Master's Thesis

Fuzzy Matching and Merging of Family Trees using a Graph Database

Hampus Lundberg

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, March 2015.

Fuzzy Matching and Merging of Family Trees using a Graph Database

Hampus Lundberg

Department of Electrical and Information Technology
Lund University

Advisor: Anders Ardö

Tuesday 10th March, 2015

Printed in Sweden
E-huset, Lund, 2015

Abstract

Association for computer aided genealogy research of Sweden(DIS) is investigating the possibility of a nationwide genealogical database (RGD) of Sweden's historical population. The finished database should contain basic information about individuals' names, birth, marriage, death, and individuals' relationships such as father, mother, husband/wife and children. This will make up a kind of graph over the ancestry of Sweden historical population were persons are connected to each other according to ancestry. The idea is that different genealogists should add already finished pedigrees (family trees). The problem is that the same pedigree can be inserted by two different genealogist. RGD has to find these duplicates and merge them. In this thesis a test application for RGD is made in small scale using the graph database Neo4j and the document search tool Lucene. The focus is on finding and merging duplicated pedigrees. The test application made was able to upload files containing multiple pedigrees and merge them into a graph that was stored in Neo4j. This merging process had a linear time complexity in relation to how many families that were merged. A family in this thesis means family unit containing father, mother and children. Storing the biggest available file (20000 persons and 6500 families) in an empty database and then inserting the second biggest (10000 persons and 3000 families) took about 2 min and 30 seconds. The result was about 1200 family merges. This was done on the laptop Lenovo N500. Lenovo N500 has a Dual CPU T3200(2GHz) and 3GB RAM. The accuracy of the algorithm was compared to a previously made application. Both applications were tested on the same data-set. The result from the tests was the overlap of families merged by both applications. Then precision and recall was calculated for the test application considering the previous application gave all the correct family merges. There were two precision and recall scores estimated. The better one of those two gave the precision 97% and the recall 81%. The F-score for the system was 88%.

Acknowledgements

First of all I would like to thank my supervisor docent Anders Ardö for his support throughout the process of making this thesis. The thesis would not have existed without him. I also like to thank my examiner Dr. Mats Cedervall for his corrections, the EIT department at LTH for providing a good work environment and my family and friends for their emotional support during my master thesis studies.

Table of Contents

Chapter 1

Introduction

The field of genealogical research has grown in popularity during the last decades. One major reason is that it has become simpler. Today many sources of ancestry data have been and is being digitalized. A common example is the old so called church books in Sweden. The information that has been stored varies but at least basic information about individuals such as name, date and place of birth, date and place of death and date and place of marriage are available. Basic family relationship such as parent, spouse and child has also been stored. There are today computer programs that can find ancestors one or several steps back. The specialities of the programs differs from drawing good looking pedigrees to advanced search capabilities. There are many different groups interested in genealogical research, all from large volunteering organisations like the Latter Day Saints church[10] to commercial websites like ancestry.com[20]. Their files are available online and can contain not only basic personal data but also parts of a family tree form earlier research. Sometimes also information about people's living conditions are stored.

There are however problems with collecting information about ancestry from different sources. The main one is that different sources may contain duplicate and conflicting information. For instance the name and place of birth is the same for two persons stored in two different files but their date of birth is different. So now the question is are the two persons actually the same person with a mistake or two separate persons. For making a unified database containing persons from several sources some kind of matching program would be necessary. The purpose with such a program would be to automatically merge duplicated persons.

A trend in the computer world is the so called noSQL databases. One type is a graph database. In these information is stored in a graph and they specializes in storing and querying highly connected data. This makes graph database interesting for the field of genealogical research where individuals are connected to each other through their family relationships. In larger databases it is not enough to validate that two person suspected to be the same are the same on their basic information such as names, dates and places. It is also necessary to see that the persons they are related to are the same. In a graph database traversal between related family members is simple which facilitates that two groups of persons easily can be traversed to check for similarity.

The oldest association in the world for computer aided ancestry research is called DIS (Datorhjälp i släktforskning) [7]. The association is currently exploring the possibility of building a Nationwide Genealogical Database (RGD) over Sweden's historical population. The idea is that data should come from different genealogists. The data should be in form of files containing basic information about individuals (name, birth, death) and their family relationships (father, mother, marriage, children). The problem is as mentioned before that different files may contain duplicate and conflicting information. The focus in this thesis is the development of a java application for detecting duplicated families. Families here means a family unit of husband, wife and children. The developed application uses the graph database neo4j[14] and the text based search tool Lucene[17]. The application was tested on small sets of families up to about 6000. The purpose of the work was to determine if this approach could be used to build a RGD. The duplicate families are detected and merged during insertion. This makes the insertion operation the most important work of the thesis.

The thesis tries to answer the following problems.

- How shall information be saved within Neo4j?
- How shall basic operations such as insertion, removal and search be implemented?
- What is the time complexity for the insertion operations?
- How accurate is the insertion method at detecting duplicates between the database and the insertion file?
- How shall information about duplicate families (conflicts, two possibilities) be represented to the user?

1.1 Similar work

Previously Neo4j's suitability for storing and traversing genealogical data has been investigated [3] [18]. In [3] it is concluded that Neo4j is superior to MariaDB(a relational database) regarding traversal of genealogical data. A study of different graph database was done in [18] as to find out which graph database was most suitable for storing and querying genealogical data. The study concluded that Neo4j was among the top two most suitable databases. There are other applications that tries to solve the problem of duplicate individuals[22]. For example is there a Danish enthusiast that has made his own matching program[16]. Currently (January 2015) he has about 6.2 million individuals in his database. His programs among other things can be used to find other researchers researching the same individuals [22]. There are also a large number of commercial products [20][9][11][13]. These products are mostly used as an aid for enthusiasts who want to know more about their ancestry. What all these sites have in common is that they enable searching through large records of genealogical data. They often have slightly different focus, some focus on cooperation between users by connecting user researching the same individuals while others focus on advanced searching capabilities. How the information actually is stored by the sites is unclear since companies protect their ideas. So whether or not commercial companies solve the same problem as in this thesis is hard to know. What is certain is that commercial products have search capabilities. Search capabilities are also needed when finding persons to merge.

The supervisor of this thesis [5] has done a similar application as the one made in this work. The difference is that in that work a MySQL database was used instead of a graph database and that the insertion and merging algorithm was different. The accuracy of the application was tested by comparing it's result with the result from the application made by [5].

1.2 Disposition

The thesis begins with the theory used for development of the test application. How the test application was implemented is explained in chapter 3. In chapter 4 the motivation for the different tests is described. Chapter 5 contains the result and discussion of the tests. Final conclusions and possible future developments are discussed in chapter 6.

Chapter 2

Background Theory

A number of different tools and theories comes together to develop a test application for RGD. The basic tools are as mentioned in the introduction Neo4j and Lucene, they are described in section 2.6 and 2.7. In order to understand how these tools work some graph theory and some information retrieval theory need to be explained. The difficulty surrounding graph and sub-graph isomorphism is explained to understand the difficulty of the matching problem. Graph isomorphism is the the problem of determining if two graphs are the same. Sub-graph isomorphism is the problem to determine if small graph is the same as a sub-graph in a larger graph. The basic computer subject of time complexity is also explained to give an audience outside the field of computer science a chance to understand the content of the thesis.

2.1 Time complexity

In computer science time complexity is an important subject. Time complexity is a measure about how long it takes to do an operation on a certain number of elements. Elements here can be all kinds of data structures such as numbers and text strings. An example could be how long time it would take to find the number of sevens in a sequence containing n numbers. Since the whole sequence needs to be looked through and it takes about the same length of time to look at every number, the execution time can be expressed as $t = k \cdot n$ where k is a constant and t stands for the execution time. When measuring time complexity exact execution time is not relevant since it varies for different computers and between different executions. Therefore the time complexity measurement is simplified from $t = k \cdot n$ to $tc = n$ where tc stands for time complexity. When talking about time complexity the expression order n is used rather than saying that the time complexity is n. In writing, order n is expressed as $O(n)$. There are a couple of common time complexities for different algorithms. These are $O(1)$ meaning something can be done in constant time, $O(n)$ the execution time is proportional to the number of elements and $O(log(n))$ which means that there is a logarithmic dependence between the execution time and the number of elements. When the time complexity is $O(n \cdot log(n))$ it is worse than $O(n)$ but still within acceptable regions for most applications.[1]

In case of making a test application for RGD time complexity is of great importance. When data gets added to the test application it will naturally take longer time to search for potential merges, since there are more individuals to search through in the database. Even if execution time is low on a small database bad time complexity will raise execution time quickly when the database grows.

2.2 Precision, Recall and F-score

When evaluating information retrieval systems the parameters precision, recall and F-score are often used. The parameter precision is a measurement regarding how many answers given to a query were correct. In this work this correspond to how many of the found matching families were correct. Recall is how many of the possible answers to a query were found. For the application this corresponds to how many of the families that were the same were found and merged. F-score is a combination of both precision and recall. The F-score is given in 2.1. The measurement F-score is an important performance measurement because it is easy to make a system that has either good precision or recall. In order to get high precision only matches that have an absolute certainty are returned. Recall is even easier when it is possible to return the entire content of the database which will result in that all possible match are found (and a lot of wrong ones). The hard goal is to have a high precision and recall which corresponds to a high F-score. [19]

$$Fscore = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.1}$$

2.3 Property Graph

A property graph consists of nodes and relationships linked together. The kind of properties depend on what the graph represents. An example is given in figure 2.1. The circles are the nodes and the arrows the relationships. The text represent the properties. In this example the nodes are assigned three different kinds of properties. These are first name, date and place. In Neo4j information is stored by the same principle as in the example. However, in this work nodes containing more information were used. The idea is that all family history will end up in one big property graph.[14, p.26].

2.4 Inexact sub-graph Matching

The problem of finding matching individuals can be thought of as finding a sub-graph in a large property graph. Many related individuals need to be considered similar to determine a match. Inexact sub-graph matching is considered an NP-complete problem in many cases[23], meaning that it has no know standard solution with a time complexity that is in the order of a polynomial [2]. This make the standard solutions unusable in our case. Therefore it is necessary to find a tailored solution.

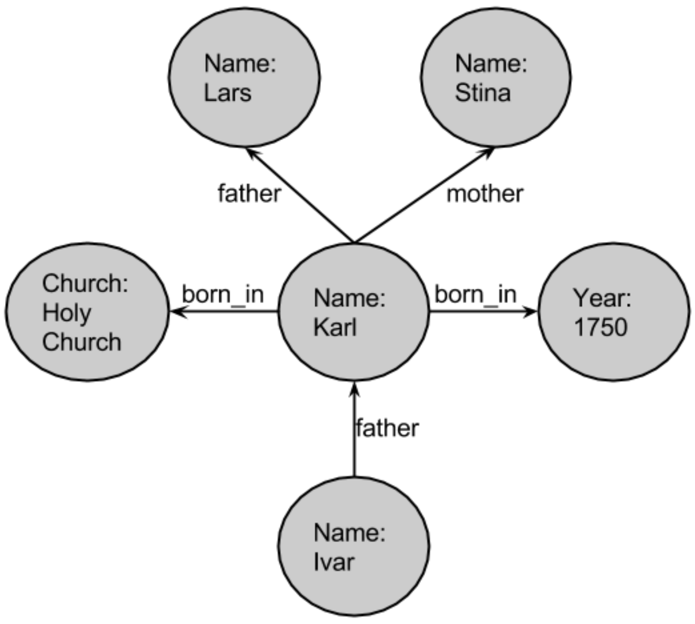


Figure 2.1: An example of what a property graph in Neo4j looks like.

