakse sinna vaid ühe korra, siis on võimalus avastada uusi mustreid.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 46 leheküljel, 6 peatükki, 29 joonist, 1 tabelit.

List of abbreviations and terms

Neo4j	Graph database management system
PHPSpreadsheet	PHP library made to read and write spreadsheet files
Synset	Set of synonyms
Semantic network	Graphic notation for representing knowledge in patterns of interconnected nodes
Semantic hierarchy	Semantic network as a directed graph
Wordnet	Lexical database
Hashmap	Map based collection class that is used for storing Key & value pairs
ACID principles	Atomic, Consistent, Isolated, Durable. Principles upon which relational databases are built.

Table of contents

Introduction

Everyday we learn that science plays a great part in our everyday life. Even though, the number of scientific research and publication has increased tremendously in past 50 years, for around 5%, [1] these publications are yet to fill the gap of knowledge still present. This work aims to fill one of those gaps.

This project focuses on wordnets and more specifically validating them. There are different tools for validating wordnets, but more often than not, each team of researchers has to create their tools, which takes time and resources, that not all teams of researchers have. This project aims to show, that creating a validation tool is more convenient and accessible than it seems when you are using the right tools.

With this work, we want to find out, if it is possible to validate a wordnet using graph databases. The reason is, that graph databases are made for relationship-heavy databases, and wordnets fit that criterion perfectly.

This work consists of 6 chapters.

Chapter 1 focuses on the background information on wordnets. We discover what wordnets are, and then take a closer on the different kinds of errors in a wordnet and how to detect them.

Chapter 2 focuses on Neo4j and graph databases in general. We take a look at databases in general and discover what makes graph databases different and how is data stored in graph databases.

Chapter 3 focuses on all the different test patterns for validating a wordnet and how to query their results from a Neo4j database.

Chapter 4 focuses on the program created and all the different parts of it.

Chapter 5 is the evaluation of the whole work, and chapter 6 is the conclusion.

1 Background

In this chapter, we will take a look at wordnets. First, we will cover what is a wordnet, and then move onto the different types of errors in a wordnet and how to detect them.

1.1 What is a wordnet?

Wordnet is a semantic network with different types of relationships connecting synsets. A synset contains a unique id, a set of word making up the synonyms as well as the definition of said words. The most important and the most used relationship in wordnet is the IS-A relationship.[2] For example, a rose IS-A flower. There are all together around 40 different types of relationships in the Estonian wordnet [3], but we will only use the IS-A relation. Because the IS-A relationship has a direction [2], then our structure forms a hierarchy. Moreover, this said hierarchy is what we will be working with, in this project.

In the wordnet, the IS-A relationship is denoted as the hyponymy or troponymy relation. Both of these are clarifying relationships, but hyponymy is for nouns only and troponymy is for verbs only. The opposite of both of these relationships is hyperonymy relationship. [4] For example, a rose is a hyponymy of a flower, but a flower is a hyperonymy of a rose. Troponymy and hyponymy relationships are very similar, but at the same time, the compiling of troponymy relations is significantly harder, and not all verbs can be gathered under one root synset. A root synset is one, which has no hyponymy or troponymy relations. Meaning, it is the most general noun or verb. For nouns, the main root synset is 'entity,' and in verbs one of the root synsets is 'be.' [2]

1.1.1 History and Applications

There are around 70 wordnets in the world. [5] All of the wordnets have links to the first wordnet ever created. It is known as the mother of all wordnets. It is called the

Princeton WordNet or WordNet [1] [2]. As can be assumed from the name, it was developed under a project in the Princeton University headed by George A. Miller. The primary purpose was learning more about how the human semantic memory is organized. [5]

Over the years, wordnets have been put to use in many different areas. They have been used as the basis of other knowledge resources, such as geographical and medical wordnets among others. A more widespread application for wordnets is different online dictionaries. For example, Synonym [2]is a dictionary, which uses a wordnet. It can be used to look up synonyms, antonyms, and definitions for any word in the English language. There are also many different dictionaries online that get their information from the Princeton WordNet, such as Visual Thesaurus [3], Visuwords [4], and WordVis [5]. All of them incorporate the visual aspects of wordnets as well as using them as dictionaries. [2]

1.1.2 Building a wordnet

Building a wordnet is time- and resource consuming. Several options make building a wordnet less consuming. Before building, three questions need to be answered: what kind of lexical resources to use, what kind of building model to use, and how automated is the building process? [2]

What kind of lexical resource to use?

Many different lexical resources can be used, and we will go over most of them. It is also possible, to use different lexical resources at the same time and also, lexical resources can be used during validation and evaluation as well as in the building phase. [2]

---

[1] https://wordnet.princeton.edu/

[2] http://www.synonym.com/

[3] https://www.visualthesaurus.com

[4] https://visuwords.com

[5] http://wordvis.com/

A monolingual dictionary can be used to get taxonomical relations as well as get definitions, synonyms, usage example and other information about words. A bilingual dictionary can be used to translate synsets from a source language to another. [6] On-line encyclopedias can be used as mono- or bilingual resources. [7]

Another lexical resource can be other wordnets. Usually, they are used as a source, and their synsets are translated to the target language. [8]

What kind of building model to use?

There are two different building model that can be used - expand and merge model. [2] The main difference is that expand model translates synsets or base concepts and also takes all the relationships from another wordnet and later expands it. [8] The merge model defines the synsets and semantic relations of the target relations first, and then aligns it with a different wordnet. For example, Princeton WordNet might be used. [2]

The expand models advantages are, that the building of a wordnet is more fluent because there is no need to think of the concepts for the target language and that fluency makes building the wordnet faster. The other benefits are that semantic relations can be borrowed from the source wordnet and the resulting wordnets are all very compatible. The disadvantage is that sometimes there are no equivalent concepts in the target language. Thus, using expand model is recommended, when the source and the target languages are semantically close. [2]

The merge models disadvantage is that there is a need to find the base concepts and it is significantly slower than the expand model. To overcome the shortcomings of both models a hybrid model was created. [2]

How automated is the building process?

Automation can be extended from the building process to the evaluation and expansion phase. There are three levels of automation - manual, semi-automated and automated. [2] The manual approach creates the best results and is reliable, but it is, understandably, time- and human resource consuming. [9] Since many research groups often don't have the needed amount of time or human resource at hand, then automated or semi-automated approaches are used. The increased popularity of automated approaches is also because several bilingual online dictionaries can help in translating

source synsets into the target ones. The negative side of automated approach is that the results are usually faulty. The synsets have wrong, or missing words and some relationships are wrong or missing altogether. [10]

However, with every automation level validation is still necessary. As we learned earlier, the manual approach is more resource consuming, but the automated approach creates an inaccurate result. If we assume, that the validation takes the same amount of time and resources with every automation level, then the automated approach is still preferable.[2]

1.2 Different kinds of errors in wordnet

Previously we learned what a wordnet is and how it is built. We also learned that mistakes might occur while building the wordnet. We will take a closer look at all the mistakes next. There are three different kinds or errors - syntactic, semantic and structural errors. [2]

1.2.1 Syntactic errors

Syntactic errors are those which appear in the source files. Syntactic errors include empty or duplicate IDs or lexical units, an empty position of speech or sense, incorrectly entered words, typographical and spelling errors. [8]

1.2.2 Semantic errors

Semantic errors are connected to synsets and their relations, but not with the source file. Semantic errors include wrong or missing relations, wrong or missing words in a synset, and wrong definitions. [2] These errors also include overgeneralization meaning when synset is given a hyperonym that is too generic and unfinished or unjustified multiple inheritances. [11] For example in version 70 on Estonian wordnet, hotel and hostel are both buildings and institutions, but motel's only parent was institution. The last example was a case of unfinished multiple inheritances. [12]

Semantic errors also include cases, where two words are mixed up that have a similar sound or spelling. [2] Also, all cases of lexical ambiguity fall under semantic errors. These are the words, which have the same spelling but different meanings. In some cases, the same word has two entirely different meanings. In other cases, the meanings

are similar. For example, a chicken can mean both a bird and food. There is a possibility to add it as one word with both meanings, or as two different words with different meanings. [13]

1.2.3 Structural errors

Structural errors can be found in the 'relation definitions and the link structure without going more deeply into the semantics of the elements linked'. These errors can be found by the following patterns: rings, dangling uplinks, orphan nodes, small hierarchies, and roots. [2] Rings have a top and a bottom node and two different chains connecting them. [14] Dangling nodes pattern refers to a subgraph, where a node has two parents. One of the parents is connected to a more prominent subgraph, but the other one is a root on its own. [2] It can also be seen in the figure below.

