Design and Implementation of a Middleware for Data Analysis of Social Networks

HIRAD CYRUS ASADI

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2008
Design and Implementation of a Middleware for Data Analysis of Social Networks

HIRAD CYRUS ASADI

Master’s Thesis in Computer Science (30 ECTS credits) at the School of Computer Science and Engineering Royal Institute of Technology year 2008
Supervisor at CSC was Joel Brynielsson
Examiner was Stefan Arnborg

TRITA-CSC-E 2008:019
ISRN-KTH/CSC/E--08/019--SE
ISSN-1653-5715
Abstract

This thesis deals with the design and implementation of a middleware that constructs network graphs for analysis purposes. We study how relational data can be collected, filtered and finally transformed into graphs—or network data. We introduce a modified version of the breadth-first search algorithm to discover outside nodes connected to subgraphs generated by Proximity, a network data filtering tool.

The results that were produced show that network data based on relational data can be created and that interesting social network graphs can be generated using different search strategies with defined predicates.
Referat

Design och implementation av ett mellanlager för dataanalys av sociala nätverk

Det här examensarbetet handlar om design och implementation av ett mellanlager som konstruerar nätverksgrafer för analyssyften. Vi studerar hur relationsdata kan samlas, filtreras och slutligen transformeras till grafer—eller nätverksdata. Vi introducerar en modifierad version av en bredden-först-sökning för att upptäcka utomstående noder anslutna till delgrafer genererade av Proximity, ett verktyg för att filtrera nätverksdata.

Resultaten som producerades visar att nätverksdata baserat på relationsdata kan skapas och att intressanta sociala nätverksgrafer kan genereras genom att använda olika sökstrategier med definierade predikat.
This Master’s thesis was a part of a EU-supported research project called HiTS/ISAC at the Swedish Defence Research Agency (FOI), Decision Support Department.

The vision of HiTS/ISAC is a more secure Europe through prevention of terrorism and organized crime. Superior situation awareness and cross-border interoperability are key enablers, leading to new technical and operational methods to work, train and co-operate across Europe. Today, information in databases at law-enforcement authorities is distributed across Europe. The information is not easily available to other authorities in Europe, especially not “on-line”.

The objective of HiTS/ISAC is to enable information analysis and data fusion from many different sources, through secure cross-border on-line group cooperation between authorities, in order to detect and provide early warnings for suspicious activities, be it communication between suspected criminals, or anomalous movement of persons, goods or money, etc.

I would like to thank my supervisor Doctor Pontus Svenson at FOI for his strong support and commitment during the entire Master’s project. I would also like to thank Professor Per Svensson at FOI for sharing his great knowledge and experience in this field. Finally, I would like to thank my supervisor Doctor Joel Brynielsson at KTH for his collaboration and knowledge. But most of all, I would like to thank my parents Hoshang and Lydia for always supporting me. Without them this thesis would not have been possible.

“They must find it difficult...Those who have taken the authority as the truth, rather than the truth as the authority.”
—Gerald Massey

“Unthinking respect for authority is the greatest enemy of truth.”
—Albert Einstein

To Parisa.
To Mahmoud Khosravi, rest in peace.
## Contents

1 Introduction .......................... 1
   1.1 Problem definition ................. 1
   1.2 Purpose of this thesis ............. 1

2 Background .......................... 3
   2.1 Definition of a social network ..... 3
   2.2 What is network data? .............. 4
   2.3 Types of networks .................. 5
   2.4 Data collection for networks ....... 5
   2.5 Filtering network data ............. 6
   2.6 Related work ....................... 7
      2.6.1 Intrusion detection systems .... 7
      2.6.2 Challenges with intrusion detection systems .... 9
      2.6.3 Similar tools to Proximity .... 9
   2.7 Outline of a general social network middleware solution .... 9

3 Designing a Middleware for Social Networks .... 11
   3.1 Design approach ................... 11
      3.1.1 Collecting data from different databases .... 12
      3.1.2 Using queries as a filtering method ......... 12
      3.1.3 Mapping objects to relational data .......... 12
      3.1.4 Transforming data to graphs and discovering nodes .... 12
      3.1.5 Exporting graphs .................. 13
      3.1.6 Design summary .................. 14
   3.2 Limitations ....................... 15

4 Gathering Network Data ................. 17
   4.1 Choosing network data ............. 17
      4.1.1 What data are important? ........ 17
      4.1.2 How much data should be extracted? .... 18
      4.1.3 Data integration ................. 18
   4.2 What relational data should be extracted? .... 18
      4.2.1 Node- and edge-semantics in relational data .... 19
# Creating Network Graphs  27

5.1 Why intermediate storage? ................................................................. 27  
5.2 Proximity and QGraph ................................................................. 27  
5.3 MySQL DB as a back-end ............................................................... 31  
5.4 Monet DB as a back-end ............................................................... 31  
5.5 Generating a unified graph from a Proximity subgraph set ................. 31  
5.6 Finding new network graph nodes with the BFS algorithm ................. 33  
5.7 Forming predicates and modifying the BFS algorithm ...................... 34  
5.8 Sharing graphs ............................................................................. 36  
5.9 JUNG ......................................................................................... 36  
5.10 GraphML .................................................................................... 36