2.5 JSON

JSON is a certain text format used for sending data between applications. The format is good for writing content contained in different data structures such as an object or an array into a text file. In a json file an object contains many different type/value pairs. A person object could for example have a type/value pair, birth date/1989-10-04. An array could then consist of many persons.[15]

2.6 Neo4j

Neo4j is a graph database where information is stored in a property graph. It also supports directed relationships. Neo4j is the most used graph database in the world today. It is open source and can be used for free. In Neo4j properties are stored as type/value pairs. The user can define both the property types and values. Neo4j has three application programmable interfaces (APIs). The most used API is Neo4j's query language Cyper. With Cyper its possible to add and remove nodes and relationships and do queries in Neo4j. The Cyper queries in Neo4j has three parts namely a start statement, a matching pattern and a return statement. The start statement contains which node or nodes the query should start from. The start nodes are defined by what property they contain. The matching pattern describes paths containing nodes and relationships with the specified properties that the query has to follow. The return statement contains which nodes or relationships that should be returned. The query works by first finding the start nodes through lookup and then trying to traverse the property graph according to the matching pattern. In case of success the nodes specified in the return statement are returned. There can be several results from one query. In a genealogical graph stored in Neo4j an example could be the graph shown 2.1 and a query for that graph could be to retrieve all individuals born in Holy church in the year 1750. The start node can be Holy church. The query then tries to go from the Holy church node through a person node to the year node 1750. For all cases when this is possible the individual node is returned. In this case the Karl node would have been returned. Cyper is not used in the test application for RGD. However, how quires are done in Cyper gave inspiration for how to find duplicated individuals with the test application developed.[14]

The core and the traversal API was used when developing the application. They are more closely connected to the internal workings of Neo4j and are used when it is important with low execution time and making specific queries as in this case. The traversal interface lets one use a traverser that will traverse nodes according to a chosen pattern from a chosen starting node. It is very similar to how Cyper works. The core API was mostly used in writing the application. Both the traverser API and Cyper is built on the core API so it gives the most freedom for designing applications with Neo4j. With the core API it is possible to get access to the primitive node and relationship data types. It is also possible to get the relationships connected to a specific node and the nodes connected to a specific relationship. This makes it possible to make non standard quires and operation as was desired in this case.[14]

In order to understand the time complexity for operations done by the test application, it is important to know the complexity for some standard operation in the core API. Getting a node in Neo4j by using it's id has the time complexity $O(1)$. This is also the case for adding nodes and relationships. In the case that the id to a node or a relationship is available it is possible to delete it in $O(1)$ time. Neo4j uses something called adjacency lookup which means that from any node it is possible to get all of it's neighbouring nodes and relationships in $O(1)$ time. This is made possible by Neo4j's internal storage having references in every node and relationship pointing to it's connected nodes and relationships. Finding nodes with a specific property has time complexity $O(log(n))$, which is implemented through a lookup table, the same as in traditional relational databases.[14]

2.7 Lucene

Lucene is a search tool for searching through documents. It uses something called reverse indexing where first documents are indexed based on their content. Then the index is used to find a document containing the text being searched for[17]. This is primly done through using the Vector Space Model (VSM)[4]. In VSM documents are represented as multidimensional vectors, where each dimension represent a separate term(word) in the document. The length and the direction of a document's vector is determined by it's term frequencies(the number of times a term appears in a document). Search is done by comparing the query text's vector to the stored document vectors. This is done using a form of dot product.[12]

To be able to quickly search through documents, only comparing documents relevant to a query is important. This is achieved by having a well structured index. The index is structured so that for every term there is a connected list with all the documents that contains the term. In that list there is also for each document the term frequency for the term in that document. These lists are called postings lists. Together with each term there is also a number for how many documents they appear in for the entire index.[17]

When doing a query, first the query text is parsed into it's containing terms. Then the postings list for each term in the query is iterated through. During iteration the term frequency times a specific weight called term frequency–inverse document frequency weight(tf-idf weight) is added to the score for each document in the posting lists. The tf-idf weights are ratios between how many times a word occur in a document contra how many documents it occurs in. Various ways to calculate the exact tf-idf weights exist, most of them include dividing the term frequency for the document with the number of documents the term occurs in. The purpose of the tf-idf weights is to boost similarity score for rare terms in the index.[12] The document similarity score for a query becomes the sum of all term frequencies times the terms' tf-idf weights. The higher this score is the higher the similarity is considered to be between the document and the query[12]. The advantage of this method is that only the documents containing the relevant terms are considered when searching[6].

An example over how the postings list are generated and used could be the following. Consider two document being stored. The first document containing the text "Hello cat" and the second "Hello world". This would generate the following postings list.

Postings lists	
Term	List
Hello	1, 2
cat	1
world	2

Then when searching for example the text "Hello world". First the postings list for "Hello" is iterated through and document 1 and 2 score will be add with one times the tf-idf weight of "Hello". The weight is taken times one because "Hello" only occur once in both documents. For this simple illustration let say that the tf-idf weight is 0.5 for "Hello" since it occurs in both documents and that "cat" and "world" has the weight 1 because they are only in one document each. So then both document 1 and 2 has the score 0.5. Finally the "world" postings list is iterated through which gives the final score of 1.5 for document 2 and 0.5 for document 1. So in this case document 2 will be the top result of the query.

Often when doing a query it's of interest to get the top k (chosen number) documents. In order to do this the method explained above needs to be improved. This is done by having a queue holding the $k + 1$ most relevant documents. The queue is ordered so that the document considered most relevant for the query is put in the back. Every time a new document is evaluated it is inserted into the queue. The position is decided by the similarity score of the document. Then the first element of the queue is withdrawn. This makes it so that only the current top k documents are kept in the queue. The total time complexity for this algorithm becomes $O(p \cdot log(k))$ where p is the number of postings in the postings lists for the terms of the query.[12]

Even though this is a decent time complexity it can be improved. This has been done using approximative methods. These approximative methods finds k 'good' documents instead of the top k documents. The approaches is based on that often the most rare terms are the most important for a query. This is because rare terms tf-idf weights are higher. So first the query terms are ranked according to their weights. Then the posting lists are iterated according to that ranking order, adding scores to the documents containing the most important terms first. When adding scores to a document an estimation is made whether it is possible to stop the search. The search process is stopped when a good enough match has been found. This technique has two advantages. First that not all postings list are iterated. Second that the shortest list are searched and longer ones are the ones that are often left out.[6]

Chapter 3

Implementation

3.1 System overview

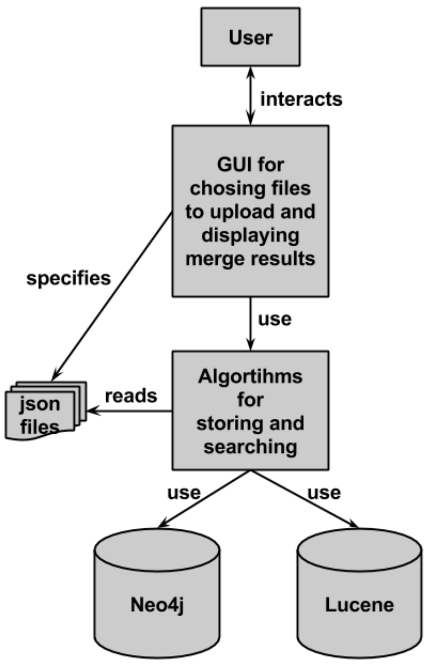

User
interacts
GUI for chosing files to upload and displaying merge results
specifies
use
Algortihms for storing and searching
reads
json files
use
use
Neo4j
Lucene

Figure 3.1: A system overview of the entire application

The system overview is shown in figure 3.1. The Graphical User Interface (GUI) is used to select the json file for insertion and also for showing merging result. The path for the selected json file is sent to the algorithm for insertion into Neo4j and Lucene. Almost the entire work for this thesis has been on developing the insertion algorithm. There has also been some development on the GUI but further work to finish this is required.

3.2 Tools and Devices

The tools used in the development of the system includes Neo4j community 1.8.3 version in it's embedded form. Neo4j uses Lucene internally [21]. This was Lucene 3.5.0. All the code was written in java. The Integrated Development Environment (IDE) used was Eclipse kepler [8]. In order to read json files the json simple library from [15] was used. Both Neo4j and the json simple library was included as jar files into the program by extending the java build path in Eclipse.

3.3 The Content of the Genealogical Data Files

In order to understand certain aspects of the algorithm it is important to know the content of the inserted data files. The insertion files are json files containing two json arrays. One for family json objects and one with person json objects. Every person and every family has a fixed set of properties. Each of the properties can be assigned a value or not. In the case a property has not been assigned a value it has the default value null. The following two tables show an example.

Person properties	
Property name	Property value
Gedcomid	14-1
Id	1
Birth family id	46
Name	Sven Svensson
Normalized first name	4045 6942
Normalized last name	144270
Sex	M
Normalized birth date	19500101
Normalized death date	null
Birth Congregation	Lund Domkyrko (M)
Normalized birth congregation	398
Death Congregation	null
Normalized death congregation	null

Family properties	
Property name	Property value
Gedcomid	14-256
Id	49
Normalized Marriage Congregation	null
Normalized Marriage Date	null
Husband Id	209
Wife Id	208
Marriage Congregation	null

Normalized here means that the name of a church or person has been replaced with a number. This is in order to make comparison easy between churches and persons since there might be slight variation in spelling. There are two types of ids for persons and families in the files. These are used for giving unique identification. The two types differ a bit. One is for connecting persons and families within a file. This is the one called Id and it is only a local identifier within a file. In other words persons and families can have the same Id between files but not within the same file. Linkage between persons and families are done using this id and the Husband Ids, Wife Ids and Birth Family Ids. Where the Husband Ids, Wife Ids and Birth Family Ids are used as pointers to indicate how persons and families are connected. The second type of id is called Gedcomid. The reason why there is also a Gedcomid is that the json files were derived from GEDCOM files. GEDCOM is a file type that stands for Genealogical Data Communication. This is the most used file type for sending genealogical data. In this work the Gedcomids were unique for all families and persons inserted during the test procedure, which made them suitable for identifying families during this procedure.

3.4 Algorithm

The main work carried out during this thesis is the algorithm explained in this section. Firstly some limitations on the algorithm will be given. Secondly how data was stored in Neo4j will be described. Then a basic overview over how the algorithm works will be explained. Certain key operations will be explained in more detail together with motivation for approach.

3.4.1 Limitations

The algorithm developed only finds duplicated families between files. Usually there are also some duplicated families within the same file. These will not be looked for or merged by the algorithm. An other problem with the algorithm is that it does not handle simultaneous updates. After insertion there will usually be some matches that need to be looked over by the user. The problem is that if a new file get inserted while the matches form a previously inserted file is looked over changes to the matches being looked over can be made. When this is the case the user needs to get notified of what has changed for the matches that are being looked over. Some kind of mechanism for doing this is necessary in a real RGD application, which is ignored here.

3.4.2 How the Genealogical Data is Stored in Neo4j

There are two types of nodes stored in Neo4j namely family and person nodes. Both family and person nodes have a set of properties. These can be seen in figure 3.2 and figure 3.3. The properties are the same in the json input files.

In Neo4j person and family nodes are not separate entities but are part of property graphs as in figure 3.4.

There are alternative ways for storing properties. Commonly in Neo4j it is not recommended to add many properties to the same node but instead storing different properties at different nodes [14, pp. 69-71]. This is due to how queries usually are done in Neo4j. In Neo4j queries are traditionally based on searching

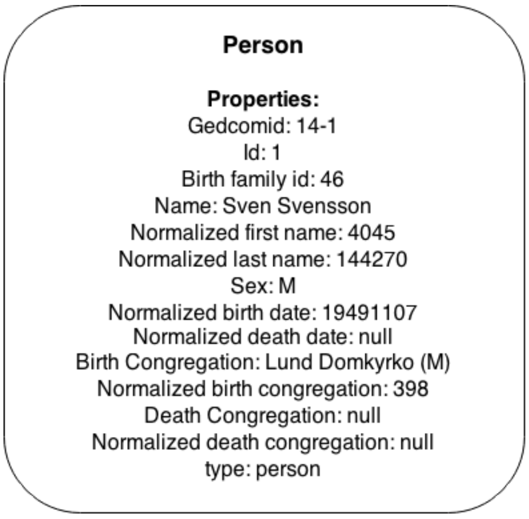

Person
Properties:
Gedcomid: 14-1
Id: 1
Birth family id: 46
Name: Sven Svensson
Normalized first name: 4045
Normalized last name: 144270
Sex: M
Normalized birth date: 19491107
Normalized death date: null
Birth Congregation: Lund Domkyrko (M)
Normalized birth congregation: 398
Death Congregation: null
Normalized death congregation: null
type: person

Figure 3.2: An example of a person node in Neo4j.

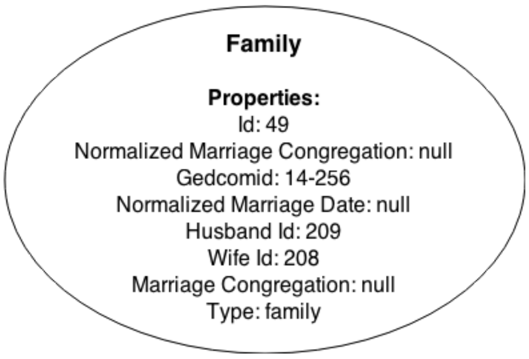

Family
Properties:
Id: 49
Normalized Marriage Congregation: null
Gedcomid: 14-256
Normalized Marriage Date: null
Husband Id: 209
Wife Id: 208
Marriage Congregation: null
Type: family

Figure 3.3: An example of an family node in Neo4j.

for something that relates to something else. A good example query could for example give all persons born in 1915 that were nationally registered in the Parish of Domkyrkan in Lund. In this type of query churches, years and persons would be represented as nodes and there would be relationship between the person node and the year and church nodes. However the way chosen for storing properties in this work disable this capability. The motivation was primly to save time during insertion. This is because it goes faster placing all properties on every person and family node instead of having a node for every specific property and then inserting a relationships from the property nodes to their respective person or family nodes. The reason for this is that a lot of searches is needed to find the property nodes for the persons and families.

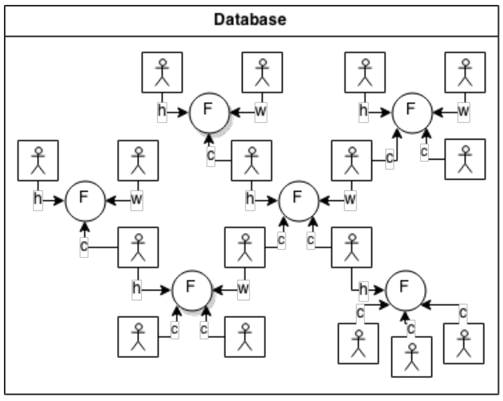


Figure 3.4: Picture showing how the family and person nodes are connected in the database. No properties are shown to make the graph easier to read and understand. The person nodes are the square nodes with a man inside and the family nodes are the round nodes with an F inside. The relationships indicate what role a person has in a family. The roles are husband, wife and child which is indicated by h,w and c.