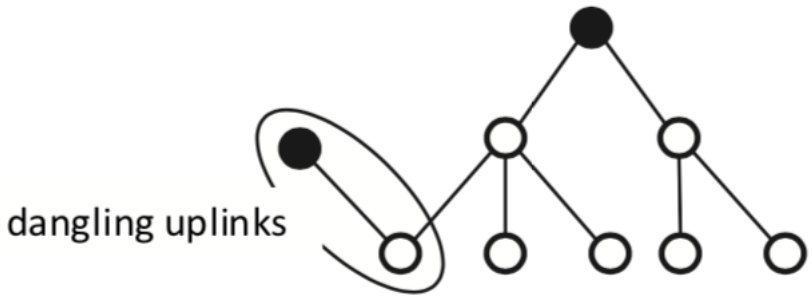

Figure 1 - example of dangling uplink

Orphan nodes refer to synsets, which are not connected to any other synsets with hyperonym or hyponym relations. [2] Small hierarchies are identified as the subgraphs, that only have three levels after the root element. Root nodes as mentioned before are at the topmost levels in the hierarchy. [15] All of those patterns can be used to find structural errors in a wordnet.

1.3 How to detect wordnet errors?

There are several ways to detect errors in a wordnet. One possibility is to use different lexical resources, that can also be used in the building process. One method of using lexical resources is to check a wordnet against a monolingual resource in the same language and compare the words presented in the same synsets and whether these words are used to together. Another method is to check the wordnet against a monolingual explanatory dictionary and check if the definition of a word contains any of its

synonyms or hyperonyms. This method helps to find outlier or missing words in a synset. Another possibility to check how good the wordnet is and whether the vocabulary is sufficient enough is to employ the wordnet in some semantic analysis tasks. [2]

The other possibility to detect errors in a wordnet is considering the semantic and lexical relations and the rules between them. One option is to use a large group of regular people mainly from the online community and ask them to rate sets of synonyms and detect errors that way. To overcome structural problems of a wordnet, then there is a possibility to concentrate on specific structural problems that directly violate the IS-A relationship. One of those problems is, for example when a hyponym contradicts its hyperonyms nature. [2]

In this work, however, we are focusing on the method of detecting errors in wordnet, where patterns are applied to a wordnet. We mentioned a few pattern under structural errors. There are also several test patterns presented in Ahti Lohk's doctoral thesis, that will be implemented in this project. We will look more closely at each test pattern in chapter 3.

2 Neo4j

In this chapter, we will take a closer look at graph databases and why, in some cases, it is more beneficial to use them over relational databases. We will go over what graph databases are, compare them to other databases, how data is stored in graph databases, and how queries are performed in Neo4j graph databases.

2.1 What is it?

2.1.1 Graph theory

Before we can delve into graph databases, we must first go over graph theory, because graph databases rely on the graph theory. Graph theory can be traced back to Leonhard Euler, who solved the Königsberg bridge problem in the year 1735. Nowadays it has become an essential area of mathematical research and has applications in chemistry, social sciences, and computer science among others. The term graph refers to a set of vertices and the edges that connect them. Moreover, there are different types of graphs, for example, in a multigraph, any two vertices are joined by more than one edge. However, graph databases use a directed graph, where each edge also has a direction. [17]

2.1.2 The start of graph databases and why companies use them.

When thinking about big world-changing tech companies, then Google, Facebook, and Twitter inevitably cross the mind. Moreover, there is a common theme in the backbone of these companies - Graph Databases. Before that, most companies biggest battle was with relational databases, because they were slow in querying large amounts of data. The birth of graph databases changed the landscape of tech companies and allowed nowadays tech enterprises to arise. Graph databases have become very popular and are used for healthcare, gaming, retail, media and social networks among many others. [18]

There are several reasons why some companies are switching to graph databases. One of them is the performance speed. People want to get information in under a second, and with a broad web application, it might be tricky. However, graph databases, with their embracement of relationship, make querying time a lot faster in databases with many relationships. The other reasons, why some companies are switching over to graph

databases, is the ease of changing the database as the application changes and also, there are graph databases, that already embrace the ACID principles. The ACID principle are the following:

- Atomic - "All operations in a transaction succeed, or every operation is rolled back."
- Consistent - "On transaction completion, the database is structurally sound."
- Isolated - "Transactions do not contend with one another. Contentious access to state is moderated by the database so that transactions appear to run sequentially."
- Durable - "The results of applying a transaction are permanent, even in the presence of failures." [18]

2.1.3 Graph database properties

The main characteristics of graph databases are, embracing relationships, performance speed, flexibility, and agility. Next, we can have a closer look at each of the qualities. [18]

The Graph database is one of the few that genuinely embraces relationships. They are the main component of the database. The nodes and relationship together make the database more expressive and more straightforward than other NoSQL and relational databases. The resulting model is a close match to the real-world model. [18]

Because relationships are a primal part of the graph database, then querying connected data is faster as well. The performance of the graph database most of the times remains relatively constant with the growth of the database. The execution time of a query is proportional to the graph traversed. [18]

Graph databases brought a wave of flexibility and agility to the database world. It is easy to add new nodes, kinds of relationships and other attributes to the graph without disrupting the application functionality. That means that developers no longer have to model the database in detail ahead of time. Instead, it can change as the application changes. [18]

2.1.4 Labeled property graph model

The labeled graph model has the following features:

- It is made up of nodes, relationships, properties, and labels.
- The node properties are in the form of key-value pairs.
- Nodes can be tagged with one or more labels, that group nodes together, and indicate their roles in the dataset.
- A relationship must always have a direction, a name, and an end node. Relationship connects nodes and gives structure and semantic clarity to the graph.
- The relationship can also have properties.

The changes between relational and graph databases can be seen already while analyzing the database. Most developers start with a whiteboard sketch to describe the domain. Also, more often than not, it is in the form of a graph. With the relational database, before creating the database, usually, a graph-like sketch is made with all the tables, and their properties as nodes and the relationships are also drawn out. Then a real database could be created. However, the difference starts there. With graph databases, the exact sketch is what is stored in the database. No transformations have to be done from the sketch to the actual database. [18]

2.2 Neo4j and other databases

2.2.1 Where it lies in the world of databases and a bit about different types of databases

There are two different types of databases - relational and NoSQL databases and graph databases are a type of NoSQL database. To make a rightful comparison between relational and graph databases, we must fully understand how relational databases work and what are other kinds of NoSQL databases. [18]

2.2.1.1 Relational databases

Relational databases are made up of tables. Each column holds a specified type of value. Each row represents one object in the table. Usually, a primary key is added to each table, that holds a unique identifier. That makes it possible to reference another table by its primary key. [18]

To guarantee us a safe environment in the relational database world, the ACID principles must be upheld. The ACID transactions are the following:

- Atomic - "All operations in a transaction succeed, or every operation is rolled back."
- Consistent - "On transaction completion, the database is structurally sound."
- Isolated - "Transactions do not contend with one another. Contentious access to state is moderated by the database so that transactions appear to run sequentially."
- Durable - "The results of applying a transaction are permanent, even in the presence of failures." [18]

2.2.1.2 NoSQL databases

Relational databases are far older than NoSQL databases. All the first web apps used relational databases as their backbone. However, as the industry grew, so did the databases. Then it became clear, that relational databases were too slow for large amounts of data. That was the beginning of the rise of NoSQL databases. They did not follow the outdated ACID transactions. Instead, the term BASE was used to describe NoSQL storage properties. BASE stands for the following:

- Basic availability - the store appears to work most of the time
- Soft-state - Stores do not have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
- Eventual consistency - Consistency is displayed at a later point.