# Implementing a Middleware for Social Networks  39

6.1 Implementation approach ............................................................... 39  
6.2 Proximity modifications ................................................................. 39  
6.3 Graphical user interfaces ............................................................... 40  
6.4 Obstacles ..................................................................................... 41  
6.5 Implementation summary ............................................................... 42

# Testing with Different Data Sets  43

7.1 Data sets ..................................................................................... 43  
7.2 Use-cases .................................................................................... 44  
7.2.1 Connecting to a MySQL and to a Monet database ...................... 44  
7.2.2 O/R mapping and creating a Proximity database ...................... 46  
7.2.3 Loading data into a Monet DB .................................................. 47  
7.2.4 Generating a subgraph based on filtered data with predicates ..... 47  
7.3 Results ....................................................................................... 49  
7.4 Comments .................................................................................. 61

# Conclusions  63

8.1 Object mapping and data transformation ....................................... 63  
8.2 Node discovery features ............................................................... 63  
8.3 Proximity & Monet DB ................................................................. 64  
8.4 Denodo Virtual DataPort .............................................................. 64  
8.5 Future improvements ................................................................... 65

Bibliography ....................................................................................... 67

Appendices ......................................................................................... 69

A SNA Middleware Memory Issues .................................................. 71

B SNA Middleware Memory Error Dislocation ................................. 77
List of Figures and Tables

Figures

Figure 2.1 An example of the concepts involved in a social network.
Figure 2.2 One-mode and two-mode networks.
Figure 2.3 Procedure for filtering data.
Figure 2.4 A General data mining ID system design proposal [Bass].
Figure 2.5 A General data fusion ID system design proposal [Bass].
Figure 3.1 Summarized design model.
Figure 4.1 Object/Relational-mapping.
Figure 5.1 Proximity (the Graph menu is a modification).
Figure 5.2 An example of the QGraph.
Figure 5.3 The subgraph section in Proximity.
Figure 5.4 Transformation from Proximity graph subsets to a connected graph (network graph).
Figure 5.5 The figure shows a subgraph to the left and new nodes that can be discovered to the right.
Figure 6.1 The figure shows modifications made to Proximity’s GUI.
Figure 6.2 The SNA Filter Manager—Proximity Data (left) and Proximity Client (right).
Figure 6.3 Programs developed.
Figure 7.1 The SNA Filter Manager—Proximity Data.
Figure 7.2 The SNA Filter Manager—Proximity Client.
Figure 7.3 The Proximity Manager.
Figure 7.4 A query in Proximity QGraph with condition “id=3” on Vertex1.
Figure 7.5 The Subgraph Manager.
Figure 7.6 Enron: QGraph query.
Figure 7.7 Enron: Default predicate.
Figure 7.8 Enron: Predicate 1.
Figure 7.9 Enron: Predicate 2, input 1.
Figure 7.10 Enron: Predicate 2, input 3.
Figure 7.11 Enron: Predicate 3, input 2.
Figure 7.12 Enron: Predicate 3, input 7.
Figure 7.14 BAUP: Default predicate.
Figure 7.15 BAUP: Predicate 1.
Tables

Table 2.1 A person table with attributes.
Table 2.2 The corresponding attribute similarity table to table 2.1.
Table 4.1 A people table with Id as a unique identifier.
Table 4.2 A table with sent e-mails.
Table 4.3 A table in 1-NF [UTA].
Table 4.4 People table in 2-NF (also in 3-NF) [UTA].
Table 4.5 E-mail sent table in 2-NF (also in 3-NF) [UTA].
Table 6.1 Programs developed.
Table 6.2 Modifications made in Proximity.
Table 7.1 Use-case 1: Connecting to a MySQL database.
Table 7.2 Use-case 2: Connecting to a Monet database.
Table 7.3 Use-case 3: O/R mapping.
Table 7.4 Use-case 4: Generate a Proximity DB.
Table 7.5 Use-case 5: Load a Proximity DB into Proximity.
Table 7.6 Use-case 6: Generating a subgraph with QGraph.
Table 7.7 Use-case 7: Generate a filtered subgraph.
Chapter 1

Introduction

Society consists of large complex social networks. These social networks are often quite hard to extract from data. This gives rise to a necessity to develop tools which can “easily” construct network data for analysis purposes. Social networks are interesting to study in various fields, e.g., many sociologists want to study interactions between different people on the Internet. Anti-terrorist analysts want to study terrorist organizations and so on.

By studying different networks we could discover different patterns and behaviors in various societies. This could, e.g., lead to further knowledge about life, its members and coexistence.

1.1 Problem definition

As information technology evolves the need for visualization of human sociological interactions increases. These interactions can be described as relations between different actors. Data are the source of information and much of human activities are stored in different databases around the world.