3.4.3 Basic Overview

For every family that gets inserted into the database there has to be a query to see if there are any similar(duplicate) families already in the database. The big issue in this thesis is how this query can be done effectively. The basic inspiration for the query used is taken from Neo4j. When doing a query in Neo4j a start node is first searched by index lookup and then traversal takes place to see if the rest of the query graph matches a sub-graph in Neo4j[14, pp. 29-30]. The algorithm is structured in the same way. When the algorithm is executed potential family matches are first searched for with Lucene and then traversal is carried out in Neo4j for validation and merging. This is done by first searching for every family in Neo4j using Lucene. This search is enabled by having every family in Neo4j stored in Lucene. More specifically, what is stored in Lucene are documents containing most of the property values from different family nodes together with their family members. These documents are indexed which makes it possible to search for a family in Lucene. In order to connect a family in Lucene to it's corresponding family node in Neo4j a id pointing to the corresponding node in Neo4j is also stored in Lucene. After Lucene has find a potential families to merges with the families being inserted an other evaluation process is gone through. This evaluation process is an extra check whether two families should be merged (see section 3.4.4). The families that pass this evaluation are then validated with Neo4j. Finally the families that pass the validation are merged(see section 3.4.5).

As can be seen from the activity diagram in figure 3.5 there are eight steps in the algorithm. Matching and merging of families are the main subject for this thesis and this is carried out in step 5 and step 6. All steps will now be described to give an overview and then the merging and matching will be described in greater

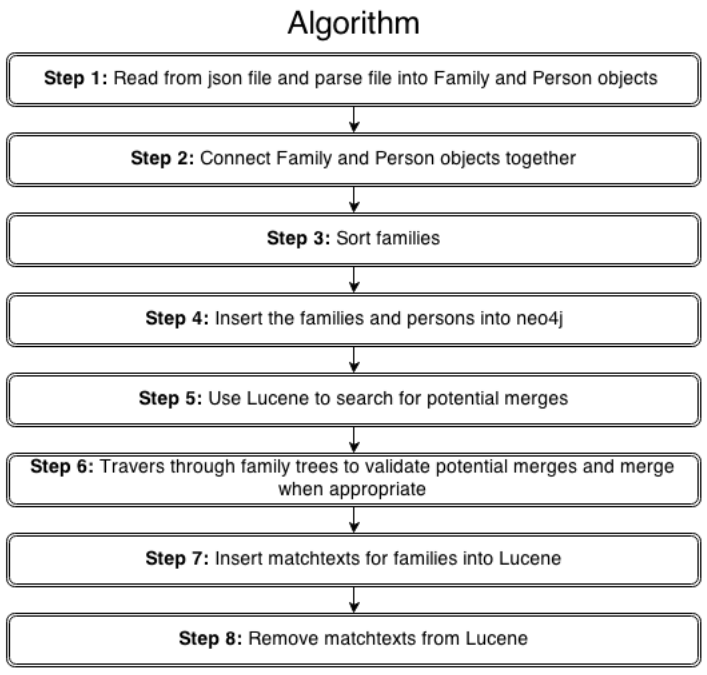

Algorithm
Step 1: Read from json file and parse file into Family and Person objects
Step 2: Connect Family and Person objects together
Step 3: Sort families
Step 4: Insert the families and persons into neo4j
Step 5: Use Lucene to search for potential merges
Step 6: Travers through family trees to validate potential merges and merge when appropriate
Step 7: Insert matchtexts for families into Lucene
Step 8: Remove matchtexts from Lucene

Figure 3.5: Activity diagram showing the different steps that the insertion algorithm goes through when inserting families from a input file.

detail.
Step 1: The input file is being parsed and stored in lists of family and person Java objects. These objects are temporary containers for storing the properties for family and person nodes.
Step 2: The family and person objects are connected. This is done by inserting the person objects into the family objects they belong too. These connections will later be used when inserting the persons and families into Neo4j. This is enabled by pointers within the json files which tell which persons and families are connected.

The families and persons have an unique family or person id in the json files. Every family also has an id pointing to the husband or wife belonging to the family and every person has an id pointing to the family within the person was born. In case that a family's husband or wife is unknown the husband or wife id has the value null and the same is true if the birth family for a person is unknown. For easier insertion into the database family objects can store person objects as either husband wife or child. The second step of the algorithm does the mapping so that every family object gets filled with the husband, wife or child person objects that belongs to it.
Step 3: All the families are sorted according to marriage date of their husbands and wifes. Families that have been married more recently are placed first in the

list. In the case the marriage date is missing the family is put last in the list. This is done to make step 6 work better. In step 6 family trees are being compared and merged. The idea is that it is easier to compare large trees because comparing two large trees means more family comparisons are made which makes the matching result more certain. Comparisons are made by traversing two trees and comparing one pair of families at a time, see section 3.4.5 for more information. A family tree's root lies in the latest married family of the tree. So in order to traverse two large trees instead of traversing and merging smaller sub-trees of the larger trees it is important to start traversal with the pair of families that has the most recent marriage dates. Ordering the list in this step leads to that the traversal start at the right place in step 6.
Step 4: Inserting all the persons and families from the file into Neo4j is done in two steps. First all persons are inserted. During the insertion of persons the persons objects ids are changed to match the person node ids in the database. This makes it quick to insert the families when the person family member nodes can be find directly with their id. Since the time complexity for finding a node using it's id is O(1) in Neo4j.
Step 5: For every inserted family a similar family in the database is searched for using Lucene. In order to use Lucene a matchtext for every family is generated. The matchtext is a text string containing property values from all family member nodes and the family node itself. When searching Lucene compares the matchtext with it's previously inserted matchtexts. In Lucene matchtext and node ids for a family are stored as pairs. Lucene retrieve the five family ids for the five families with the most similar matchtext compared to the family matchtext searched for. The number of five families was chosen to be retrieved instead of one because it is possible that the family Lucene considered most similar to the query family is not the most similar one in reality. This potential error is eliminated by retrieving more families and checking all of them. On the other hand retrieving to many families slows down execution time, therefore five families were considered enough. The ids retrived by Lucene are used to find the corresponding family nodes in Neo4j. The five found families together with the query family make up five pairs (the query family is in every pair together with one of the found families). The pairs are evaluated using a family comparison method described in section 3.4.4. The method gives five similarity scores for every pair. When the similarity score for a pair exceed the threshold value of 0.5 and has higher similarity score then the other pairs the pair is stored in a Java structure called a map. The pairs stored in the map are considered potential merges.

It is hard to set the number for how many families Lucene should search for and the threshold value for considering a family match. Since what are appropriate values for these variables are dependent on the accuracy and completeness of the data intended to be inserted. One way to do it is to test a couple of different values for the variables to see which worked best. The disadvantage of this is that it is hard to tell if the algorithm is improved or just becoming better adapted to the tested data-set. An other way to go about it is to try to change the values to get optimal results on part of the data-set and then trying them on the entire data-set. If the optimized values preforms better on the entire data set the variables are changed. In this way the disadvantage of changing the values to only

preform better on a specific data-set is avoided. There was however not that much data available, so instead a small test just to check if the result remained similar when changing the variables slightly was done. The results remained almost completely the same. Therefore it was concluded that it was not worth putting time on optimizing the variables and the initial values of 5 and 0.5 was kept.
Step 6: The potential merges found have to go through a more narrow validation before merging can take place. This is because an inconsistency would arise if two families where merged without merging the birth families of the husband and wife. This inconsistency would be that the husband and wife would be children in two families which is impossible (adoption is not considered a possibility). Therefore when merging two families also the families where the husband and wife was born has to be merged. This process repeats itself as long as the families being merged has a husband or wife that is connected to his/her birth family. This leads to that entire family trees has to be traversed and validated before merging can take place. The same does not apply for the children of the merged family. This is because remarriage is a possibility which makes it possible for a child to later become husband or wife in two families. How the traversal is done is more thoroughly explained in section 3.4.5. After approved validation all compared families are merged. Otherwise the trees are either not merged or sent to the user for validation. When the families have been validated they are removed from the Java map.
Step 7: During this step all inserted family matchtexts and all merged family matchtexts are being inserted and indexed by Lucene. This makes the new matchtexts searchable the next time a file is to be inserted.
Step 8: After insertion of matchtexts into Lucene there are still outdated matchtexts from families that has been merged. This is because merging two families result in a new family being made that is the result of the merge. The two families that were merged are still in Neo4j connected to the merged family (see section 3.4.4. This is good because information does not get lost when storing in this way. It is however not desirable that the families that were merged show up in a Lucene search. Instead what is wanted is that the resulting merged family is returned. In order to ensure this the two merged families are removed from Lucene's index so that only the merged family can be found when searching.

3.4.4 How Persons and Families are Being Compared and Merged

After finding potential family merges with Lucene these family merges were evaluated by a family comparison method. The method used a person comparison method to compare all the person nodes connected to the family nodes. How these comparison methods work will now be explained in more detail. First the person comparison method will be described and later on the family comparison method. It is important to understand that the matching methods should be able to deal with incomplete data. When properties or family members are missing there is an uncertainty of the matching result. This uncertainty should be expressed in the value the methods return. It is at the same time important not to regard a disability to compare two persons or families with that they can not be duplicates. Meaning that if there are missing property values so that two persons or families only can be partially compared. The missing property values can not be counted as mismatching property values.

The person comparison method compares two persons and then gives a decimal value between -1 and 1 over how similar they are. The method does this by comparing the normalized properties of the two persons and their sex and year properties. The other properties are not compared because they can contain minor errors which makes computer comparison difficult, this was mentioned in section 3.3. Getting -1 from the method means that all properties for the two persons are dissimilar, 1 that they are all similar. In the case some property values are missing or some properties are similar and others dissimilar the value will be somewhere in between -1 and 1. When there is no possibility to compare any property the method will return 0 . The score is calculated by first comparing the properties. The result of comparing two properties is either -1 if mismatch, 0 if comparison was not possible, 1 if match. The value from the comparison is than multiplied by the property weight for the corresponding property. A property weight is a weight for boosting a property. For example if it was most important that two persons had the same first and last name to establish similarity then the first and last name weights would be set higher then the rest. For the person comparison method used by the algorithm all weights were set to 1. After multiplication all values are summed up and then divided by the total sum of all property weights as to calculate the similarity score. This was done according to equation 3.1.

$$score_p = \frac{\sum\limits_{i=1}^{n}(w_i \cdot sim(p_{1,i}, p_{2,i}))}{\sum\limits_{i=1}^{n} w_i} \tag{3.1}$$

Equation 3.1: The equation gives the similarity score between two persons. The sim function compare two property values and gives the result -1, 0 or 1. The notation $p_{1,i}$ stands for a property value from one family and the notation $p_{2,i}$ from the other. The weight for each property is written as w_i.

The mean value is finally returned by the method. When comparing date properties the result is not -1, 0 or 1 but instead a scale between -1 and 1, depending on the dates proximity. All property matches or mismatches are multiplied by a corresponding match property weight.The match property weights where implemented in order too fine tune matching by assigning more important properties a heavier weight. In this algorithm all person properties were considered equally important so every weight was set to one. There were also weights for how different kinds of family members and family properties should be prioritized.

The family comparing method also returns a value between 1 and -1. When comparing two families first the family nodes are compared in the same way as person nodes. Then the person comparing method is used to compare the family members. The main problem with developing the family matching method is that it is hard to know which children to compare. This is due to that the families compared might be incomplete. Families can be incomplete in two ways either the nodes have missing properties or members of the family might be missing. In order to be certain of comparing the right children the family comparison method only compares children born closer than 4 months or children with the same first name. The other children does not get included in calculating the matching score. The values from matching the husband, the wife and the children gets multiplied with their corresponding weights. The husband and wife have weight 3 while the children have weight 1. This is due to that there is no possibility matching the wrong parents. The matched properties from the family nodes consisting of marriage date and marriage congregation are also multiplied by their corresponding property weights that are also set to 1. Then all values are summed up. The sum is finally divided by the sum of the husband and wife weights, the children weights times the number of children and the sum of the family property weights. This is expressed in equation 3.2. Finally the resulting value is returned by the method.

$$score_f = \frac{f_w \cdot simF(f_1, f_2) + \sum_{i=1}^{n}(p_w \cdot simP(p_{1,i}, p_{2,i}))}{f_w + \sum_{i=1}^{n} p_w} \tag{3.2}$$

Equation 3.2: The equation gives the similarity score between two families. The $simP$ function compare two persons and the $simF$ function compare two family nodes. The notation p_w stands for person weight. This weight is different for husband, wife and child nodes. The weight is 3 for both husband nodes and the wife nodes and 1 for the child nodes. f_w are the weight for the family node. The weight of the family node can be seen as the sum of the family node property weights divided by the number of properties.

It is important to realize that the families that are finally merged are also checked by Lucene's scoring formula. This formula can be seen in [17, p. 86]. Lucene is a state of the art search engine so it's scoring formula can be considered very good. There are however in this case two drawback with the check done with Lucene. When using Lucene two family matchtexts are compared. The matchtexts are made up of the family property values. So what Lucene does is that it checks how many words(property values) are the same for both matchtexts, see section 2.7 and [4] for more information. The foremost problem with this is that the individuals of the families compared are not distinguished from each other. So for example if the death date of a child in one family is the same as the death date of a completely different child in an other family the similarity score will still go up. In order to partly stop this one precaution was made. That was to set an extra letter for all properties indicating what role the person it belonged to had. The different roles are husband, wife and child and the respective letter h,w and c. This precaution however does not stop that different children properties still can become mixed up, as was mentioned above. The other problem with Lucene is that Lucene does an exact match between years. So slight variations in birth dates are considered no match at all. It was to overcome these limitations the comparisons method was implemented.

There were also merging methods developed for both merging persons and families. The problem with merging two persons is that they might have conflicting property values. In this case the newly inserted person's property values replaces the property values of the person node in the database. However if there are to many mismatches within two family trees both trees will be looked over by the user before merging. The idea is that the GUI should be developed to a level so that the user can decide which property values is correct. In the case of merging a person that is the result of a merge from two or several other persons the computer could of course also look at the property value of the previous inserted persons to make a decision. This was not implemented here because it would take to much time but can be a good idea for the future.