So we can see, that NoSQL stores value availability and have to be consistent but only in the future. Those looser properties allow more freedom for the developers. [18]

Next, we will take a closer look at the four leading types of store.

Document stores

Document databases resemble a filing cabinet in that they are used for storing and retrieving documents by their ID. Documents can also be accessed by their attributes, but the database has to be indexed for that. Indexed can also be used to retrieve an array

of documents. With indexes, the database queries are faster, because that allows the database to scan fewer documents, to retrieve the correct ones. Without indexes, all of the documents would have to be scanned. [18]

Key-Value stores

Key-Value stores are similar to Document stores. In the most straightforward term, they are a large hashmap. The hashmap is distributed into buckets which are replicated onto several machines, to avoid failures. They are easy to use for the client. They just have to store and retrieve data by the key. [18]

Column family stores

Column Family stores are made up of rows, which have a simple column, with one name and the value, or a super column, with multiple names and corresponding values. The rows make the data into a nested hashmap structure. Unlike Document and Key-Value stores, Column family stores are more expressive. The store does lack joins and thus getting an insight of a more extensive amount of data requires some external applications processing. [18]

2.2.2 Comparison between relational and graph databases.

The main difference between relational and graph databases is their purpose. Relational databases were made especially for different forms and tables. They fulfill that purpose exceedingly well. However, they were not made for connected data. It can be done with the help of foreign keys and joining tables, but it makes the queries significantly slower. Graph databases, on the other hand, are made for connected data, and the database is built for that specific purpose. To further bring out that difference we will look at an example. [18]

When we think about a basic friends database, where we have people and their friends. It is quite easy to find the 'friends-of-friends' in both relational and graph databases. However, if we want to find the second, third or even sixth-degree friends, then the difference is significant. Looking at the same network, if there would be 1,000,000 people and each has 50 friends. Looking at first degree friends, 'friends-of-friends,' then the difference is not that evident, and both databases can finish the query in around 0.01 seconds. The difference between the graph and relational database query time get bigger

and bigger as we go into a higher degree. For example, if we look at fourth-degree friends, graph database can finish the query in 1.4 seconds. However, the relational database takes 1550 seconds to complete. Moreover, that is where we can most clearly see the difference between relational and graph database. They are both great, at what they were made to do. [18]

2.3 Data storage in graph databases

An understandable question at this point is 'How are graph databases able to do all that?'. This is what we are focusing on in this chapter. There are two key terms we will focus on - native graph storage and native graph processing. Native graph storage means that the storage is designed primarily for graphs. Some non-native graph storage databases, in reality, use a relational, object-oriented or some other database in the background. Native graph processing means that in the database there are no global indexes. Instead, each node has a direct reference to its adjacent nodes. All of the benefits of graph storages that we discussed before, the only come into play, when the graph database use native graph storage and processing. Next, we will learn, how this is implemented in the database using Neo4j as the example. [18]

In NoSQL databases, the data is stored in stores, and all the records, in the stores, have a fixed-size length. The benefit, of having fixed-size records, is enormous. If we look at the nodes store, for example, then we see, that each node has a fixed size of 9 bytes. [18] If we want to find the node with ID 100, then we just look at the node starting from byte 900. That makes the cost of that specific query only O(1).

The node record is made up of a flag, with the size of 1 byte, the ID of the first relationship and ends with the ID if the first property, both with the length of 4 bytes. The flag represents, whether the node is currently being used. There is more information in each record in the relationship store, but the size is still fixed. The ID in the node store, in fact, points to the first relationship in a relationship chain. We already know, that one node can have multiple relationships. Each record in the relationship store has the start and end nodes ID, a pointer to the relationship type, pointer for the next and previous relationship records for start and end node and a flag indicating if the node is the first node in the list. [18]

So, when we want to find all the outgoing relationships for a node, we find out its first relationship ID. Then we go to the relationship store, get the end node. Then we go the next start node relationship by the stored ID. And repeat that process. [18]

In the property store, there are the key-value property pairs for nodes and relationships. There is a different system for properties since the string or array type values can be of a varying length, but the property store is still of fixed length. [18]

2.4 Queries in Neo4j

For this project, we chose Neo4j as our graph database. Neo4j queries are written in cypher query language. [18] For this reason, we will look at cypher and how to build queries with the cypher.

Cypher is made to be user-friendly and easily understandable by anyone. That is because the queries are similar to how we would describe graphs on paper. Cypher enables us to get data that matches a specific pattern. [18] It is most comfortable to start with a simple example and explain from there.

```
(s1:Synset {sh:'{jook_n_1}'})-[:has_hyponym]->(s2:Synset)
```

Code snippet 1 - cypher pattern example

In the code snippet above, the pattern describes all the hyponyms of {insert sample sh}. We have a node s1 with the label Synset. The labels help us get only synonyms. In our project, we only have Synsets, but in a bigger database, there could be numerous different labels for nodes. Our node s1 also has a specified value for its property 'sh' which in our case is the string containing all the words making up the synset. We also have specified, that we want all outgoing Synsets that have the relationship type 'has_synset'. [18] In our case, this pattern helps us get all the hyponyms of the synset {jook_n_1}

The code sample above is only the pattern part and does not give as any results. The get the results, this pattern has to be turned into a working query. [18] An example of a full query with the previous pattern is shown in the code snippet below.

```
MATCH (s1:Synset {sh:'{jook_n_1}'})-[:has_hyponym]->(s2:Synset) RETURN s1, s2
```

Code snippet 2 - cypher match statement example

As can be seen from the code snippet below, to get the results, a match statement has to be used. The pattern is to be put after the match clause, and all the nodes that we want to get in return have to be put after the return clause and separated by a comma. [18] With the match statement, a where clause can be added as shown in the code snippet below.

```
MATCH (s1:Synset)-[:has_hyponym]->(s2:Synset)
WHERE s1.sh = '{jook_n_1}'
RETURN s1, s2
```

Code snippet 3 - cypher match statement example

As it can be seen, then the property sh of node s1 is stated in the match clause. The outcome does not change with this query from the last one. There are, of course, more to the cypher queries then what we learned so far. [18] In the next chapter, we will learn more about cypher with our test patterns.

3 Test patterns overview

In this chapter, we will take a look at all the test patterns presented in Ahti Lohk's doctoral thesis and the queries for said patterns. We will start with the most straightforward patterns and move on to more complicated ones.