In many cases we would like to examine how some data are related to other data and visualize the results. If a person exists in a car registry, can that data be combined with, e.g., the person’s health insurance registry or tax registry and so on? Can social network data be created based on relational data sources and then visualized? In this thesis we—the author—will try to answer these questions with the focus on creation of networks.

1.2 Purpose of this thesis

The purpose of this thesis is to show a design and an implementation of a middleware which can create filtered network data based on relational data for analysis purposes. The middleware should be considered as ad hoc and will be integrated with Proximity—a network graph filtering tool [KDL].
The middleware should be able to perform certain default tasks:

- Map internal class structures to relational tables for data extraction.
- Transform relational data to the Proximity format (filtering tool).
- Create network graphs for analysis purposes.
- Provide exporting functions.

The results retrieved from the middleware will be used for analysis purposes with an analysis tool.
Chapter 2

Background

This chapter gives a general description of social networks and data collection. Related work will also be shown as well as a general solution proposal.

2.1 Definition of a social network

Social network analysis has been defined as: “Social network analysis is a part of sociology that was introduced in the 50’s as an extension of sociograms that describe relations in groups. Social network analysis is a collection of mainly statistical methods to support the study of communication relations in groups, kinship relations, or the structure of behaviour, to mention a few application areas. This methodology assumes that the ways members of a group can communicate affect some important properties of that group” [SST].

A social network consists of a social structure made of nodes that are connected with each other through different kinds of edges. The edges are in fact relations that define how one object is related to another. Figure 2.1 shows an example of the concepts involved in a social network.

![Figure 2.1. An example of the concepts involved in a social network.](image)

When analyzing social networks the focus is put on the relationships among social
entities and how these influence each other [WF]. There are formal methods to calculate different measurements in a network. The measurements are supposed to show valuable information about the structure of the social network.

2.2 What is network data?

Sociological data or conventional data consist of arrays of measurements, such as a table with people and personal data. The rows are subjects or observations and the columns indicate data (attributes) being measured. Network data can simply be described as data that has similarities to other data. For example, if two persons have the same attributes (gender, age and so on) then they are similar in the sense measured by those attributes.

We could represent the similarity with two nodes (the persons) and an edge between them (similarity indication). The major difference between conventional data and network data are that the latter focuses on actors and relationships rather than actors and attributes [HR]. Table 2.1 and 2.2 show the difference between conventional data and network data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Weight</th>
<th>Age</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>50</td>
<td>31</td>
<td>F</td>
</tr>
<tr>
<td>Alex</td>
<td>90</td>
<td>31</td>
<td>M</td>
</tr>
<tr>
<td>Fred</td>
<td>50</td>
<td>15</td>
<td>M</td>
</tr>
<tr>
<td>Hillary</td>
<td>55</td>
<td>31</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 2.2. The corresponding attribute similarity table to table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>Mary</th>
<th>Alex</th>
<th>Fred</th>
<th>Hillary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Alex</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Hillary</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

The similarity matrix [Skill] varies depending on how we define similarity and there are many types of similarity matrices. In the above example we showed similarity between attributes, that is, if two different persons have attributes that are the same, then for each of those attributes an integer is added in the similarity matrix.
2.3 Types of networks

There are many types of social networks and they are categorized by the nature of the sets of actors and the properties of the ties among them [WF]. Basically the modes of network—the number of sets of entities on which structural variables are measured [WF]—categorize a network. For social networks one-modes dominate. This should be considered as studying just a single set of objects whereas a two-mode or higher network consists of two or more types of objects. Many of the social network methods have been designed for low mode networks—one or two modes. Figure 2.2 shows an example of different network modes.

![One-mode and two-mode networks](image)

Figure 2.2. One-mode and two-mode networks.

2.4 Data collection for networks

Data can be collected for social networks using different techniques. We can collect data by questionnaires, interviews, observations, archival records and so on. The data used in this Master’s thesis should primarily be considered as different legacy data from archival records. This can, e.g., be transaction records from a bank, e-mails sent within a company and so on. Furthermore we should consider the network to be of multi-mode type since actor and event sets tend to become large in relational databases. This is because there are lots of different tables in large relational databases and each tuple can be considered as an actor or as an event. It must also be added that even though the relational database networks are of multi-mode type one seldom looks at all the actors and events at the same time.

Another aspect when examining legacy data sets is the study of longitudinal data [WF]. Basically longitudinal data describes how data are changed as time passes. When we study longitudinal data we try to answer two questions [WF]:

- How has the process changed over time?
- How well can the history predict the future?
For this Master’s thesis longitudinal data has no special purpose directly—even though it is interesting from an analysis point of view—but instead all kinds of relational data are considered.

### 2.5 Filtering network data

Social networks are usually very large scale networks, many times containing several millions of entities. Therefore visualization does not give the most useful information simply because it is hard to display an intuitive picture that shows millions of relations and objects. We could still calculate network measurements but we are more interested in visualizing some parts of the network. Thus, displaying network data calls for filtering mechanisms or else it would not be possible to manage the large data quantities.