The hard thing when it comes to merging families is to make sure that the right children are merged together. This is done by using the same procedure as when deciding which children to compare. When two families are merged the families that gets merged are kept and referenced too by the new merged family. This is shown in figure 3.6. The figure only shows merging of two families not connected to any other families. Merging is more complicated when merging connected families which will be discussed in section 3.4.5. The reason for keeping the old families is to prevent lose of information. For example does lose of information occur when two persons have conflicting property values because then information from one individual is overwritten. Preventing lose of information is important because if a user wanted to verify that a merge has been done correctly it would be helpful to have the two families that were originally merged.

3.4.5 Iteration and Merging of Family Trees

One of the main goals of the thesis is making sure that the family merges are correct. The way to assure this is by traversing the family trees of the pairs of families suspected to be merged. A family tree for a family here consist of the family itself and the birth families of the parents and the family of the parents' parents and so on. Neo4j is built for fast traversal between nodes (O(1) for going from one node to the nodes connected to it)[14]. This makes Neo4j perfect for this validation method. Figure 3.7 below illustrates how two family trees are being traversed for comparison.

The evaluation procedure is done by pre-order traversal. The traversal method in the database uses an Evaluator object to evaluate if a merge is going to take place or not. The Evaluator object has three major methods. These are shown in the table below.

Evaluator methods
boolean stop(...)
void merge(...)
void storeInfo(...)

For every pair of families visited the stop method is called. The stop method

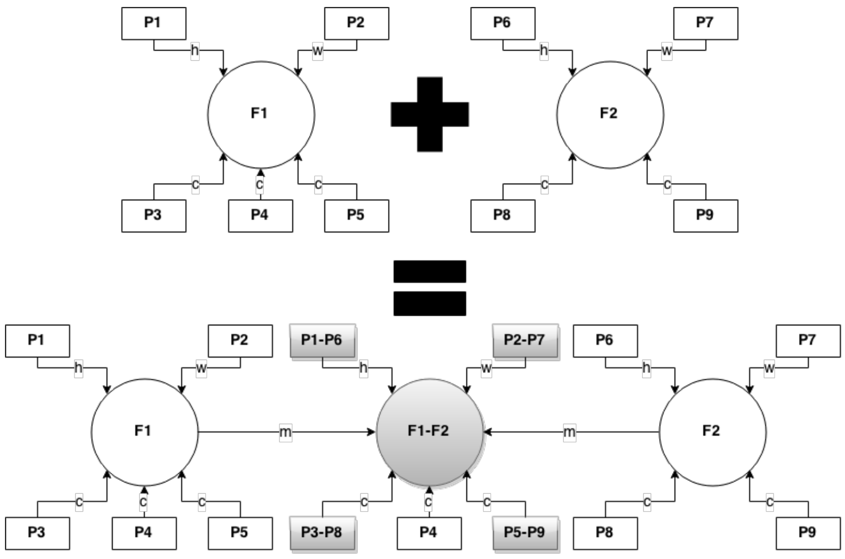


Figure 3.6: The figure illustrates two families being merged together. In the figure the round nodes are family nodes and the square nodes are person nodes. The notation F1 and F2 stands for two families with family id one and two. The notation F1-F2 means that the node is a merge of F1 and F2. The same goes for person nodes. The only difference is that a P is used to indicate persons. The green color represents that the node is the result of a merge. The different notations for relationships h, w, c, m stands for husband, wife, child and merge. In this example the two families are the same with one child missing in F2.

makes an evaluation whether to abort the merge or continue. This prevents unnecessary traversal of family nodes when the evaluation already has given a negative result. The stop method also classifies if two trees need to be manual merged by a user or can be merged automatically (henceforth families that belonging to trees that need to be manual merged will be called manual merges and family merges that are carried out by the computer automatic merges). The stop method takes a similarity score between the currently visited pair of families as an argument to make it's evaluation. The specific working of the stop method depends on which Evaluator object that is used. There were many Evaluator objects made in this thesis due to constant revision of the code. The Evaluator used to generate the result for this thesis simply takes the mean value for all family comparisons and checks that it is over a certain threshold set between zero and one. The threshold differs depending on how many families that has been compared. In the case many families have previously been compared during traversal the certainty is higher for a match between trees. This is compensated for by setting the threshold lower when many families have been compared. To preserve time the simple solution to

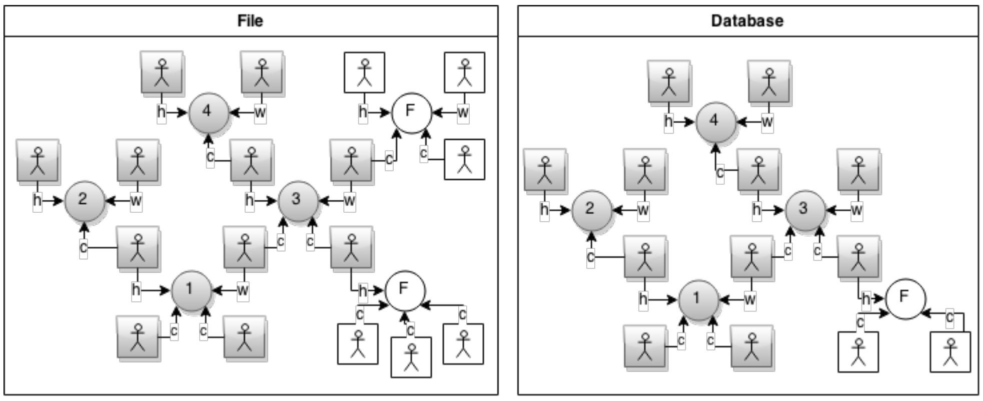


Figure 3.7: The figure shows how two trees being evaluated for merging are traversed. The left are from the a new input file and right has been generated from earlier inserted files. The square nodes are person nodes and the round nodes are family nodes. The marked nodes are the nodes being traversed. The numbering shows the traversal order.

use mean value was chosen. This part of the algorithm should be improved.

When the Evaluators stop method does not stop the traversal, it is stopped when there is no possibility to go further through older generations to compare families.

The Evaluator object merge method and store information method are used in combination to merge two trees. The store information method is used while traversing through the tress during evaluation. In the case the trees pass evaluation the information that was stored is used by the merge method to merge the trees. Figure 3.8 show two families being merged.

When two family trees are merged only the traversed family nodes are merged together. Family nodes only belonging to one of the family trees simply gets transferred over to the merged tree. This makes the merged family graph more complete which is the goal of the algorithm. From looking at figure 3.8 it looks odd how the families F6 and F11 has the same husband. This is however possible because remarriages are possible. So in this case the husband of F6 and F11 is considered remarried. It is still possible though that these families will later be merged. The pair F6-F11 might also be in the map containing pairs of families that need to be validated before merging. In that case they might later be merged by the algorithm. The problem with this is that now when traversing the trees of F6 and F11 the same node will be reached when doing the traversal namely node (F3-F9). When this happens the evaluation will not travel further along the branch belonging to the already merged node. The merging process is also tailored to handle this situation so that the already merged node will not be merged again. This makes the algorithm robust enough to cope with these cases. When the same node is visited during traversal the similarity score of one is also sent to the stop method of the evaluator to boost likelihood of a merge between two families having

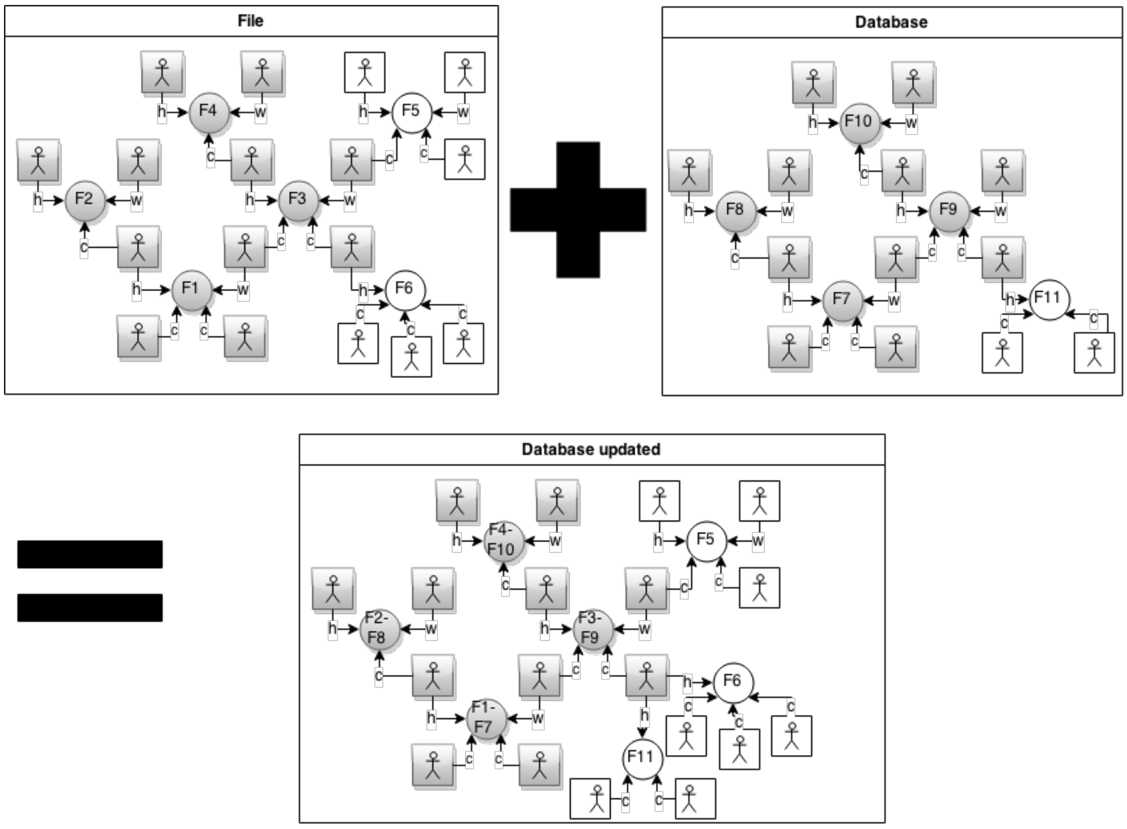


Figure 3.8: The figure show how the two graphs shown at the top are being merged together to form the graph at the bottom. The numbers are the ids of the family nodes. Two ids put together with a hyphen means that two families with the specified ids has been merged into a new family. The nodes of the top two trees that were going to be merged were marked and the nodes that were merged to form the bottom tree were also marked.

the same father or mother. The process for merging F6 and F11 is shown in figure 3.9. When the two families are merged the husband that they have in common are basically merged with itself which of course result in an exact copy of itself.

It was mentioned before that families that are merged are stored together with the family that is the result of the merge. This is shown in figure 3.6. However figure 3.6 only show how two families that have no connection to other families are merged. The merging procedure for merging two families in two family trees is a bit different. It was previously shown in figure 3.8 how family trees are merged. This figure is simplified and leaves out the process of merging and how the families that was merged are stored. The merging is in reality done for one family at a time so it can be said that the merging process goes through several stages before reaching the final merged family tree. This process is illustrated in figure 3.10. The

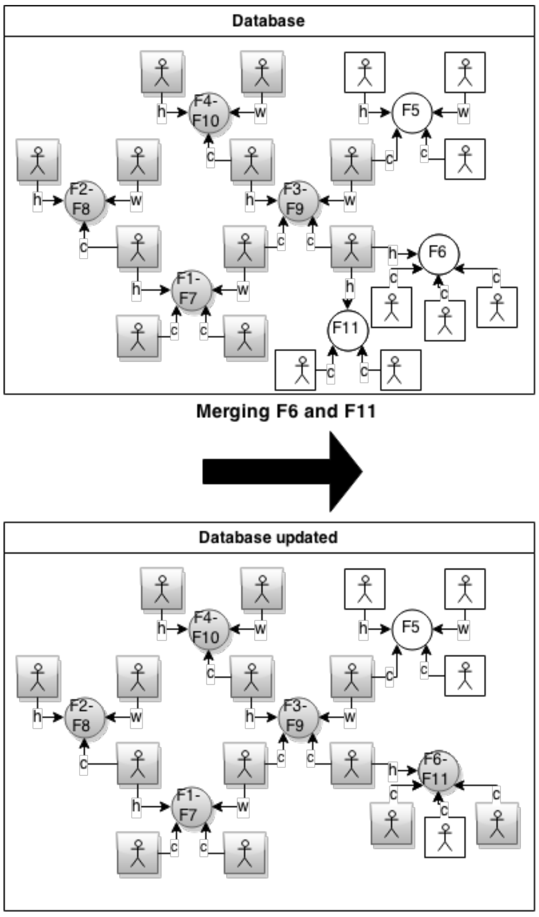


Figure 3.9: The figure show how a merge between F6 and F11 would look.

figure show three stages. In the first stage the two separate trees are shown. There are both actually in the database at this point. The file and database containers are there to mark that one of the trees is newly inserted and the other has already been integrated into the database. The merging process is done post-order, goes from the top of the tree downwards. Down to the family nodes F2 and F4 which were the starting nodes when traversing the trees for validation. In the second stage the two top families has been merged. The family nodes F2 and F4 are disconnected from F1 and F3. This is because the resulting merged family should be part of the network not the families that has been merged. The person nodes P3 and P11 also has to be duplicated. So that when families they belong to later are merged there can be references to two complete family nodes. This can clearly be seen in stage 3 where F1 and F2 does not share P4 but now has one copy each. In stage 3 the families F2 and F4 has also been merged. F5 is transferred to (F2-F4). This is done to make sure that nodes that has been merged are not part of the network, but instead the nodes resulting from the merge.