3.1 Heart-shaped substructure

Heart-shaped substructure contains five synsets connected as shown in the figure below. The two parent nodes s1 and s3 are both directly connected to the child node s2. With the child node s4, parent node s3 is directly connected, and the other parent node s1 is indirectly connected through node s5. [2]

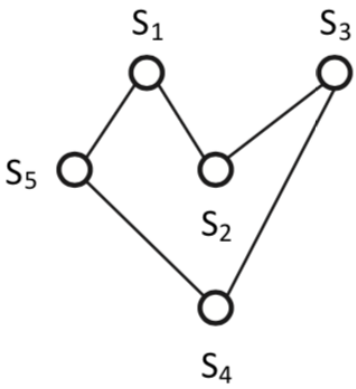

Figure 2 - heart-shaped substructure

There is also a full heart-shaped substructure which is shown in the figure below. In this pattern, the only difference is, that with the lowest level child node s4, both top-level parent nodes are indirectly connected through another node. Both of these patterns can be used to identify wrong relations. [2]

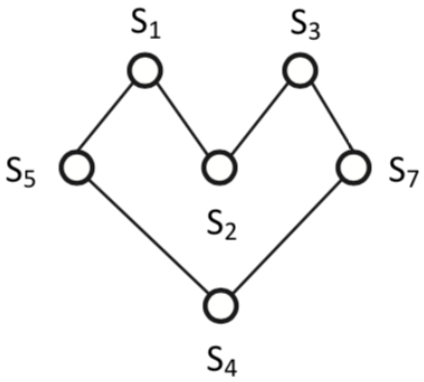

Figure 3 - full heart-shaped substructure

In all the figures in our work, all hyperonymy relations go from top to bottom, and all hyponymy and toponymy relations go from bottom to top. In our code examples, we will be using 'has_hyponym' relation and thus we look at all figures with the relation going from bottom to top. [2]

The Neo4j query for this pattern is similar to our example in the previous chapter. The queries for both of the heart-shaped substructures is shown below.

```
MATCH (s5:Synset)<-[r2:has_hyponym]-(s1:Synset)-[r1:has_hyponym]->(s2:Synset)
<-[r3:has_hyponym]-(s3:Synset)-[r4:has_hyponym]->(s4:Synset)-[r5:has_hyponym]->(s5)
RETURN  s1, s2, s3, s4, s5
```

Code snippet 4 - heart-shaped substructure query

```
MATCH (s6:Synset)<-[r2:has_hyponym]-(s1:Synset)-[r1:has_hyponym]->(s2:Synset)<-[r3:has_hyponym]-
(s3:Synset)<-[r4:has_hyponym]-(s4:Synset)-[r5:has_hyponym]->(s5)-[r6:has_hyponym]->(s6)
RETURN  s1, s2, s3, s4, s5, s6
```

Code snippet 5 - full heart-shaped substructure query

3.2 Short-cut

In this subgraph, two synsets are directly connected, but also have a longer chain of nodes connecting them. It can also be seen in the figure below. One of the links in the subgraph is redundant, and thus there is a mistake made in the building process. This error may occur when a lexicographer has made a more precise link between two synsets but forgot to remove the wrong relations. This pattern may appear to be a case of multiple inheritances, but the difference is, that the bottom synset is connected to the top one both directly and indirectly. [2] This pattern can also be seen in figure below.

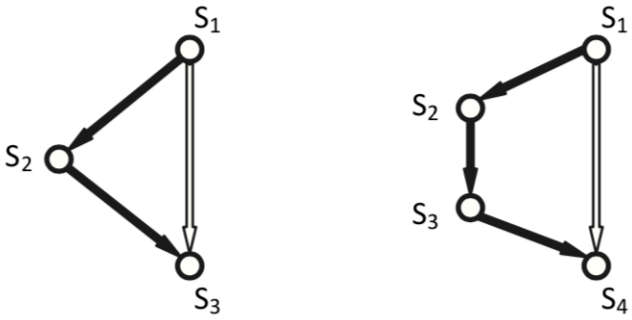

Figure 4 - Example of a short-cut

When it comes to realizing this pattern, then we cannot just use a pattern as we learned before. The chain of nodes may be of varying length. As a solution, we can 'multiply' a relationship as shown in the code snippet below {[:has_hyponym*2..10]}. [19] If we define the relationship between two nodes as this, then we are looking for a length of 2 - 10 nodes between the two nodes, all with the same direction. In our example, we start with two, because we want at least one node to be in our chain. The working example is shown in the code snippet below.

```
MATCH Path=(b:Synset)-[:has_hyponym*2..10]->(a:Synset), (b)-[:has_hyponym]->(a)
RETURN a, b, Collect(extract(n IN nodes(Path)| n.sh) ) AS path
```

Code snippet 6 - short-cut pattern query

In our match statements, we have specified 'b' as our top node and 'a' as our bottom node. We can declare a variable in our match statement. Also, we do not have one chain of elements in our match statement. We can separate two chains with a comma. In our return statement, Collect() helps us collect all results. For example, if we have more than one different chain connecting the top and bottom node, then we can get them all under the variable path as an array. The extract statement, in our case, returns all properties 'sh' of every node in the Path variable as a list. The syntax of the extract statement is as follows: {extract(variable IN list | expression)}. To get our path a list, we needed to put it in the nodes() statement. [20] So, in the results, we get the path from the top node to bottom as a list of strings containing all the synsets.

3.3 Ring

We learned already a little bit about the ring pattern. It is quite similar to short-cut structure because we have a top and bottom node that are connected to each other. The

difference is that in short-cut there is a direct link between the two nodes. In the ring structure, the top and bottom nodes are connected indirectly to each other by two different chains of synsets. There are two different types of rings - symmetric and asymmetric. In the symmetric ring topology, the two chains connecting the top and bottom nodes have the same length, but in the asymmetric topology, the length of the two chains is different. The error that may occur is that sometimes the two parents of the bottom node have different properties that are inherited to the bottom node. [2] This is also shown in the figure below.

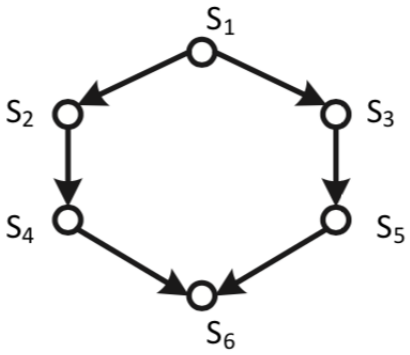

Figure 5 - Example of a ring structure

The query for this pattern is also similar to the short-cut pattern. However, in this case, we declared two different paths in our match statements, but we returned only one of them with the same Collect() statement as above. In this case, however, the collect() is necessary, because we always get the same result twice with the two paths switched. The collect statement allows us to get every result only once and thus there is no need to deal with duplicates. The full query is presented in the code snippet below.

```
MATCH Path1 = (a:Synset)<-[:has_hyponym*1..20]-(b:Synset), Path2 = (b)-[:has_hyponym*1..20]->(a)
RETURN a, b, Collect(extract(n IN nodes(Path2)| n.sh) ) AS path
```

Code snippet 7 - ring pattern query

3.4 Synset with many roots

In all of the previous patterns, we dealt with synsets with many parents in the bottom. However, in the previous patterns, the multiple inheritances came together as a short-cut or a ring. In this pattern, the multiple inheritances flow into different root elements. It is beneficial, to look at the branches and whether all the connections are correct. The

typical error connected to this pattern is that some of the root synsets are too specific to be a root synset. [2] This pattern is also shown in the figure below.

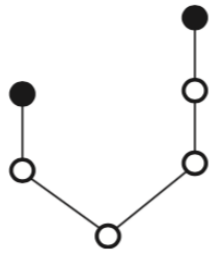

Figure 6 - Example of synset with many roots pattern

The query for this pattern is similar to the one for the ring. Only, in this case, there are two parents, and we have to check, that they do not have an incoming hyponym relation. The specification has to be put in the where clause. The full working query is in the code snippet below. In this case, as well, we collect only one path and extract it to be an array of synsets.

```
MATCH Path1 = (b:Synset)-[:has_hyponym*1..10]->(a:Synset), Path2 = (a)<-[:has_hyponym*1..10]-(c:Synset)
WHERE (NOT (b)<-[:has_hyponym]-(:Synset)) AND (NOT (c)<-[:has_hyponym]-(:Synset))
RETURN a,  Collect(extract(n IN nodes(Path1)| n.sh) ) AS path
```

Code snippet 8 - synset with many roots pattern query

3.5 Closed subset

The closed subset is a pattern, where all of the nodes are among two levels. In a closed subgraph, there are all children of every parent and every parent of every child. There could be only one parent and all of its children, but we are interested in the cases, where at least one child node has two parents. Each set of children for a parent is called a subcomponent. This pattern is useful for examining connected subcomponents and all of the relations. Let's take a look at a closed subset with two parent nodes and thus two subcomponents and also a synset that is the child of both parents. We can examine why other members of subcomponents number 1 are children of parent number 2 and other similar questions. [2] Example of a closed subset is shown in the figure below.