Filtering tools provide these mechanisms and there are different kinds of methods to downsize data. This thesis will mainly focus on filtering data with queries—other, e.g., probabilistic methods could be used as well—because it is a simple and straightforward way to create subsets of large relational data volumes. Query-filtering is frequently used not only for social network analysis purposes but also generally when we, e.g., would like to query relational databases with SQL and retrieve a subset of a table. Figure 2.3 shows a procedure to filter network data to manageable sizes and then transforming the data to graphs. The result—a filtered network graph—is stored in some format and in some kind of a medium.

![Diagram of filtering data](image)

*Figure 2.3. Procedure for filtering data.*
2.6 Related work

Social network analysis is a broad research field and much research still needs to be done. Middlewares for social networks can be constructed in different ways, therefore the focus should be on whole systems that deliver data that may be transformed to social network data.

2.6.1 Intrusion detection systems

Intrusion detection—ID—systems can be considered as an advanced real-time application of data analysis which can be used for threat detection [Bass]. Social network data can be retrieved through similar techniques to ID systems in the sense of collection, transformation and analysis. Most of social network data come from legacy databases whereas ID systems mainly deal with real-time data using data fusion techniques, although data fusion may sometimes be of interest in social networks as well—e.g., advanced data fusion techniques combined with real-time based social network analysis tools.

The ID systems consist mainly of three layers: knowledge-, information- and data-layer [Bass]. ID systems use different data processing techniques as well as design interfaces but their layer based structure of information storage and processing can be considered as a general solution platform for middlewares that handle relational data. The ID systems can be grouped into two parts: data fusion and data mining.

Data fusion ID systems are used to combine data from different sources—or sensors. The information gathered can be used to make conclusions about events, activities and so on [HL]. Data mining is used to discover knowledge in “offline data”, mainly legacy databases or other old records. The data mining ID systems search for hidden patterns in old data whereas data fusion focuses on new data based on past data [HMS] [HK]. Figure 2.4 and 2.5 show the difference between the two ID systems.
Figure 2.4 demonstrates how different sensors pick up the raw data. The data is stored into data warehouses which are used for information extraction. The information is transformed and queried to later be analyzed by humans at the knowledge level.

Figure 2.5 demonstrates the same principle as figure 2.4 but instead we have different “refinement levels” which contain the data that have been processed. The object
base contains the object representation of the information and from there links are established to the knowledge representation level. The situation is always updated based on incoming data—newly discovered threats are added to the situation base [Bass]. Data fusion is typically used in real-time based information systems.

2.6.2 Challenges with intrusion detection systems

Both of these design solutions give a general perspective when designing information systems that use data from multiple sources for analysis purposes and there are also lots of challenges that come with them. One great challenge is to maintain data security in order to guarantee that no outsider retrieves information that can be used to harm other individuals [PP].

Another great challenge in the case of data fusion ID systems is the synchronization of data streams from multiple sources [Bass]. We would also like to include more semantics in, e.g., data extraction and retrieve relevant data without manually having to go through the metadata of the databases. A modern approach for integration of heterogeneous data sources is to use mediators which connect different systems—in this case DBMS's—together [SST]. Mediator systems work as automatic translators between different concepts, conventions, schemas and so on [WG].

2.6.3 Similar tools to Proximity

We found one interesting tool that was similar to Proximity. Graph-Tool [Peixoto] is a program written in C++ that supports statistical analysis of graphs. It has many features, most of them statistical algorithms that can be performed on graphs for data analysis purposes, but also parallel algorithms and community detection.

Graph-Tool also has a generic filtering mechanism which filters edges and vertices. We can filter graphs based on their directedness, we can reverse the edge direction and most importantly we can filter out vertices and edges based on their properties. Although Graph-Tool has many interesting features it does not, e.g., have the strong and fast query language support which characterizes Proximity.

2.7 Outline of a general social network middleware solution

There are no known existing general solutions to developing middlewares for social network analysis purposes. Basically, in order to create meaningful network graphs we have to study where data come from, how we can access data and how we can understand the data we have accessed. This divides our field of study into two parts which in the future need to be combined: distributed information access and semantic interpretation of data.

The data can be distributed in multiple formats and data sources, in an optimal environment we would like to have a general “interface” to access these sources as
if they were a single source, or at least try to integrate them. We would like to
have a mediator system (see section 2.6.2) with well defined protocols that allows
communication between different devices but at the same time allows transparency
[WG]. Once this has been established we would like some kind of interpretation
of data. This can, e.g., be done by semantic ontologies such as OWL [WC2] with
RDF [WC3] or perhaps by data mining techniques [JN]. Data need to be classified
in some way in order to extract information. After extraction, data need to be
transformed and processed using different filtering techniques and finally it can be
output as a graph. The middleware should be a layer of software whose purpose is to
mask heterogeneity and to provide a convenient programming model to application
programmers [WG].
Chapter 3

Designing a Middleware for Social Networks

In this chapter we propose a general design of a middleware for data analysis of social networks. Methods chosen, suggestions made and so forth should be considered as our choice of design if no direct references to, e.g., other similar design patterns are made.