There was additional functionality needed for the family merging method for merging families that are part of a tree. The reason for this need is that when merging two families the husband or wife of the families below that later will be merged then already has been merged. In those cases the already merged parents need to be connected to the family node that is the result of the merge of the families below. So the family merge method was modified to receive already merged parents as arguments. In the case the merged method received any parents it did not merge the parents again but instead connected the them to the merged family node. This can be seen in figure 3.10 were (P4-P11) is merged in stages 2 and then connected to (F2-F4) directly in stages 3.

It was previously mentioned that the merging process was done post-order, going from the top of the trees to the bottom. It can seem odd that the merging process was not done pre-order as the traversals during validation. The reason behind doing the merging process post-order has to do with that families only can have one husband and wife while they can have several children. This leads to that it is easier to connect a family node to a husband or wife node because there can only be one husband or wife node to chose from while there can be several child nodes. This however leads to that it becomes very important to merge the correct children of one family that has a family below that should be merged next. Take for example P4 and P11 in figure 3.10. Both persons are merged in stage 2. There would be a problem if the wrong persons were merged at this stage. For example if P4 was accidentally merged with P10. Then the family trees would be put together inconsistently in stage 3. This can happen since the comparison procedure that chose which children to merge is not perfect. In order to prevent this error the children that were also parents of families to be merged in a later stage was stored together with the families where they were children. By storing those children it was made sure that they later were merged by the family merging method. It is during the validation procedure for merging tree the information necessary for merging is stored. During this procedure the person nodes that connects two families are collected and then stored together with the family it is a child in. This makes sure the merge method later knows which children that has to be merged together. Even though these procedures are put in place to prevent erroneous child merging. Errors still can accrue because the other children can be merged in an incorrect way. This is however not a huge problem because it only becomes one person that is merged incorrectly, but the person is stilled connected to the right family.

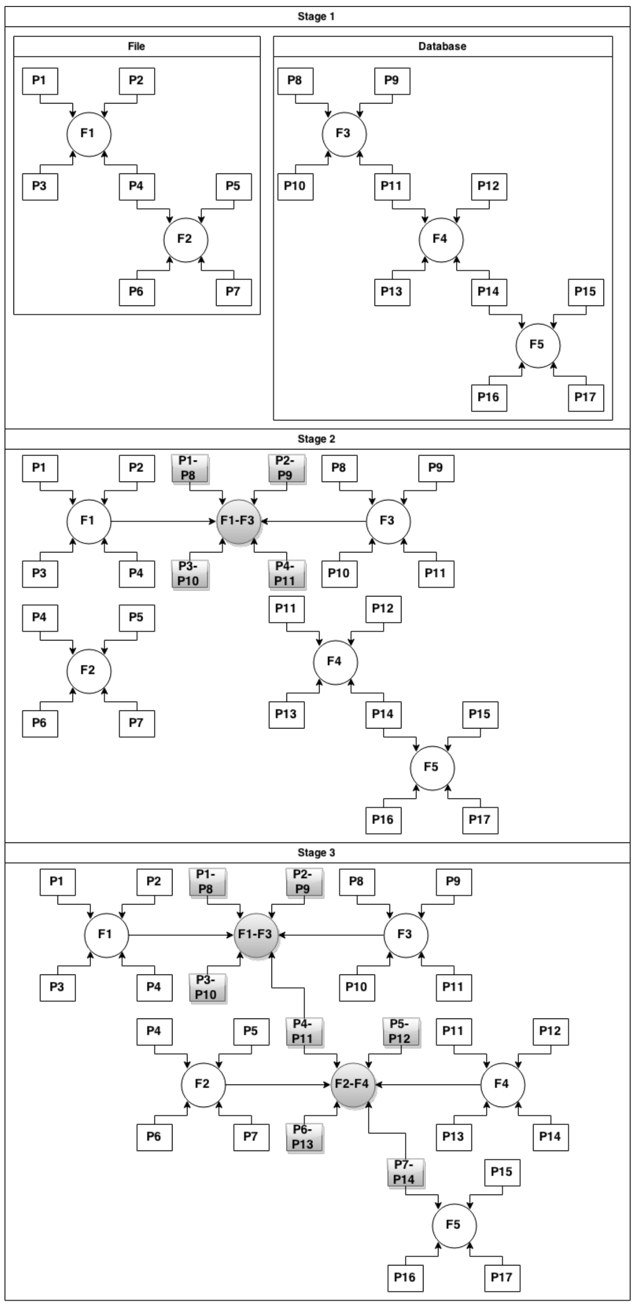


Figure 3.10: The figure show how two trees are being merged. The ids for each node is shown by the numbering. The person and family nodes are marked by a P and a F respectively. So a family node with id 1 is marked F1. The green nodes are the merged nodes.

3.5 Class Structure

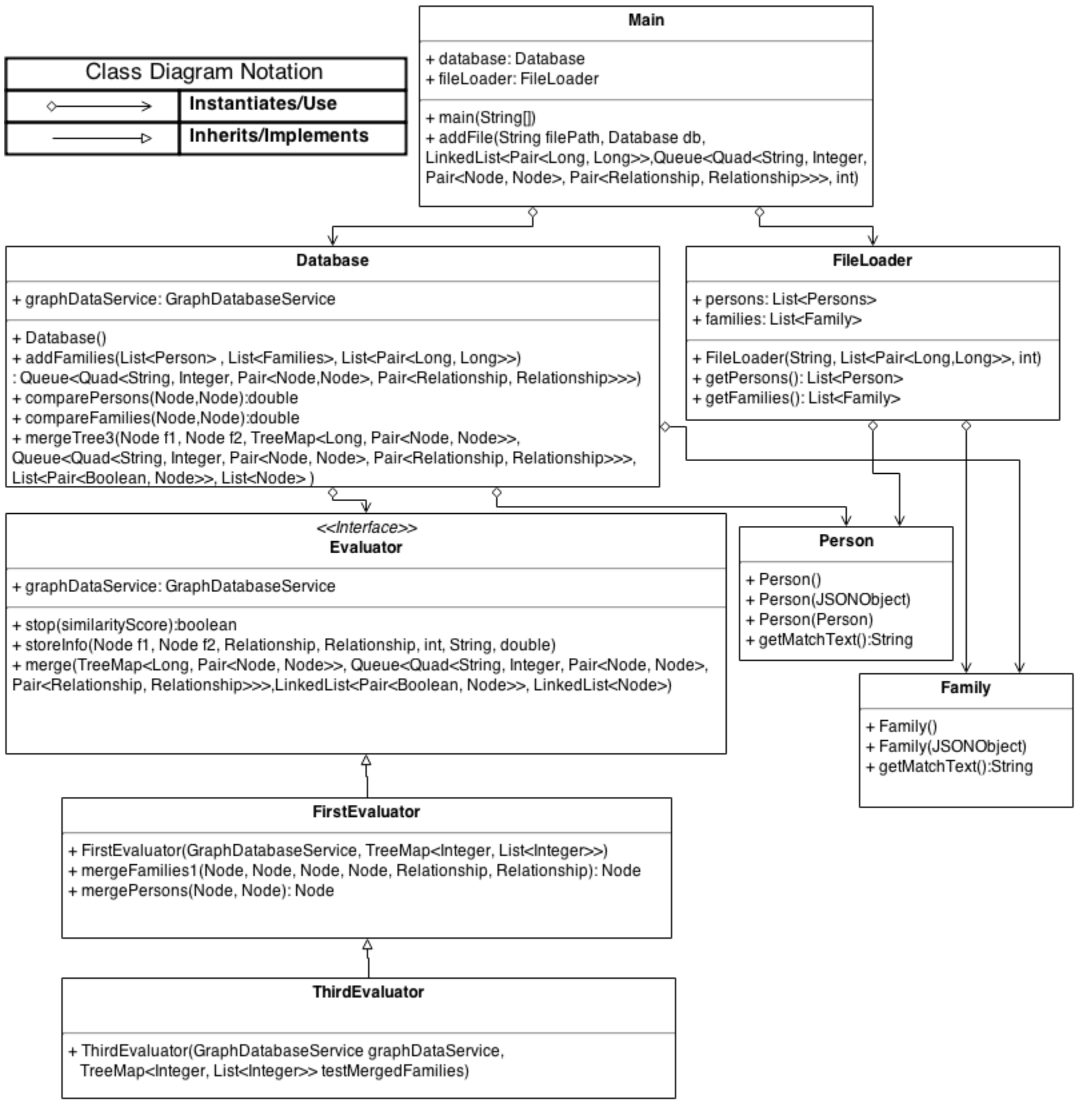

Class Diagram Notation
Instantiates/Use
Inherits/Implements
Main
+ database: Database
+ fileLoader: FileLoader
+ main(String[])
+ addFile(String filePath, Database db, LinkedList<Pair<Long, Long>>,Queue<Quad<String, Integer, Pair<Node, Node>, Pair<Relationship, Relationship>>>, int)
Database
+ graphDataService: GraphDatabaseService
+ Database()
+ addFamilies(List<Person> , List<Families>, List<Pair<Long, Long>>)
: Queue<Quad<String, Integer, Pair<Node,Node>, Pair<Relationship, Relationship>>>)
+ comparePersons(Node,Node):double
+ compareFamilies(Node,Node):double
+ mergeTree3(Node f1, Node f2, TreeMap<Long, Pair<Node, Node>>, Queue<Quad<String, Integer, Pair<Node, Node>, Pair<Relationship, Relationship>>>, List<Pair<Boolean, Node>>, List<Node>)
FileLoader
+ persons: List<Persons>
+ families: List<Family>
+ FileLoader(String, List<Pair<Long,Long>>, int)
+ getPersons(): List<Person>
+ getFamilies(): List<Family>
<<Interface>>
Evaluator
+ graphDataService: GraphDatabaseService
+ stop(similarityScore):boolean
+ storeInfo(Node f1, Node f2, Relationship, Relationship, int, String, double)
+ merge(TreeMap<Long, Pair<Node, Node>>, Queue<Quad<String, Integer, Pair<Node, Node>, Pair<Relationship, Relationship>>>,LinkedList<Pair<Boolean, Node>>, LinkedList<Node>)
Person
+ Person()
+ Person(JSONObject)
+ Person(Person)
+ getMatchText():String
Family
+ Family()
+ Family(JSONObject)
+ getMatchText():String
FirstEvaluator
+ FirstEvaluator(GraphDatabaseService, TreeMap<Integer, List<Integer>>)
+ mergeFamilies1(Node, Node, Node, Node, Relationship, Relationship): Node
+ mergePersons(Node, Node): Node
ThirdEvaluator
+ ThirdEvaluator(GraphDatabaseService graphDataService, TreeMap<Integer, List<Integer>> testMergedFamilies)

Figure 3.11: A simplified class diagram for the implementation of the algorithm. Smaller classes as well as less important methods were left out in order to make the diagram readable.

The class diagram for the algorithm is shown in figure 3.11. The two major classes for the algorithm is the FileLoader and the Database class. The FileLoader receive a file path for a json file when initiated. The json file is parsed by the FileLoader and the persons and families in the file get inserted into family and person objects. These persons and family objects are connected according to step

2 of the algorithm. Then they are put into lists. These lists are then sorted according to step 3. The lists are retrieved from the FileLoader by using it's getPersons and getFamilies methods. After the lists have been retrieved they are sent to the Database addFile method. This method carries out the remaining steps of the algorithm. The method first inserts all the persons and families into Neo4j. Lucene then searches through it's index to see if there are any similar families already in Neo4j. The compareFamilies method is used on the top five families resulting from the Lucene query. In the case one of the five families is similar enough to the query family both are put as a pair in a Java Map as was mentioned in step 5 of algorithm. The pairs in the map is then validated using Neo4j. This is done by repeatedly calling the mergeTree3 method for traversing the two family trees of one pair at a time. The mergeTree3 method uses the recursive method mergeTree4 to traverse the pair of families two corresponding family trees. This process was described in section 3.4.5. As mentioned before it is during this traversal an Evaluator is used. For an object to be an Evaluator it must implement the interface Evaluator. There were three evaluators implemented during the implementation. This was because the first two needed improvements. The ThirdEvaluator was the one that was finally used. What is unique about the ThirdEvaluator is that it's merging process is more advanced. The ThridEvalutor's merging method does not need to merge two entire trees but can merge sub-trees of two trees that are partly matching.

3.6 GUI

Unfortunately there was not enough time to add any functionality to the GUI (Graphical User Interface). So only the shell of the GUI was created. Using the shell a couple of example graphs were uploaded to it. This to illustrate how the merging results was intended to be represented to the user. The GUI is shown in figure 3.12. The parts of the GUI implemented so far is only for looking at the algorithm's resulting merged trees and trees that need manual merging. In a real application the GUI would probably be extended to also include different search capabilities such as searching for a family and getting back potential family trees for that family. Browsing around the entire graph instead of just looking at trees would probably also be of interest. The way of representing information to the user here has this in mind. So that the GUI later easily can be extended to included these capabilities.

The structure used in Neo4j for storing family trees is also used when representing the information to the user. So person and family nodes are shown linked together into trees. This is good for giving the user a understanding of how the data is stored. Information in Neo4j can also easily be accessible this way by having references(ids) for the nodes on screen to the stored nodes in Neo4j. This is for example useful when editing nodes in Neo4j through the GUI. All that needs to be implemented then is some method that retrieves the node directly from Neo4j using it's id(O(1)) and then changes it the same way that was done in the GUI. Searching and browsing can also easily be implemented using Lucene to search for potential start families and then Neo4j for retrieving connected families.