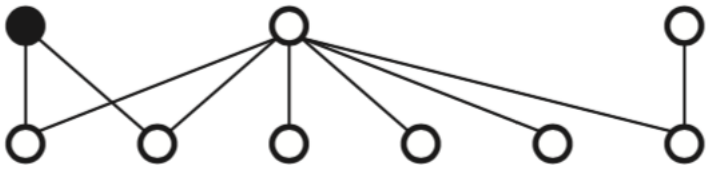

Figure 7 - Example of a closed subset

Realizing this pattern was a little bit more difficult. Doing it with a single query is a bit too complicated. For this reason, the results are collected in the front-end recursively. It all starts with getting every node that has two parents. Then collect all the nodes into one list and recursively get all parents of every child and all children of every parent. If a node has already been checked for parents or children, then it will be overlooked. That ensures that all the nodes will be collected and the recursion will stop at one point. All the code needed for that query is in the code snippet below.

```
private Map<String,Object> getParents(Map<String, Object> result, Map<String, Object> node) {
    List relationships = (List) result.get("links");
    List nodes = (List) result.get("nodes");
    String id = (String) node.get("id");
    int targetID = nodes.indexOf(node);
    nodes.remove(node);
    node.put("checked", true);
    nodes.add(node);
    List<Map<String,Object>> getParents = cypher.query( statement: "MATCH (:Synset {id:{id}})<-[:has_hyponym]-(a:Synset) RETURN a", map( ...objects: "id", id));
    if (getParents.size() > 1){
        checked.add((String) node.get("title"));
    }
    for (int i = 0; i < getParents.size(); i++){
        Map<String, Object> row = getParents.get(i);
        Map<String, Object> a = (Map<String, Object>) row.get("a");
        boolean exists = false;
        for (int j = 0; j < nodes.size(); j++){
            Map<String, Object> comparingNode = (Map<String, Object>) nodes.get(j);
            if (comparingNode.get("id").equals(a.get("id"))){
                if ((boolean)comparingNode.get("checked")) {
                    exists = true;
                    Map<String, Object> rel = map( ...objects: "source", j, "target", targetID);
                    if (!relationships.contains(rel)){
                        relationships.add(rel);
                    }
                }
            }
        }
        if (!exists){
            Map<String, Object> parentNode = map( ...objects: "title", a.get("sh"), "id", a.get("id"),  "label", "level3", "checked", false, "parent", true);
            List<Map<String,Object>> isRoot = cypher.query( statement: "MATCH (:Synset {sh:{sh}})<-[:has_hyponym]-(a:Synset) return a", map( ...objects: "sh", a.get("sh")));
            if (isRoot.isEmpty()){
                parentNode.put("root", true);
            } else {
                parentNode.put("root", false);
            }
            nodes.add(parentNode);
            Map<String, Object> rel = map( ...objects: "source", nodes.indexOf(parentNode), "target", targetID);
            relationships.add(rel);
            Map<String, Object> resultTemp = map(  ...objects: "links", relationships, "nodes", nodes);
            result = getChildren(resultTemp, parentNode);
            relationships = (List) result.get("links");
            nodes = (List) result.get("nodes");
        }
    }
    return result;
}
```

Code snippet 9 - getParents() function

```
private Map<String,Object> getChildren(Map<String, Object> result, Map<String, Object> node) {
    List relationships = (List) result.get("links");
    List nodes = (List) result.get("nodes");
    String id = (String) node.get("id");
    int targetID = nodes.indexOf(node);
    nodes.remove(node);
    node.put("checked", true);
    nodes.add(node);
    List<Map<String,Object>> children = cypher.query( statement: "MATCH (:Synset {id:{id}})-[:has_hyponym]->(a:Synset) RETURN a", map( ...objects: "id", id));
    for (int i = 0; i < children.size(); i++){
        Map<String, Object> row = children.get(i);
        Map<String, Object> a = (Map<String, Object>) row.get("a");
        boolean exists = false;
        for (int j = 0; j < nodes.size(); j++){
            Map<String, Object> comparingNode = (Map<String, Object>) nodes.get(j);
            if (comparingNode.get("id").equals(a.get("id"))){
                if ((boolean)comparingNode.get("checked")) {
                    exists = true;
                    Map<String, Object> rel = map( ...objects: "source", j, "target", targetID);
                    if (!relationships.contains(rel)){
                        relationships.add(rel);
                    }
                } else {
                    //error?
                }
            }
        }
        if (!exists){
            Map<String, Object> childNode = map( ...objects: "title", a.get("sh"), "id", a.get("id"),  "label", "level2", "checked", false, "parent", false, "root", false);
            nodes.add(childNode);
            Map<String, Object> rel = map( ...objects: "target", nodes.indexOf(childNode), "source", targetID);
            relationships.add(rel);
            Map<String, Object> resultTemp = map(  ...objects: "links", relationships, "nodes", nodes);
            result = getParents(resultTemp, childNode);
            relationships = (List) result.get("links");
            nodes = (List) result.get("nodes");
        }
    }
    return result;
}
```

Code snippet 10 - getChildren() function

```
private List<Map<String,Object>> getClosedSubsets() {
    checked = new ArrayList<>();
    List results = new ArrayList();
    List<Map<String,Object>> startChildren = cypher.query(
            statement: "MATCH (c:Synset)-[:has_hyponym]->(a:Synset)<-[:has_hyponym]-(b:Synset) RETURN a, Collect(b)", map());
    for (int i = 0; i < startChildren.size(); i++){
        List nodes = new ArrayList();
        List relationships = new ArrayList();
        Map<String, Object> row = startChildren.get(i);
        Map<String, Object> a = (Map<String, Object>) row.get("a");
        if (checked.contains(a.get("sh"))){
            continue;
        }
        Map<String, Object> node = map( ...objects: "title", a.get("sh"), "id", a.get("id"),  "label", "level2", "checked", false, "parent", false, "root", false);
        nodes.add(node);
        Map<String, Object> result = map(  ...objects: "links", relationships, "nodes", nodes);
        result = getParents(result, node);
        results.add(result);
    }
    return results;
}
```

Code snippet 11 - Starting function for closed subsets

3.6 Large closed subset

The smallest closed subset is the size of 1 x 1. the first number refers to the number of parent nodes and the second number child nodes. In our project, we consider as a large closed subset if it has over 100 nodes all together. The size of the largest closed subset in a certain wordnet says something about the accuracy of the wordnet. [2]

To get the results for the large closed subset, the same query is made as with the closed subset, but in the front-end, only the large closed subsets are presented.

3.7 Dense component

The dense component pattern is also a type of closed subset, the only difference is, that there are two child nodes with two identical parents. With this pattern, it can be deducted, whether the multiple inheritances are justified. It can also show if the multiple

inheritances have been finished. That is because we can see all of the other children, not only the children that have multiple parents. [2] Example of a dense component is shown in the figure below. It can be seen from the figure, that nodes s2 and s6 share common parent nodes s3 and s5.