3.1 Design approach

The main approach is based on the layer concept which was discussed earlier in section 2.7. This concept is chosen due to the fact that it provides a very good abstraction which eases implementation tasks. If we study figure 2.4 and 2.5 we will see that each layer (knowledge-, information- and data-layer) can be treated as a single module, hence adding and removing modules becomes easy. The focus of this thesis is mainly on the information- and data-layer which can be considered as the middleware.

Recalling the purpose of this thesis, we obtain a specification for the middleware. The specification divides our complete design into sub-goals which need to be reached separately.

The following sub-goals—or concepts—are proposed for the complete design:

- Collecting data from different databases
- Using queries as a filtering method
- Mapping objects to relational data
- Transforming data to graphs and discovering nodes
- Exporting graphs

Each item will be discussed in the forthcoming sections.
3.1.1 Collecting data from different databases

The data come from different sources, hence a common view is needed since it requires a large mental effort for the analysts to work simultaneously with different connections and interpreting data on-the-fly.

The view should consist of single tables or sometimes joint tables from multiple data sources. The sources can, e.g., be XML files [WC1], different DBMSs [Wiki3] and so on. The common view enables us to access data as if it was one single database, thereby reducing parallel work.

3.1.2 Using queries as a filtering method

Working with social networks means dealing with enormous amounts of data. Different filtering mechanisms must be introduced in order to reduce the volume of data [HMS]. The proposed and trivial solution to reduce relational data size is to use queries to retrieve subsets.

Queries enable us to select particular rows and columns, thus reducing the volume. The first level of queries are done with SQL—“selection of nodes and edges”—towards a database, the second level of queries are done with Proximity QGraph [KDL] to retrieve subgraph sets.

3.1.3 Mapping objects to relational data

Social network graphs consist of nodes and edges—or objects—and these elements can be considered as the foundation which any future application must rely on. Since data is derived from relational databases the term object lacks definition—there is no definition that tells us what data are nodes or edges.

The proposed solution is to create a model which can map internal representations of nodes and edges to relational data. This is needed because data must be extracted in a manner that allows nodes and edges to be represented in computer memory. The model can also be considered as a filtering schema since copying all of the relational data to the primary memory is impossible due to memory size limitations.

3.1.4 Transforming data to graphs and discovering nodes

Before the data can be stored into Proximity as a graph [Biggs] it needs to be transformed. The first transformation step is to convert the data from the mapped relational model to a text file format which can be read by Proximity. Proximity has its own internal storage format, but that format can not be used for graph algorithms without extensive work. Proximity transforms the text files to an XML [WC1] based format which it uses for data import. Proximity itself uses the Monet DB [CWI] to store the loaded data.

It is not always trivial how to store a graph but two basic methods are considered in this case. The first method is to store a graph by an adjacency matrix [BG].
This method allows us to perform matrix operations quickly, but, e.g., retrieving all neighbors requires us to go through the entire matrix. Therefore this storage method is not appropriate for sparse graphs since it has a total time complexity of $O(n^2)$ [BG].

The second method—also the method that is used in this thesis—is to store a graph using linked lists [BG]. A linked list representation of the graph contains only the nodes that are present in an edge, thus in general decreasing the information amount. This method is chosen mainly because many graph traversing algorithms requires us to go through the neighbors of a node. For small graphs the time complexity is $O(n \cdot m)$ [BG], where $n$ is the number of nodes and $m$ the number of edges ($m < n$).

Many times we would like to generate subgraphs and then expanding them based on some conditions. This means that if we compare the subgraph with the original main graph we can decide to expand the subgraph if certain conditions are fulfilled. The proposed method to discover new nodes is to use the breadth-first search algorithm [BG] and modify it so it can use predicates—or conditions—that need to be met if a new node is to be added. This method is fairly straightforward since the BFS algorithm is a known and approved graph search algorithm.

### 3.1.5 Exporting graphs

The graph must be read by other programs, thus the output format needs to follow a standard. The proposed solution is to store the graph in JUNG [MDT] and GraphML [GPG] since these formats are frequently used. JUNG is for the internal representation of the graph and GraphML is for exporting the graph to an XML version.
3.1.6 Design summary

The previous design features can be summarized into a single model. Figure 3.1 shows from top to bottom how relational data is transformed into graphs.

![Diagram showing the summarized design model.](image)

**Figure 3.1.** Summarized design model.
3.2 Limitations

There are different design limitations that need to be explained. The first limitation is how data is selected. We must have in mind that the relational data must make some kind of sense, that is, we have to interpret some tables as nodes and some as edges—which data need to be selected? These selections are done manually when creating the object-relational model. The previous design concept—the common view—only integrates data without any known or predefined semantics other than which tables and table data we select.