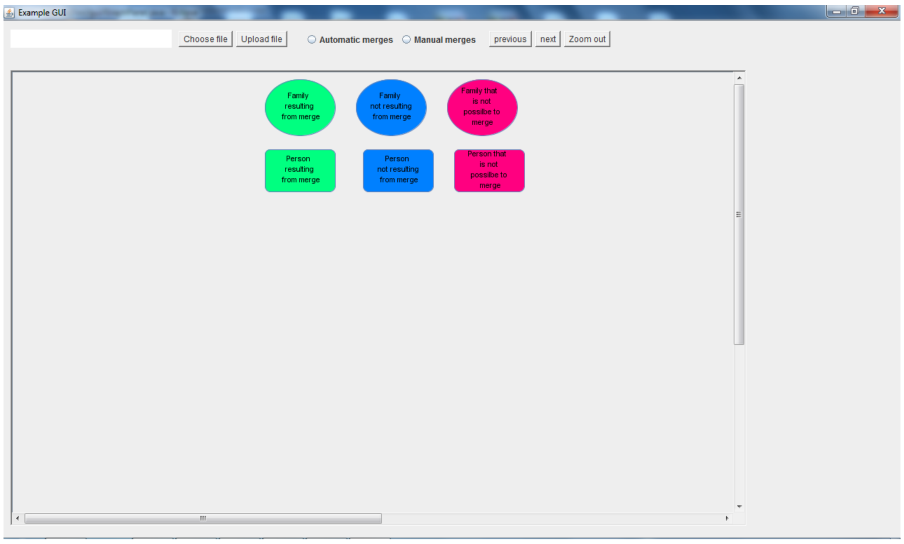

Example GUI
Choose file
Upload file
Automatic merges
Manual merges
previous
next
Zoom out
Family resulting from merge
Family not resulting from merge
Family that is not possilbe to merge
Person resulting from merge
Person not resulting from merge
Person that is not possilbe to merge

Figure 3.12: A picture of the GUI. The field to the upper left is for uploading files to the GUI. When clicking on "chose file" a dialogue window pops up which makes it possible to select a json file for uploading. The clicking on "upload file" calls on the algorithm which inserts the selected file. The upper right field is for checking the tree merges of the previously inserted file. Filling the boxes lets one chose between looking at automatic tree merges or trees that need to be manual merged. The next and previous buttons are to go back and forth between the different automatic merged trees or the trees that need to be manual merged. The zoom out button is for zooming out. The big area in the lower center of the GUI is for displaying the different trees. The boxes with the text in are for showing the color and shape coding for both person and family nodes and if it is a merged person/family, not merged person/family or a person/family that it was not possible to merge.

When representing the merge results to a user it is important that the person and families are shown in a way that does not make the user overwhelmed with information. It is also important that the information that is represented is understandable. An idea for this GUI was to cut down the information shown to the user by having a zoom in/zoom out function. When zoomed out either a tree resulting form a merge is shown or a tree that would have been the result of a merge depending on if automatic merged trees are shown or the trees that need to be manual merged. The trees that need to be merged manual are represented as they were already merged(even though they are not) but the persons and families that could not be merged are highlighted. Examples of how this can look is shown in figure 3.13 and 3.14. The zoom in functionality was intended to work in the way that the user double clicks on a family node to zoom in. When zoomed in on an automatically merged family the GUI shows all the members of the merged family and the families that were used to make up the merged family. When zooming in on families for manual merging the same is shown, but here the family that is the resulting merge is not in the database but show what is wrong with the merge.

The idea is that it should be possible to click on the person and family nodes to correct the errors. Then the merge gets stored for real in Neo4j and Lucene. Examples of when the GUI is zoomed in on families is shown in figure 3.15 and 3.16. Different shapes are used to distinguish between person and family nodes. The persons are boxes and the families are ellipses. A color scheme was also included to be able to see the difference between three types of nodes namely nodes that are the results of a merge or should be merged, nodes that are not the result of a merge and nodes that should be but can not be merged. The color green was used for merged nodes, blue for nodes that are not the result of a merge and red for the nodes that could not be merged. When showing manual merges the same colors where used even though nothing was ever merged, here green means that a person or family node should be merged instead. It can be seen in figure 3.13 that all family nodes are green for the automatically merged families. Figure 3.14 show that for the manual merged trees the families not possible to merge was highlighted red. The color highlighting is also present when zooming in.

A side note is that the insertion algorithm make judgements on the entire family tree. While the GUI make judgements on each individual family and person. There can therefore be families among the merged trees that have more conflicts then the families that are looked over for manual merging. This can become confusing to the user. One way to fix this would be to change the algorithm by making a new Evaluator were it is enough that only one family or person is conflicting to consider a whole tree to need to be manually merged. This was not tested here but it can be a good idea. In order to do this some threshold need to be implemented so that some properties of family and person nodes can conflict without it needing to be manually merged.

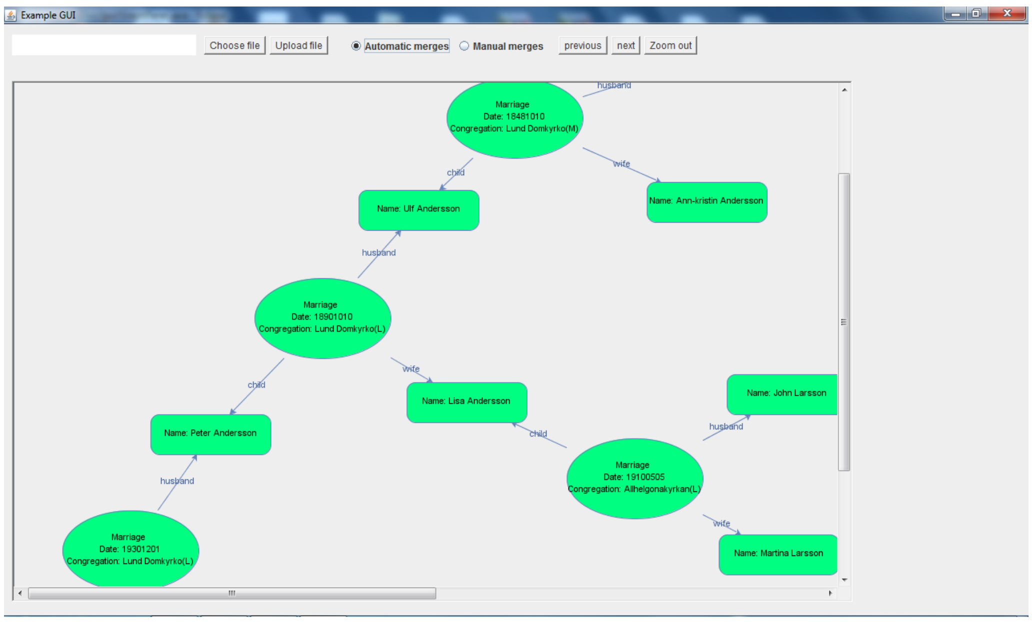


Figure 3.13: The figure shows an example of an automatic merge. The tree that is shown by the GUI is the resulting tree from a merge of two family trees. The two trees that has been merged are not shown only the resulting tree. Children not connecting two families in the tree are also not shown. Only partial information of both person and family nodes are shown. All this to give a clear view over how the merged tree turned out.

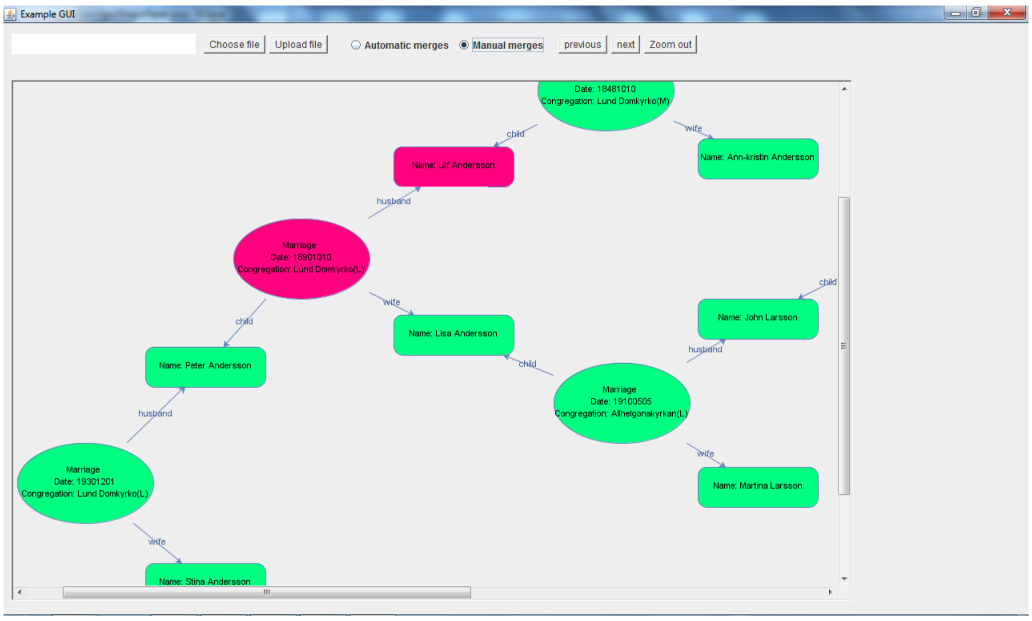
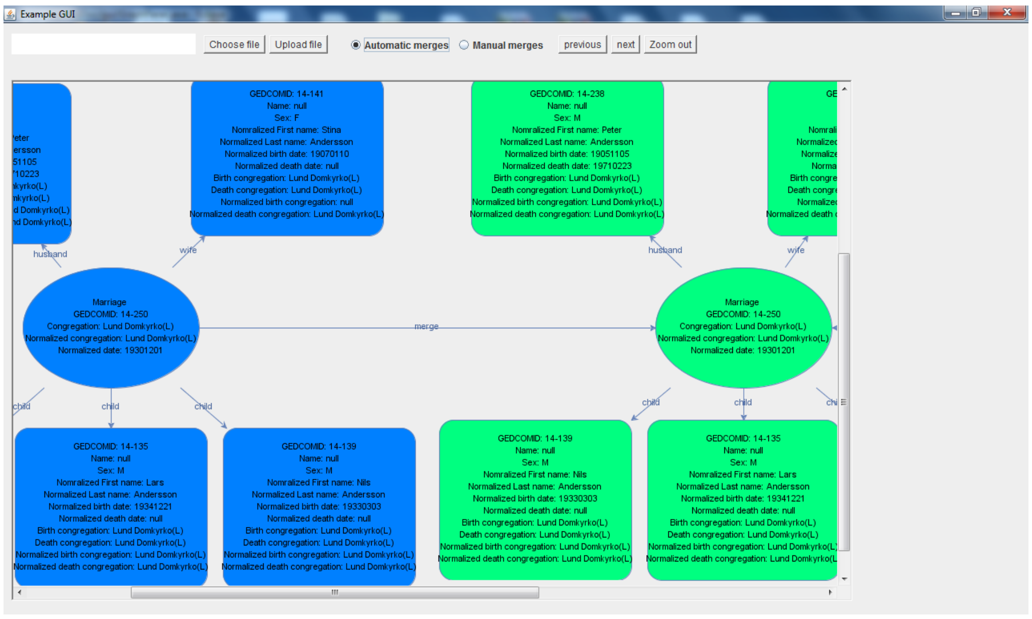


Figure 3.14: The figure show an example of an manual merge. The tree that is shown by the GUI is what would have been the result of merging two trees. The red color here indicates a family or person mismatch. This makes it easy for the user to manual look over what in the tree that might need to be changed.



GEDCOMID: 14-141
Name: null
Sex: F
Nomralized First name: Stina
Normalized Last name: Andersson
Normalized birth date: 19070110
Normalized death date: null
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: null
Normalized death congregation: Lund Domkyrko(L)

GEDCOMID: 14-238
Name: null
Sex: M
Nomralized First name: Peter
Normalized Last name: Andersson
Normalized birth date: 19051105
Normalized death date: 19710223
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

Marriage
GEDCOMID: 14-250
Congregation: Lund Domkyrko(L)
Normalized congregation: Lund Domkyrko(L)
Normalized date: 19301201

Marriage
GEDCOMID: 14-250
Congregation: Lund Domkyrko(L)
Normalized congregation: Lund Domkyrko(L)
Normalized date: 19301201

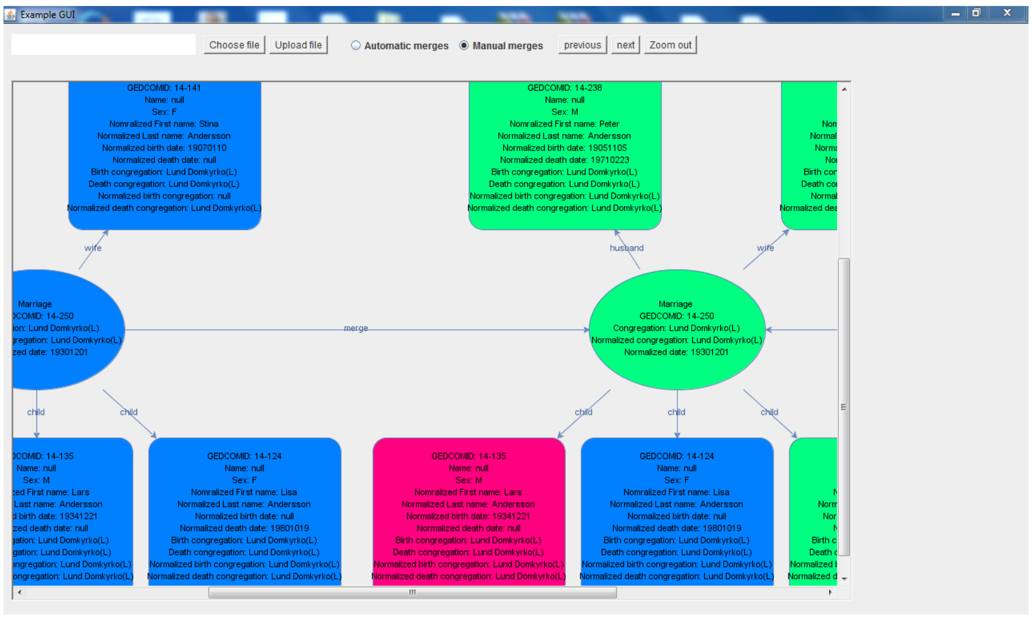
GEDCOMID: 14-135
Name: null
Sex: M
Nomralized First name: Lars
Normalized Last name: Andersson
Normalized birth date: 19341221
Normalized death date: null
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

GEDCOMID: 14-139
Name: null
Sex: M
Nomralized First name: Nils
Normalized Last name: Andersson
Normalized birth date: 19330303
Normalized death date: null
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

GEDCOMID: 14-139
Name: null
Sex: M
Nomralized First name: Nils
Normalized Last name: Andersson
Normalized birth date: 19330303
Normalized death date: null
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

GEDCOMID: 14-135
Name: null
Sex: M
Nomralized First name: Lars
Normalized Last name: Andersson
Normalized birth date: 19341221
Normalized death date: null
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L

Figure 3.15: The figure show an example of an automatic merge zoomed in on a family. The resulting merged family is shown and one of the families that were put together to create it. In this view most of the information for every member of the family and the family node itself is shown.