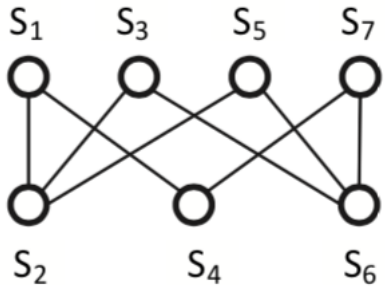

Figure 8 - Example of a dense component

As with all closed subsets, the realization of the pattern is done not only with a single query. With the dense component, each dense component is checked. If two child nodes have the same two parents, then the pattern is shown to the user. The sorting of dense components from other closed subset is shown in the code snippet below.

```
case "dense-component":
    thead.append('<tr><th></th><th>Parent</th><th>Children</th></tr>');
    var graph = false;
    var row = 1;
    for (var i = 0; i < data.length; i++){
        var nodes = data[i].nodes;
        var links = data[i].links;
        if (isDenseComponent(nodes, links)){
            if (!graph){
                getSubnetGraph(i);
                graph = true;
            }
            var rowspan = 0;
            var parentChildren = "";
            var first = true;
            for (var j = 0; j < nodes.length; j++){
                var isParent = nodes[j].parent;
                if (isParent){
                    rowspan++;
                    var parentID = j;
                    if (!first){
                        parentChildren += "<tr>";
                    } else {
                        first = false;
                    }
                    parentChildren += "<td>" + nodes[j].title + "</td><td>";
                    var isFirst = true;
                    for (var k = 0; k < links.length; k++){
                        if (links[k].source == parentID){
                            var targetID = links[k].target;
                            if (!isFirst){
                                parentChildren += ", ";
                            } else {
                                isFirst = false;
                            }
                            parentChildren += nodes[targetID].title;
                        }
                    }
                    parentChildren += "</td></tr>"
                }
            }
            tbody.append("<tr class='results'><td rowspan='" + rowspan + "' onclick='getSubnetGraph(" + i + ")'>" + getSize(data[i].nodes) + "</td>" + parentChildren);
            row++;
        }
    }
    break;
```

Code snippet 12 - sorting dense components from closed subsets

3.8 Root synset in a closed subset

As it can be presumed from the name, this pattern is also a type of closed subset. In this case, a root synset is one of the parent nodes. Looking the whole subset lets us check if the root synset is justified to be a root synset. Sometimes the pattern shows us, that the work is unfinished. Both of these mistakes are because root synsets cannot be on the same level as non-root synsets. [2]

With this pattern, each parent node is checked if it is a root synset. If so, then the subset is shown to the user. This solution is also shown in the code snippet below.

```
case "root-closed-subset":
    thead.append('<tr><th></th><th>Parent</th><th>Children</th></tr>');
    var graph = false;
    for (var i = 0; i < data.length; i++){
        if (!hasRoot(data[i])){
            continue;
        }
        if (!graph){
            getSubnetGraph(i);
            graph = true;
        }
        var nodes = data[i].nodes;
        var links = data[i].links;
        var rowspan = 0;
        var parentChildren = "";
        var first = true;
        for (var j = 0; j < nodes.length; j++){
            var isParent = nodes[j].parent;
            if (isParent){
                rowspan++;
                var parentID = j;
                if (!first){
                    parentChildren += "<tr>";
                } else {
                    first = false;
                }
                parentChildren += "<td>" + nodes[j].title + "</td><td>";
                var isFirst = true;
                for (var k = 0; k < links.length; k++){
                    if (links[k].source == parentID){
                        var targetID = links[k].target;
                        if (!isFirst){
                            parentChildren += ", ";
                        } else {
                            isFirst = false;
                        }
                        parentChildren += nodes[targetID].title;
                    }
                }
                parentChildren += "</td></tr>"
            }
        }
        tbody.append("<tr class='results'><td rowspan='" + rowspan + "' onclick='getSubnetGraph(" + i + ")'>" + getSize(data[i].nodes) + "</td>" + parentChildren);
    }
    break;
```

Code snippet 13 - code for showing root synsets in closed subset patterns

3.9 Substructure that considers the content of synsets

This pattern is the only one that considers the context of a synset. This is a pattern, where we are looking for all hyponyms of a word in synset, where the same word is a part of its hyponym. For example, paper and its hyponym newspaper. This pattern helps us find wrong relations in a wordnet. [2]

At this point, this pattern is not realized. That is because reading the contents of a synset is too complicated at this time. That is because the whole synset is saved as a string and

for this pattern, there is a need to get the whole word in a synset and not a part of the word.

3.10 Connected root synsets.

In this pattern, each root node is equipped with the number of hierarchy levels and the number of child nodes. The roots are connected by an edge. On each edge, there is the number of common child nodes and the level at which the first child element located. If two root nodes have no common child nodes, then they are not connected. This pattern helps identify unfinished work or if a synset is too specific to be root synset. This pattern was not implemented in this version of the program because it was too complicated.

4 Implementing test patterns with Neo4j

To get an idea, of how Neo4j based applications work, and to make the building of the program more manageable, the author used an example as a basis [1]. Neo4j offers many different example apps in different languages. Thus, the base example used has the name 'neo4j-movies-java-bolt' [2], taken from the examples provided by Neo4j. This example was the chosen one because it was pinned to the top 6 examples and after looking at all the top examples, this one seemed to be the most logically built.

4.1 The data

For the importing of data to work, the data must be in an excel file containing three sheets. The sheet named SH contains all IDs and their corresponding synsets. The sheet named DEF contains all IDs and their corresponding definitions. Finally, the sheet named REL contains the source ID, the type of the relationship and the target ID.

There are two ways to import data into our database. The first one is to make a script, which reads the contents of the excel file and converts them into a Cypher format queries. The other option is to use 'load CSV' [3] query in Neo4j. With the first option, significantly more work has to be done. The first possibility, which comes to mind, is to make a PHP script using PHPSpreadsheet [4], for example. With the second option, Neo4j would do a lot of the work itself. That is why I will choose the second options. For this reason, it is also essential to save the sheets with the right name. When the excel file is imported, then it is saved as separate CSV files according to the sheet names. For 'load CSV', all the files have to be in the database folder in your computer, under the folder 'import'. So then, in the query, the file name should be written as 'file:///fileName.csv'. All the import queries are shown below.

---

[1] https://github.com/neo4j-examples

[2] https://github.com/neo4j-examples/neo4j-movies-java-bolt

[3] http://neo4j.com/docs/developer-manual/current/cypher/clauses/load-csv/

[4] https://github.com/PHPOffice/PhpSpreadsheet

```
LOAD CSV FROM "file:///SH.csv" AS row FIELDTERMINATOR ';'
WITH row CREATE (:Synset {id:row[0], sh:row[1]});

CREATE CONSTRAINT ON (n:Synset) ASSERT n.id IS UNIQUE;

LOAD CSV FROM "file:///DEF.csv" AS row FIELDTERMINATOR ';'
WITH row MATCH (w:Synset {id:row[0]}) SET w.def = row[1];

USING PERIODIC COMMIT LOAD CSV FROM "file:///REL.csv" AS row FIELDTERMINATOR ';'
WITH row MATCH ( a:Synset{id:row[0]})
MATCH ( b:Synset {id:row[2]})
WITH a, b, row CALL apoc.create.relationship(a, row[1], {}, b) YIELD rel
RETURN row[1];
```

Code snippet 14 - data importing queries

As it can be seen in the code snippet above, to create the relationships, we have to use the help of apoc [21]. That is because cypher does not allow for a dynamic relationship on import. The other possibility would have been a lengthy switch statement. Considering, that there are over 40 different relationships, then that is not a sensible option.

4.2 The project

To explain the project, we will start with the front-end. It consists of HTML, CSS and javascript files and jQuery plugin. The HTML file has buttons for every pattern and a panel where a table of all the results are held. Because the resulting graph takes up too much space, then it is displayed in the background. Both panels containing the buttons and the table of results collapse. Pictures of the program are shown below.