The transformed data is stored into the “vertical database” Monet DB [CWI] which is used by Proximity [KDL]. The second limitation is how Proximity stores subgraphs. Proximity stores subgraphs as sets of object-link connections instead of a whole connected graph. This storage format is not easy to work with if we want to apply different graph algorithms. Each subgraph set is stored as an independent data structure instead of one complete structure—which is what we want. Fortunately this can be overcome by exporting the subgraph and then accumulating the exported structure—which is XML based—into an internal data structure that is more suited for graph algorithms.
Chapter 4

Gathering Network Data

This chapter describes how the raw data should be selected and how a model could be adapted to map relational data to objects. In later stages Proximity is used as a filtering tool to generate subgraphs, thus the model needs to act as a transformation bridge between relational data and Proximity’s data format for import.

4.1 Choosing network data

Network data are used for creating graphs, but the process from relational data to network data is rather abstract. What data are important? What data should be extracted? How much data should be extracted? These are some questions that need to be answered in order to fully understand how relational data can be visualized as graphs.

4.1.1 What data are important?

The data that are important are basically the objects and links of interest—or the elements that we would like to study. We could, e.g., study e-mails sent between different people. The Enron scenario is an example of this, see section 7.1 In this scenario the network graph contains objects representing people and edges between objects representing e-mails sent back and forth. This principle is not always trivial but in most cases we can model concepts as objects and links between objects. The important data depend on the study, but first we must always classify data [HK] in order to gather knowledge.

Advanced classification models [HK] can be used to predict unknown data but in this thesis the concept of important data is defined by the user. It is also important to remember that we could produce different kinds of visualizations, especially between the same objects. For example, an edge may represent an e-mail sent, but another—maybe parallel—edge could represent an event—e.g., person A fired person B. Using different visualizations we could retrieve interesting information that could ease knowledge discovery.
CHAPTER 4. GATHERING NETWORK DATA

4.1.2 How much data should be extracted?

How much data that should be extracted depends on the visualization task. Too much data would give much information, but the cost could be information overload, that is, a human would not easily be able to analyze the visualized network. But sometimes this does not have to be the case. We could extract large volumes of data and then apply smart analytic methods to create small views and visualize the views.

On the other hand, too little data would produce insufficient results, causing the analysis to be incomplete. Basically the line is drawn by the capabilities of the analysis tools used by the analyst and therefore each study case needs to be treated separately.

In this Master’s project we would like to acquire manageable data volumes for visualization purposes, but we would also like to apply different social network algorithms—not covered in this thesis. The size of the data that is managed is determined by the time complexity of the social network algorithms. We must be able to downsize data so that we can apply the social network algorithms that we are interested in.

4.1.3 Data integration

We would like to integrate different database sources into a single intermediate storage. This would allow us to process data easier without having to deal with multiple connections. There are different tools to overcome this problem, one of them is Denodo Virtual DataPort (DVDP) [DT] which is considered a prerequisite in this Master’s project.

DVDP is a tool which allows us to integrate different database sources into a common view. This view basically allows us to see all the database sources as one database. Therefore it is a practical tool for data integration. This tool is not the focus of this thesis but it is considered for the whole application since it can be used to integrate different data sources. DVDP is an example of a mediator [WG] discussed in section 2.6.2.

4.2 What relational data should be extracted?

Relational data are structured into data sets—tables with rows and columns—and there is no predefined structure which automatically can be interpreted as nodes and edges—which we need to produce network graphs. Therefore we need to discover knowledge in tables by using a classification model [HK] which can map nodes and edges to relational data.
4.2. WHAT RELATIONAL DATA SHOULD BE EXTRACTED?

4.2.1 Node- and edge-semantics in relational data

A relational data set can be classified—or mapped—as objects, see figure 4.1. The objects encapsulate relational data depending on the mapping. However, there is no trivial way to classify network nodes and edges in a general relational data set. To solve this problem we can classify nodes and edges as objects and then map the relational data into these objects [Ambler]. The object model works as a transformation bridge between relational data and Proximity’s internal format, therefore the internal format for import regulates how the model should be constructed.

![Object/Relational-mapping](image)

Figure 4.1. Object/Relational-mapping.

4.2.2 A classification model: mapping objects to relational data

Proximity enables us to import data from text files. The text files are processed and transformed to an XML format for data import. The Proximity data format consists of four text files: two specification files and two data files, one specification and one data file for the objects and the same with the links. The formats are shown below:

Object/link specification file format:

```
element1 name
    attribute1 DATA TYPE
    attribute2 DATA TYPE
    attribute3 DATA TYPE
    ...
element2 name
    attribute1 DATA TYPE
    attribute2 DATA TYPE
    attribute3 DATA TYPE
    ...
```

Object data file format (each row):
```
id attribute1 attribute2 attribute3 ...
```

Link data file format (each row):
id objectid1 objectid2 attribute1 attribute2 attribute3 ...

The id fields are of numerical type.