GEDCOMID: 14-141
Name: null
Sex: F
Nomralized First name: Stina
Normalized Last name: Andersson
Normalized birth date: 19070110
Normalized death date: null
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: null
Normalized death congregation: Lund Domkyrko(L)

GEDCOMID: 14-238
Name: null
Sex: M
Nomralized First name: Peter
Normalized Last name: Andersson
Normalized birth date: 19051105
Normalized death date: 19710223
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

Nom
Normal
Norma
Nor
Birth cor
Death co
Normal
Normalized dea

Marriage
DCOMID: 14-250
ion: Lund Domkyrko(L)
gregation: Lund Domkyrko(L)
zed date: 19301201

Marriage
GEDCOMID: 14-250
Congregation: Lund Domkyrko(L)
Normalized congregation: Lund Domkyrko(L)
Normalized date: 19301201

DCOMID: 14-135
Name: null
Sex: M
zed First name: Lars
Last name: Andersson
d birth date: 19341221
zed death date: null
gation: Lund Domkyrko(L)
gation: Lund Domkyrko(L)
ongregation: Lund Domkyrko(L)
ongregation: Lund Domkyrko(L)

GEDCOMID: 14-124
Name: null
Sex: F
Nomralized First name: Lisa
Normalized Last name: Andersson
Normalized birth date: null
Normalized death date: 19801019
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

GEDCOMID: 14-135
Name: null
Sex: M
Nomralized First name: Lars
Normalized Last name: Andersson
Normalized birth date: 19341221
Normalized death date: null
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

GEDCOMID: 14-124
Name: null
Sex: F
Nomralized First name: Lisa
Normalized Last name: Andersson
Normalized birth date: null
Normalized death date: 19801019
Birth congregation: Lund Domkyrko(L)
Death congregation: Lund Domkyrko(L)
Normalized birth congregation: Lund Domkyrko(L)
Normalized death congregation: Lund Domkyrko(L)

N
Norm
Nor
N
Birth c
Death c
Normalized b
Normalized d

Figure 3.16: The figure show an example of an manual merge zoomed in on a family. How the merge result would have turned out if both families were merged is shown to the right and to the left one of the families that should be merged. The members of the family that can not be merged are highlighted red. Family members that could be merged are green and family members not needed to be merged are marked blue. The family node also changes color depending on it could be merged or not.

Chapter 4

Tests

This chapter will present the different performance tests carried out on the algorithm and give motivation for their relevance. There were two types of tests namely for time complexity and accuracy. In order to carry out the tests, code was added to the algorithm to retrieve information about execution time and which families that were merged. More on this in section 4.1 and 4.2. All tests were carried out on a Lenovo N500 running Windows 7 Professional N 32 bit. The processors were Intel(R), Pentium(R) Dual CPU T3200, both with a speed of 2GHz. The RAM memory was at 3GB and Java version 7 was used to run the program. Five files were available for testing [5]. Each file contained a different amount of families and persons. The file size varied from about 50 families and 200 persons in the smallest file to about 6500 families and 20000 persons in the largest. A list of the files can be seen below.

- pilot1.json
- pilot2.json
- pilot3.json
- pilot4.json
- pilot5.json

From [5] also files containing merged pairs of families from an earlier genealogical database prototype was available. In that work a MySQL database was used for storage. The solution from the earlier prototype was created by inserting different combinations of the listed input files into it. Before doing the insertion the earlier prototypes database was always empty then one file was inserted to make up the database and then an other one to be the input file. The same method was used in this work but the major difference is that in this application a graph database and a different insertion algorithm was used. The different insertion combinations that were carried out are shown in table 4.1.

This resulted in ten different results for each approach(this work and [5]). The results contained both measurement for execution time and which families that were merged together (see section 4.1 and 4.2 for more information). The point with doing the test this way was to let one file constitute the database and the other the input file. This in order to make it possible not only to see how the execution time varied with input file size but also with database size.

File combinations	
Input file	**Database containing**
pilot1.json	pilot3.json
pilot1.json	pilot4.json
pilot1.json	pilot5.json
pilot2.json	pilot1.json
pilot2.json	pilot3.json
pilot2.json	pilot4.json
pilot2.json	pilot5.json
pilot3.json	pilot4.json
pilot3.json	pilot5.json
pilot4.json	pilot5.json

Table 4.1

4.1 Time complexity

In order to determine the time complexity of the algorithm the five json files where inserted into the algorithm. The execution time for different algorithm steps and the entire algorithm was calculated using the java command System.currentTimeMillis(). The command returns a value from javas internal clock regarding current time in ms. Using the command once before and once after an algorithm step, an algorithm step's execution time can be calculated. This resulted in eight sets of execution times for the eight algorithm steps. The execution time for the entire algorithm was also calculated. During execution of the algorithm other relevant data was also collected such as number of families in the input file, number of families in the database and number of merged families. This was done for all the different combinations of files shown in the table above.

All information was retrieved by adding lists to the code. The lists were then filled with the information previously mentioned. The lists were finally processed and written to a Matlab file. Matlab was then used to generate graphs for time complexity for the entire algorithm and for the different algorithm step's. This was done in relation to number of families in the input file, number of families in the database and number of merged families. When making a graph every data point represented a measurement from merging a combination of files. For example when drawing the execution time for the entire algorithm all ten combinations of files where needed so that the correlation between execution time and number of merged families could be seen (see section 5.1). In Matlab many different graph were generated, only a few of those were considered relevant. The relevant graphs are shown in section 5.1.

Comparisons	
Algorithm	**Previous prototype**
Automatic merges	Automatic merges
Automatic merges	Manual merges
Manual merges	Automatic merges
Manual merges	Manual merges

Table 4.2

4.2 Accuracy

The point of the accuracy testing was to find out the precision, recall and F-score of the algorithm (see section 2.2). This was done by comparing the pairs of merged families from the algorithm with the pairs of merged families from the previous prototype[5]. In this case the previous prototype was regarded as a correct solution. The merges for the algorithm were retrieved by adding code to the algorithm so that gedcomids representing the merged families were inserted into two lists. There were one list for storing automatic merges and one for manual merges. The comparison was then made by comparing the gedcomids representing the merges from the algorithm with the gedcomids representing the merges from the previous solution. It was previously mentioned in section 3.3 that the gedcomids were unique for every family in the five files. Therefore it was enough only to compare gedcomids to determine if two families were the same.

The previous prototype's solutions were available as ten files [5]. There were one file for every combination of input files merged. When testing one of the combinations, the input files were first inserted into the algorithm and then corresponding previous solution's file was uploaded for comparison. This was repeated for all ten combination resulting in ten different comparisons between the algorithm and previous prototype.

The comparisons were made by comparing four lists with each other. Two lists were from the algorithm's merges and two were from the previous prototype's merges. There were two for each solution since there were two kinds of merges namely automatic and manual merges. The lists were compared with each other in four combinations. These combinations are shown in table 4.2. Finally the number of automatic and manual merges was extracted to Matlab together with the comparison results. In Matlab bar plots were generated to show the comparison results. Calculations were also made to determine the precision, recall and F-score of the algorithm.

Chapter 5

Results and Discussion

5.1 Time Complexity

This section will both present results from the test on time complexity explained in the test section and a deeper analysis. The analysis will mainly be about what time complexity to expect when the database becomes large.

The algorithm seems to have a linear time complexity in relation to families being merged according to figure 5.1. About 75 percent of the execution time is during algorithm step 6 (validate and merge) which can be seen by comparing figure 5.1 and figure 5.2.

In order to understand the relation between database size and execution time a deeper analysis of the algorithm is necessary. Before the validation and merging step Lucene searches the database several times to find potential family merges. This step is most dependent on database size. In section 2.7 it was written that a Lucene search had the time complexity $p \cdot log(k)$ where k is the number of families retrieved and p is the number of postings in the postings lists for the terms of the query. The algorithm uses Lucene to search the database one time for every family in the input file, for every query the top five most similar family candidates are retrieved. This makes the time complexity for searching for every family in the input file containing n families $n \cdot p_{average} \cdot log(5)$. The p_{averge} stands for average number of postings in the posting lists for the terms in an averaged sized query. There is of course a linear dependence between the input file size and the execution time for searching all families in the input file. It is however harder to know the dependence between the database size and the execution time for searching. This is because it is hard to know the dependence between $p_{average}$ and the database size. The $p_{average}$ is dependent on two factors. The average number of terms in a query and the average length of the postings lists. The average number of terms in a query is dependent on the average size of the families. The average length of the postings lists however is indirectly dependent on the database size. A postings list is basically a list containing the id's of the documents(families) containing a specific term(property value). So in the case new families are inserted that have few property values in common with previously inserted families the postings lists do not grow much. Instead new postings lists are created. In this case the execution time for the Lucene search is not effected much because the $p_{average}$ does not grow. This is what makes it hard to know the relation between database

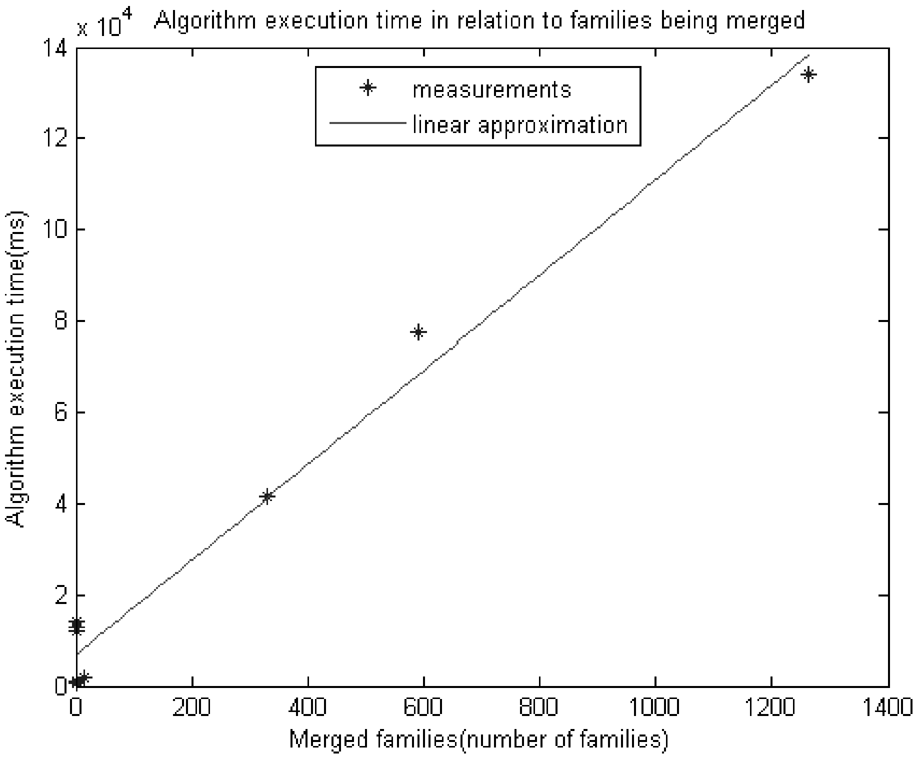


Figure 5.1: The graph show how long time it takes to execute the entire algorithm in relation to how many families are merged.

size and execution time. So tests on larger sets of data is necessary. In the case the search would take to long on a large database it would be possible to start using approximative methods to lower the execution time, these were shortly discussed in section 2.7. For the test carried out in this thesis all Lucene searches did not take more than a few seconds.

The more trivial steps in the algorithm can of course also start to play a role when the database becomes large. Adding the nodes to Neo4j or documents to Lucene might for example become slower when the number of persons and families in the database are already large. Theoretically insertion into Neo4j should not depend on the number of already inserted nodes. It is worse for Lucene where the postings lists has to be iterated through to remove the documents (families) that were taken away. Insertion and removal of documents in Lucene is quite an advanced subject, see [17] for more information.