The back-end uses all java. It starts with the file 'util.java.' In that file, all the constants are kept, that is needed to connect to the Neo4j database. The file is shown in the code snippet below.

```
public class Util {
    public static final String DEFAULT_URL = "bolt://neo4j:asdf1234@localhost";
    public static final String WEBAPP_LOCATION = "src/main/webapp/";

    public static int getWebPort() {
        String webPort = System.getenv( name: "PORT");
        if(webPort == null || webPort.isEmpty()) {
            return 8082;
        }
        return Integer.parseInt(webPort);
    }

    public static String getNeo4jUrl() {
        String urlVar = System.getenv( name: "NEO4J_URL_VAR");
        if (urlVar==null) urlVar = "NEO4J_URL";
        String url =  System.getenv(urlVar);
        if(url == null || url.isEmpty()) {
            return DEFAULT_URL;
        }
        return url;
    }
}
```

Code snippet 15 - util.java file contents

There is two main constant kept there. When opening the Neo4J, a username and a password must be assigned. Those are kept as a part of 'DEFAULT_URL.' In this program, the username is 'neo4j, ' and the password is 'abcd1234' and fitted into the 'DEFAULT_URL' in the highlighted place.

Another important constant is the port number. Under the function 'getWebPort()', the number of the port is returned. In this program, it is 8082, but it can be changed at any point if needed.

Next, we will take a look at the file 'patternServer.java.' This file is where the main method is located. This file remained as it was in the example because the main methods function is to start the 'PatternService.java' and 'PatternRoutes.java' classes.

In the front-end, the back-end is called out by an URL path. The path is processed in PatternRoutes class. This class takes the paths with all the variables and calls out a correct method from PatternService. Those methods return a map of results and PatternRoutes makes a JSON object of the map which is returned to the front-end. To put it more simply, PatternRoutes acts as a middleman between front-end and back-end, so that the two would not be in direct contact. That process is also presented in the code snippet below.

```
public void init() {
    get( path: "/patterns/:patternName", (req, res) -> gson.toJson(service.getPatternResults(
            URLDecoder.decode(req.params("patternName")), req.queryParams("type"))));
    get( path: "/graph", (req, res) -> {
        return gson.toJson(service.graph());
    });
}
```

Code snippet 16 - path routing function example

PatternService is the class, where all the data is queried from the database and returned in the right format. In the next chapter, we will delve more deeply into how the patterns are formatted and displayed to the user.

4.3 The patterns

We will take a closer look at how each pattern is implemented. In front-end, it starts with getting results to display them as a table. In the back-end, this part is relatively simple, because most of the queries already produce a list of map elements. Those patterns are all except for all closed subset patterns. With those patterns, the comes as array list from the database and are converted to a JSON list. Each element of the list contains all the variables that are in the return statement in our query. The elements that are returned by using the Collect() statement are returned as arrays. In the front end side, this data is formatted into a table. Understandably, that formatting process is different to each pattern. With the short-cut structure, the chain of nodes is also displayed in the table, but that requires only more nested for loops.

After the table is created, then another function is called out that starts producing the graph. For that process, another query is made to the back-end. The reason for two different queries is that the graph requires more specific result and those are harder to present to a table. The front-end part for graphing was taken from the example, and it uses an SVG element and d3 library to create the nodes and their relationships visually. To do that, the back-end has to produce a map, which contains a list of nodes, and a list of the relationships between the nodes. The relationship list has a target and a source node's location in the nodes array.

What makes the graph visualization special, is the fact, that all nodes across all the results for a pattern are added only once. Meaning, potentially new patterns will be formed. The reason for that is that no node is added twice. For that reason, if any node

exists in two different ring patterns for example, then the two ring will also be connected. An example of that is added as a picture below.

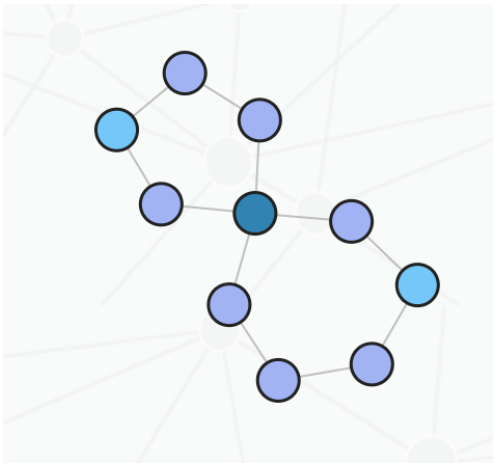

Figure 9 - Example of connected ring patterns

To get the map for the graph, then again the process is different for every pattern. For the pattern that does not fall under closed subset, Then the same query is used again. In general, for every result, each node is turned into a map and added to the nodes list. Then all the relationships are added as well. With the structures that have a list of nodes, then the list is looped, and all nodes are added, and relations tracked down just the same. The function looks a bit long-winded because to get all the nodes and add them nodes and relationship, much variable casting is needed. The reason is that the database returns a list of Map<String, Object> and all other maps have the same variables. That gives us the freedom to have the object as either a new list or a string.

Thus far, we have not mentioned all closed-subset patterns so far. The main reason is that the approach is significantly different for those patterns. We talked in chapter 3 that the results are queried recursively. To make things more convenient, the results are saved in the form as needed for the graphing function. The child and parent nodes are differentiated by a variable, and all relationships are added to a separate list. The only difference is that the maps are added to a list. This method makes the creating of the table harder. That is because several loops are needed. First to get all the parent nodes, and then every child to each parent. In case of the large closed subset and other that need only some of the closed subset, then the checking for the correct pattern is done in front-end as well.

To get the graph for the closed subset is a lot more convenient. The only problem was that each subset was on its list. However, they would not fit on the screen at once, because several of them have over 100 nodes in the subset. For that reason, with every subset, its position in the subset list is also saved. Moreover, when clicking on a result, that said result will appear as a graph. By default, the first result is shown.

The first issue that occurred was the querying time. Some of the queries take around 20 seconds to complete. With the example app, the class BoltCypherExecutor returned the query as an iterator. The problem was that an iterator could be only returned or looped over once. That meant that the only option with the iterator was getting the results twice from the database. However, that takes too much time. The best option was to change BoltCypherExcecutor class and make the query return a list of results and save it to a variable. The changed function is shown in the code snippet below.

```
@Override
public List<Map<String, Object>> query(String query, Map<String, Object> params) {
    try (Session session = driver.session()) {
        List<Map<String, Object>> list = session.run(query, params)
                .list( r -> r.asMap(BoltCypherExecutor::convert));
        return list;
    }
}
```

Code snippet 17 - changed query() function

5 Evaluation

5.1 What was solved and what not

The program created realized almost all the patterns. Those results are presented as a graph and as a table. The data in the table is different with each pattern, but they all contain each node in said pattern. For example, with the ring pattern, the top and bottom nodes are presented along with both of the chains connecting them. And with the closed subnets, all the parents and their corresponding child nodes are displayed. That helps the user get a better understanding of a wordnet and each pattern in particular. There is also a possibility to search for specific results.

The graph representation of a pattern gives the user a possibility to visualize the results. In our solution, all of the different results are presented in one graph, and each node is only added once to that graph. As a result of that, results are connected through shared nodes. Thus there is a possibility to find new patterns. For example, in the ring pattern, a big web of patterns is formed. This example is presented in the figure below, where the top nodes are differentiated by the darkest shade and the bottom node by the lightest shade.