**Mapping structured relational data**

The Proximity format for import of data regulates how we should structure relational data. The regulations are:

- Relational data need to be identified as unique objects.
- A link representation must be present, see, e.g., table 4.2.

This is true since each Proximity data file format specifies a unique identifier as the first column. The link elements refer to two objects by their identifiers \( \text{objectid1} \) and \( \text{objectid2} \), thus enough data is stored to represent links.

Relational data are often stored in the third normalized form—although not proven. Table 4.1 shows a people table where each person is identified by an “Id” and table 4.2 shows e-mails sent by persons from table 4.1—both tables are normalized [UTA]. Note that the identifying column does not generally need to be of numerical type in relational databases.

<table>
<thead>
<tr>
<th>Table 4.1. A people table with Id as a unique identifier.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.2. A table with sent e-mails.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>From</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
4.2. WHAT RELATIONAL DATA SHOULD BE EXTRACTED?

This normalized form also happens to be the format which Proximity uses as data input, see earlier in this section. Clearly we can see that if the relational data is stored in this format, then mapping becomes rather trivial. The mapping is simply:

(Relational data → Proximity format)

Object data file:
Name → Name (attribute1)
Weight → Weight (attribute2)
Age → Age (attribute3)
Gender → Gender (attribute4)

Link data file:
Subject → Subject (attribute1)
Message → Message (attribute2)
Sent → Sent (attribute3)

The “Id” fields are omitted simply because not every identifying column is guaranteed to be of numerical type—the necessary format for Proximity—and need to be generated afterwards. The “From” and “To” fields follow the same principle.

Mapping unstructured relational data

The mappings shown so far are trivial, but how do we map tables that are not normalized? The easiest way is to normalize those tables. This procedure can be done with DVDP [DT]. After normalization we can deal with tables as whole objects which enables us to interpret some tables as nodes and some as edges. The tables on the next page show an example of how this can be done.
### Table 4.3. A table in 1-NF [UTA].

<table>
<thead>
<tr>
<th>Name</th>
<th>Weight</th>
<th>Age</th>
<th>Gender</th>
<th>Email_To</th>
<th>Subject</th>
<th>Message</th>
<th>Sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>50</td>
<td>31</td>
<td>F</td>
<td>Hillary</td>
<td>Hi!</td>
<td>Hello how are you...</td>
<td>2007-08-03 20:00:31</td>
</tr>
<tr>
<td>Alex</td>
<td>90</td>
<td>31</td>
<td>M</td>
<td>Fred</td>
<td>Re: Meeting soon</td>
<td>Sorry busy, but I can...</td>
<td>2007-04-03 17:17:41</td>
</tr>
<tr>
<td>Fred</td>
<td>50</td>
<td>15</td>
<td>M</td>
<td>Alex</td>
<td>Meeting soon</td>
<td>Remember to schedule...</td>
<td>2007-04-03 12:10:53</td>
</tr>
<tr>
<td>Hillary</td>
<td>55</td>
<td>31</td>
<td>F</td>
<td>Mary</td>
<td>Re: Hi!</td>
<td>Im fine how are you...</td>
<td>2007-08-03 22:30:01</td>
</tr>
</tbody>
</table>

Table 4.3 is normalized to table 4.4 and 4.5 (surrogate keys have been introduced in order to fully identify a person):

### Table 4.4. People table in 2-NF (also in 3-NF) [UTA].

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Weight</th>
<th>Age</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Mary</td>
<td>50</td>
<td>31</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>Alex</td>
<td>90</td>
<td>31</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>Fred</td>
<td>50</td>
<td>15</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>Hillary</td>
<td>55</td>
<td>31</td>
<td>F</td>
</tr>
</tbody>
</table>

### Table 4.5. E-mail sent table in 2-NF (also in 3-NF) [UTA].

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Subject</th>
<th>Message</th>
<th>Sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Hi!</td>
<td>Hello how are you...</td>
<td>2007-08-03 20:00:31</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Re: Hi!</td>
<td>Im fine how are you...</td>
<td>2007-08-03 22:30:01</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Meeting soon</td>
<td>Remember to schedule...</td>
<td>2007-04-03 12:10:53</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Re: Meeting soon</td>
<td>Sorry busy, but I can...</td>
<td>2007-04-03 17:17:41</td>
</tr>
</tbody>
</table>
4.2. WHAT RELATIONAL DATA SHOULD BE EXTRACTED?

The object model

The class below (ProximityElement) in pseudo code shows the basic outline of the object model:

```java
class ProximityElement
{
    /* ...Member variables */

    /* Constructor for nodes */
    ProximityElement(node_name, node_attributes,
                     SQL_condition, node_id)

    /* Constructor for edges */
    ProximityElement(edge_name, edge_attributes, SQL_condition,
                     edge_from, edge_to, node_from, node_to)

    /* ...Get() & Set() methods */
}
```

The _name field is the name of the relational table and the _attributes field a list of table attributes. The SQL_condition is for downsizing the relational data to a manageable size. The node_id field indicate which column is unique in the relational table (primary id). The edge fields edge_from and edge_to correspond to the node fields node_from and node_to. The edge_ fields are columns in the edge table and the node_ fields columns in the node table(s).