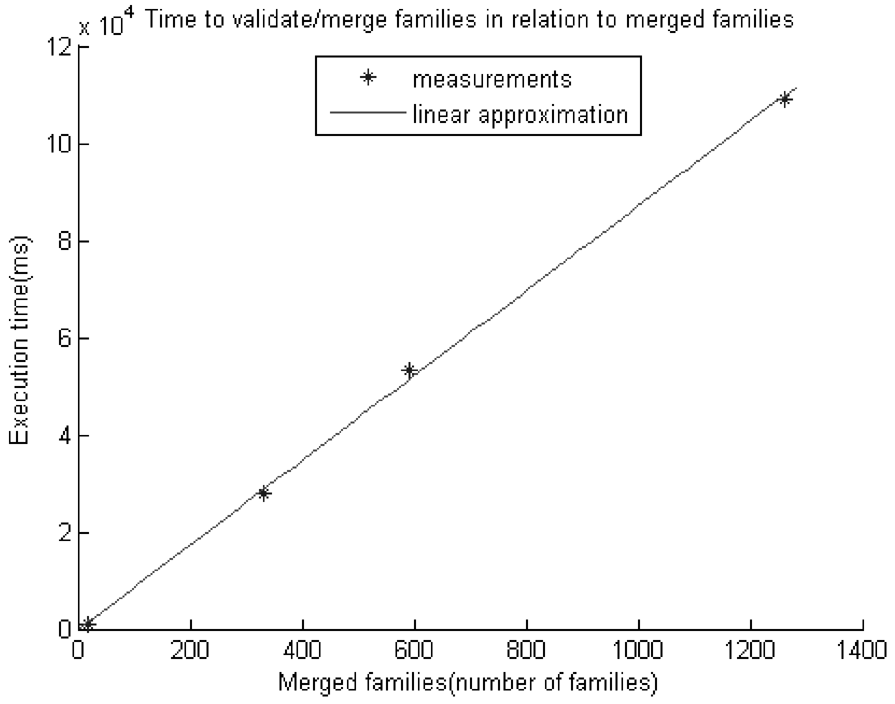


Figure 5.2: The graph shows how long time it takes to validate and merge families in relation to how many families are merged.

5.2 Accuracy Compared to Previous Research

This section will first present the result regarding accuracy of the algorithm as compared to previous implementation[5] and then give deeper analysis into the result. It is in this case hard to give any absolute correct answer regarding accuracy because it is not certain that previous prototype gave a correct solution. For easy reading the previous prototype's result will here be called the previous solutions. Matches and manual matches found by the previous solution will be called previous match or previous manual match. As mentioned in the test section were the json files inserted into the algorithm and previous prototype in ten different combinations. Then the result from inserting the files into the the algorithm and the previous prototype were compared. This resulted in ten comparisons in total. The results from the comparison all gave a similar distribution of families that were merged by both solutions in relation to how many families that were merged individually by the solutions. The ten results where therefore combined into one result. This result is illustrated in figure 5.3.

The great difference between the applications is that there are almost no manual matches found by the algorithm. This is because the algorithm compares two trees with each other and not two families. This results in that the matches found become very certain or disregarded completely. This might result in some families

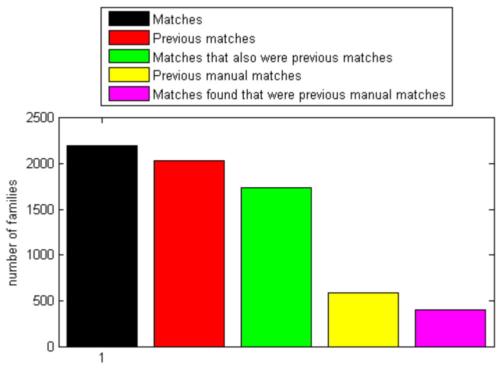


Figure 5.3: Shows a summation of the result from comparing the five files in the ten different combinations presented in the test chapter. There were so few manual merges done by the algorithm so these are not shown in the bar diagram.

Automatic merges		
Precision	Recall	F-score
79%	85%	82%

Table 5.1

within a tree being merged wrongly. The current version of the algorithm does not take care of this. Looking at figure 5.3 it becomes clear that the algorithm finds more family matches but it also becomes clear that many matches found were considered manual matches by the previous solution. The tables 5.1 and 5.2 show the precision, recall and F-score for the automatic and manual merges. The two tables does not take into consideration that many of the automatic merges were found as manual merges by the previous solution. In order to show this the table 5.3 was made.

In table 5.3 separation between manual and automatic merges were not considered. Instead it is shown how many manual or automatic merges that were found by the algorithm that were also found as either automatic or manual merges by the previous solution. The precision shown in table 5.3 is much higher then in table 5.1. This is due to what was previously written, that many automatic merges found by the algorithm were considered manual merges by the previous solution.

Manual merges		
Precision	Recall	F-score
0%	0%	0%

Table 5.2

Automatic and manual merges		
Precision	Recall	F-score
97%	81%	88%

Table 5.3

From looking at table 5.3 it becomes clear that the automatic or manual merges found by the algorithm is also found by the previous solution. Meaning that the algorithm has a high precision. On the other hand there are many automatic and manual merges found by the previous solution that are not found by the algorithm. However it is hard to know whether or not these extra merges are correct or not. One theory could be that the extra merges are discarded by the algorithm during the tree traversal. In this case it can be interesting to consider that here are places were the genealogist have connected the wrong families. So an interesting idea can be to extend the current algorithm with some sort of detection mechanism for these situations. This could be done by taking cases were a couple of connected families had been given a big similarity score and then the similarity score declined when traversing a specific branch. This could improve the recall.

Chapter 6

Conclusions and Future Work

There were a number of questions in the introduction of this thesis. Short answers or references where to find information to these question will now be given as a conclusion. The following list is the same list that was shown in the introduction.

1. How shall information be saved within Neo4j?
2. How shall basic operations such as insertion, removal and search be implemented?
3. What is the time complexity for the insertion operations?
4. How accurate is the insertion method at detecting duplicates between the database and the insertion file?
5. How shall information about duplicate families (conflicts, two possibilities) be represented to the user?

Question number one is answered in section 3.4.2. The second question is basically three questions. The algorithm that was developed was for insertion. So the answer for how to implement the insertion is the algorithm, which is explained in it's totality in section 3.4. A basic overview of the algorithm is shown in subsection 3.4.3. Removal is actually a pretty trivial issue. Removing a family or person is basically just to use the removal operations in Neo4j and Lucene. There could however be ramification for removing a family node in Neo4j. In the case there would be person nodes that were not connected to any other families it would probably be desirable to remove them as well. This can be accomplished by traversing from all family member nodes in order to check if they are connected to any other nodes and if not remove them. It is desirable to remove disconnected person nodes because there are no way they can be retrieved through a search. This is due to that they can not be found with Lucene and not traversed to from an other node in Neo4j when disconneted. An other obvious problem with removing a person or family node is that this can lead to that families that were previously related are being disconnected by removing the person or family. This is however something that the user has to look at when deciding whether or not to remove the person or family.

The removal process in Lucene is not that complicated either. One of the main problems is that Lucene has to be queried to find the family that has to be removed. Lucene can only search for families because there are only family matchtexts stored in Lucene. Therefore when removing a person, a queries for the families nodes it is connected to has to be made. This is done by traversing from the person node to one of it's neighboring family nodes and then search for those families with Lucene. Then what basically needs to be done is that for all of those families the specific part of their matchtexts coming from the person that is being removed has to be removed. This is accomplished by first removing the matchtexts in Lucene and then inserting new ones without the removed person's property values. Searching can basically only be done for families or family trees. Searching for a family is just to use Lucene. Implementing searches for family trees can de done using the Lucene search on the root family node of the tree and then use the validation procedure in Neo4j.

The answer to the third question is linear time-complexity in relation to number of families merged but further testing on larger data-sets is necessary to determine this. Question number four is answered in section 5.2. Section 3.6 covers the last question.

The problem of finding and merging family trees can be considered an open problem, there are therefore an endless number of solutions. In this thesis an algorithm using Lucene and Neo4j in collaboration was developed. The point of this collaboration was to use the strength of both tools to get the best result in regard to time complexity and accuracy. The basic idea was to first use Lucene to find which families to merge. This was accomplished by Lucene through the use of the vector space model. The gives a very time effective way of finding which families to merge. Neo4j is then used to traverse trees for validation and merging. This is done quickly because the time complexity for going between neighboring nodes is O(1) in Neo4j. This leads to the algorithm being very time effective. With the longest execution time being about two minutes. This makes the algorithm promising for future work. The remaining problem for improving the algorithm time complexity lies in optimizing the queries made by Lucene. This can hopefully be done using the approximative searches discussed in the theory chapter.

In this thesis the application was not tested against any proven earlier results. So there is a need to test the application on two merged files where the result has been looked over by genealogist. There is also a need for testing that the application keeps the data within Neo4j in a consistent state when many consecutive insertions of files are being made.

The algorithm can also be improved in several ways. The most obvious improvement would probably be to change the Evaluation method so that more merges become manual merges instead of automatic merges. This can be accomplished by instead of only calculating the average value of all family comparisons also calculate the variance of the family comparisons. A high variance would mean that there are probably families that have to low similarity score to be merged automatically. An other way of accomplishing the same goal can be to look at the comparison value individually for each pair of families compared. In the case the comparison value is too low for one or a couple of the family pairs the merge can be considered manual.

The GUI was also not completed. So some development as to how the GUI should be connected to the algorithm and how the nodes represented by the GUI can give access to the nodes stored in Neo4j and the documents stored in Lucene also need some further investigation. The way the algorithm chooses which children that should be merged can be improved. For example by comparing the other families the children are connected to in order to decided which children are the same.

Bibliography

[1] URL: http://web.mit.edu/16.070/www/lecture/big_o.pdf (visited on 12/20/2015).

[2] Eric W. Allender et al. *Computer Science Handbook (Second Edition).* Ed. by Allen B. Tucker. Chapman & Hall/CRC, 2004. ISBN: 978-1584883609.

[3] G. Kirby et al. *Comparing Relational and Graph Databases for Pedigree Data Sets.* URL: http://socialhistory.org/sites/default/files/docs/kirby_et_al_-_database_comparison.pdf (visited on 03/03/2014).

[4] *Apache Lucene - Scoring.* URL: http://lucene.apache.org/core/3_5_0/scoring.html (visited on 11/20/2014).

[5] Anders Ardö. Department Electrical and Information Technology, Lund University.

[6] Douglass R. Cutting and Jan O. Pedersen. "Space Optimizations for Total Ranking *". In: *In Proceedings of the RIAO97 Conference - Computer-Assisted Information Searching on Internet* (1997), pp. 401–412. URL: http://lucene.sourceforge.net/papers/riao97.ps.

[7] *Datorhjälp i Släktforskningen.* URL: www.dis.se (visited on 01/27/2014).

[8] *Eclipse, Kepler.* URL: http://eclipse.org/kepler/ (visited on 12/16/2015).

[9] *Family Tree & Family History at Geni.com.* URL: http://www.geni.com (visited on 12/01/2015).

[10] *Free Family History and Genealogy Records.* URL: https://familysearch.org/ (visited on 12/19/2015).

[11] *Free Family Tree, Genealogy and Family History - MyHeritage.* URL: http://www.myheritage.com (visited on 12/03/2015).

[12] A Wong G. Salton and C.S. Yang. “A Vector Space Model for Automatic Indexing”. In: *Communications of the ACM* 18.11 (Nov. 1975), pp. 613–620. URL: http://mall.psy.ohio-state.edu/LexicalSemantics/SaltonWongYang75.pdf.

[13] *Geneanet: Your family history.* URL: http://en.geneanet.org (visited on 12/02/2015).

[14] Jim Webber Ian Robinson and Emil Eifrem. *Graph Dattabases.* Ed. by Mike Loukides and Nathan Jepson. O’Reilly Media Inc., 2013. ISBN: 978-1-4493-5626-2.

[15] *JSON.* URL: http://www.json.org/ (visited on 02/15/2014).

[16] Lars Lundin. *GEDCOMP - GEDcom COMParison.* URL: http://www.lklundin.dk/gedcomp/english.php (visited on 12/15/2015).

[17] Eric Hatcher Michael McCandless and Otis Gospodnetic’. *Lucene In Action.* Ed. by Sebastian Stirling and Liz Welch. Manning Publications Co., 2010. ISBN: 978-1933988177.

[18] Lucía Pasarin Perea. “Analysis of alternatives to store genealogical trees using Graph Databases”. MA thesis. Universitat Politècnica de Catalunya, 2013.

[19] David M. W POWERS. “EVALUATION: FROM PRECISION, RECALL AND F-MEASURE TO ROC, INFORMEDNESS, MARKEDNESS & CORRELATION”. In: *International Journal of Machine Learning Technology* 2.1 (Dec. 15, 2011), pp. 37–63. URL: http://dspace2.flinders.edu.au/xmlui/bitstream/handle/2328/27165/Powers%20Evaluation.pdf?sequence=1.

[20] *Släktforskning, släktträd och släkthistorisk information på Ancestry.se.* URL: ancestry.se (visited on 12/20/2014).

[21] *The Neo4j Manual v1.8.3.* URL: http://neo4j.com/docs/1.8.3/index.html (visited on 12/15/2015).

[22] PETER WAYNER. “From Shared Resources, Your Personal History”. In: *The New York Times* (Apr. 22, 2004).

[23] Peixiang Zhao and Jiawei Han. “On Graph Query Optimization in Large Networks”. In: *Proceedings of The Vldb Endowment - PVLDB* 3.1 (), pp. 340 –351. URL: http://web.engr.illinois.edu/~hanj/pdf/vldb10_pzhao.pdf.

Series of Master’s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2015-432

http://www.eit.lth.se