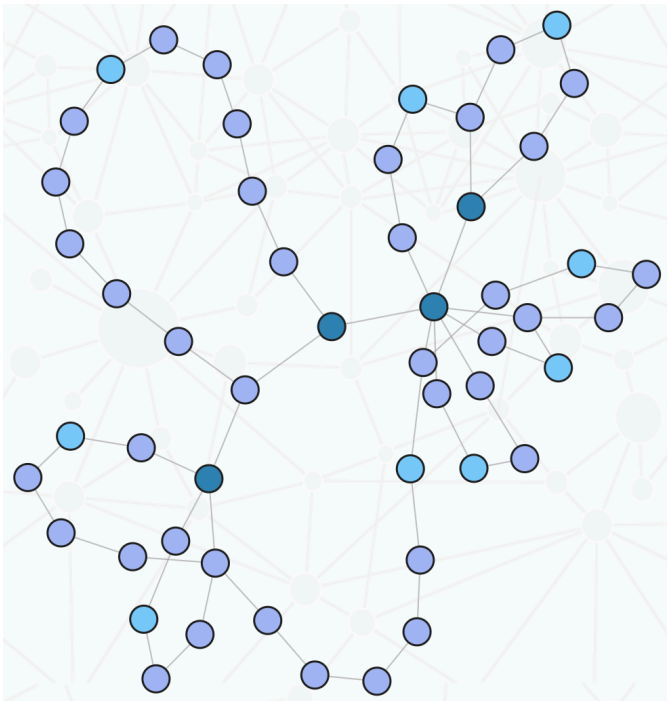

Figure 10 - Cluster of connected result from the ring pattern

In the table below, that are presented three sets of results. The test results are taken from Ahti Lohk's doctoral thesis, and other results are from this project. As it can be seen, then most results are the same in our test results and our results, both using EstWN version 70, are the same. The only difference is in synset with many roots. The reason for that could be having a wrong understanding of the pattern. All in all, it can be seen that there are a few mistakes in the implementation of different patterns, but the majority of the patterns implemented are precise. It can also be seen, that the number of results did not decrease in version 71. It could be, that different modification was made that did not include hyponym relationships.

Version	Multiple inheritance cases	Short-cuts	Rings	Synset with many roots	Heart-shaped substructure	Dense component	Largest closed subset
Test results EstWN v70	51	7	21	70	0	3	123x4
EstWN v70	51	7	21	33	0	3	123x4
EstWN v71	51	7	21	33	0	3	123x4

Table 1 - Results

5.2 Limitations

There were also several limitations to the solution. One of the limitations is the fact that in this solution 2 of the test patterns are not implemented. Those patterns are the substructure that considers the content of synsets and connected root synsets. Both of these patterns were too difficult to achieve at this version of the program but will be implemented in the next release.

The other major limitation was the fact that the data cannot be imported by uploading an excel file. This was because it was challenging to integrate this to the example product. This is also the main priority for the next release.

The last two limitations have to do with how our program presents the results. Especially as a graph. Our graph presentation is not hierarchical as it should. Our code produces a cluster of results, but the heart pattern turned into a ring for that reason. To solve that, the graphing function would have to be changed significantly. The other issue with the graphs is that there are no names on the nodes. The name of the synset appears on hovering after a few seconds. That makes reading the graph more difficult because there is a need to hover over every node in order to get the idea of the graph. This solution was chosen because some of the names are too long and they would make the node significantly longer. Since our solution presents all the results in one graph, then it already takes up almost all the space on the screen. Thus, making the nodes larger would mean, that only some of the results would be shown at once. At this moment we opted to show all the results at once and have the names show only on hover.

6 Conclusion

In this work, we focused on wordnet type semantic hierarchies and graph databases. There are groups of researchers working on different wordnets all over the world. Several aspects go into creating wordnet and during that process, many different mistakes might occur. All wordnets need to be validated, to correct said mistakes. The problem is that there is no universal tool for validating wordnet and most research teams do not have the resources, both human and monetary, to create a tool on their own. There were several patterns presented in Ahti Lohk's doctoral thesis that are the universal pattern that can be used to detect errors in wordnet. This work aims to create a universal program, which can detect errors in all the wordnets with the help of said patterns.

To create this program, graph databases were used. That is because graph databases are made for data with many relationships and thus are an excellent solution for wordnet. In the created program, a user can choose the desired pattern, and the results will be presented as a table and also a connected graph.

Most of the test patterns were realized. Only the substructure that considers the contents of a synset and connected root synsets patterns were not realized. Both of them were too difficult to realize. One reason was the pattern themselves, but the other reason was, that both wordnets and graph databases were unknown to me before this work. Thus, much time was put into understanding both concepts, and it took time away from realizing the patterns. There were also two more shortcomings – the importing of synsets was not realized, and the visual graph presented to the user was not hierarchical, as it should be. Both of these shortcomings are planned to be implemented in the next version of this program.

This program did, however, realize all the other patterns, and because in the visual representation all the nodes are represented only once, then there is a possibility to discover new patterns. The aim of this project was accomplished, and a universal tool was created that can be used to verify all wordnets.

References

1. Larsen, Peder Olesen, and Markus Von Ins. “The Rate of Growth in Scientific Publication and the Decline in Coverage Provided by Science Citation Index.” *Scientometrics*, vol. 84, no. 3, Oct. 2010, pp. 575–603., doi:10.1007/s11192-010-0202-z.

2. Lohk, Ahti. “A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-Type Dictionaries.” Thesis. Tallinn University of Technology, 2015.

3. *Estonian Wordnet*, www.cl.ut.ee/ressursid/teksaurus/index.php#sec-3.

4. Shu, L.h. “A Natural-Language Approach to Biomimetic Design.” *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 24, no. 04, 2010, pp. 507–519., doi:10.1017/s0890060410000363.

5. Collins, Allan M., and M. Ross Quillian. “Retrieval Time from Semantic Memory.” *Journal of Verbal Learning and Verbal Behavior*, vol. 8, no. 2, 1969, pp. 240–247., doi:10.1016/s0022-5371(69)80069-1.

6. Prószéky, Gábor, and Márton Miháltz. "Automatism and User Interaction: Building a Hungarian WordNet." *LREC*. 2002.

7. Sagot, Benoît, and Darja Fišer. "Extending wordnets by learning from multiple resources." *LTC'11: 5th Language and Technology Conference*. 2011.

8. Lindén, Krister, and Jyrki Niemi. “Is It Possible to Create a Very Large Wordnet in 100 Days? An Evaluation.” *Language Resources and Evaluation*, vol. 48, no. 2, Nov. 2013, pp. 191–201., doi:10.1007/s10579-013-9245-0.

9. Fišer, Darja, and Benoît Sagot. “Combining Multiple Resources ToÂ BuildÂ ReliableÂ Wordnets.” *Text, Speech and Dialogue Lecture Notes in Computer Science*, pp. 61–68., doi:10.1007/978-3-540-87391-4_10.

10. Sagot, Benoît, and Darja Fišer. "Automatic extension of WOLF." *GWC2012-6th International Global Wordnet Conference*. 2012.

11. Měchura, Michal Boleslav. "What WordNet does not know about selectional preferences." (2010).

12. Lohk, Ahti, Heili Orav, and Leo Vohandu. "Some structural tests for WordNet with results." *Proceedings of the Seventh Global Wordnet Conference*. 2014.

13. Verdezoto, Nervo, and Laure Vieu. "Towards semi-automatic methods for improving WordNet." *Proceedings of the Ninth International Conference on Computational Semantics*. Association for Computational Linguistics, 2011.

14. Richens, Tom. "Anomalies in the WordNet verb hierarchy." *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*. Association for Computational Linguistics, 2008.

15. Lohk, Ahti, et al. "New Test Patterns to Check the Hierarchical Structure of Wordnets." *International Conference on Information and Software Technologies*. Springer, Cham, 2014.

16. Sagot, Benoît, and Darja Fišer. "Cleaning noisy wordnets." *LREC 2012-Eighth International Conference on Language Resources and Evaluation*. 2012.

17. Carlson, Stephan C. “Graph Theory.” Encyclopædia Britannica, Encyclopædia Britannica, Inc., 19 May 2017, www.britannica.com/topic/graph-theory.

18. Robinson, Ian, et al. Graph Databases Ian Robinson; Jim Webber; Emil Eifrem. O'Reilly & Associates, 2015.

19. *3.3.1. MATCH - 3.3. Clauses*, neo4j.com/docs/developer-manual/current/cypher/clauses/match/.

20. *3.4. Functions - Chapter 3. Cypher*, neo4j.com/docs/developer-manual/current/cypher/functions/.

21. Needham, Mark. “Create Dynamic Relationships With APOC - DZone Database.” Dzone.com, 2 Nov. 2016, dzone.com/articles/neo4j-create-dynamic-relationship-type.