Model processing

The model is processed by retrieving the node- and edge-information in the classes and constructing SQL-queries which are executed. The queries are constructed in the following way (pseudo code):

```
[Object queries]
/****************************/
/* Get object data */
/****************************/

id = ProximityElement.getObjectId
/* Object attributes (list) */
attributes = ProximityElement.getAttributes
objecttable = ProximityElement.getName
condition = ProximityElement.getCondition
```
/* The object query */
SQL_QUERY =
SELECT id, attributes
FROM objecttable
WHERE condition

[Link queries]

/******************/
/* Get link data */
/******************/

/* Id in object table 1 */
objectid1 = ProximityElement.getObjectid1
/* Id in object table 2 */
objectid2 = ProximityElement.getObjectid2
/* Conditions of object1 */
objectCondition1 = ProximityElement.getObjectCond1
/* Conditions of object2 */
objectCondition2 = ProximityElement.getObjectCond2
/* Link attributes (list) */
attributes = ProximityElement.getAttributes
object1table = ProximityElement.object1table
object2table = ProximityElement.object2table
linktable = ProximityElement.getName
condition = ProximityElement.getCondition

/* The link query */
SQL_QUERY =
SELECT l.objectid1, l.objectid2, l.attributes
FROM linktable l, object1table o1, object2table o2
WHERE o1.id = l.objectid1
AND o2.id = l.objectid2
AND objectCondition1
AND objectCondition2
AND condition

We check for empty attributes and conditions when we generate the queries and we also make sure that the condition attributes are referred to the corresponding tables. The results are stored along with generated identifiers (Proximity identifiers). We now have enough information to create nodes and edges, therefore it is only a matter of implementation and is not discussed in the design. Note that we separate the SQL queries into two different sets: object set and edge set. This is done because
4.2. WHAT RELATIONAL DATA SHOULD BE EXTRACTED?

each set needs unique Proximity identifiers, if this was not the case the link query would almost be sufficient by itself.

Strengths and weaknesses with the model

The strength of the model is that it is very easy to map nodes and edges—basically we only need the table names and identifiers. This also allows the model to be built manually from, e.g., a graphical interface. The main weakness is that the model does not automatically map to a data set. There is no object schema available which can be mapped to a database schema directly without any knowledge of the existing data.

For future work a better mapping schema is needed. The schema should be able to map patterns—node and edge patterns—perhaps indicated by foreign key relationships in relational data. In this way user input is only limited to choosing an existing data set, the network data will be constructed automatically.
Chapter 5

Creating Network Graphs

In this chapter we discuss why intermediate storage is important. We also discuss the transformation process from Proximity data to network data. We will show how new network nodes can be discovered. We assume the relational data has been processed with the model introduced in section 4.2.2 and stored in the Monet DB which is used by Proximity. We also assume a QGraph query has been defined and executed.

5.1 Why intermediate storage?

The purpose of intermediate storage is almost totally based on the limited amount of memory. Whole networks are very large and can not be processed easily without data reduction. Data is stored into back-ends and these can be managed by different DBMS’s. For this thesis MySQL DB and Monet DB are used as back-ends (prerequisite). Another important aspect is that it is hard to deal with multiple database connections and performing table joins and data manipulation on the fly.

5.2 Proximity and QGraph

Proximity is an analyst tool which enables us to filter data—network graphs—and analyze different patterns, see figure 5.1. The QGraph works as a tool for querying the data set. It is basically a graphical query tool which enables us to create graphical nodes and edges with conditions and settings. The graphical query is executed and the results are stored as subgraph sets, see figure 5.2 and 5.3.
The network data is stored as the “root container” in Proximity—or “main container”. Proximity allows us to browse the objects and links that exist in these containers. We can also see the attributes that exist and their values. It is possible to analyze schemas and through different selections get different information. There are many functionalities available that allow us to discover knowledge in relational data.
5.2. PROXIMITY AND QGRAPH

The QGraph is used to downsize data and retrieve interesting patterns through graphical queries. We define a query in the QGraph—see figure 5.2—and then we execute it. The results are stored as subgraph sets shown in figure 5.3. The QGraph allows us to define complex relations between objects and we can also set conditions to narrow the match.

Figure 5.2. An example of the QGraph.
The results are stored in a sub-container—chosen by us—where each matched object-link combination defined in QGraph is stored in its own set as a “subgraph” identified by a number, see Figure 5.3. A subgraph could, e.g., consist of three matched elements: two objects and one link, or anything else matched by the QGraph query.
5.3  MySQL DB as a back-end

The MySQL DB [MP] is a powerful open source DBMS with strong support of much of the SQL language. It is widely used as a web database but also performs very well when integrated with different applications. We use MySQL as a back-end for intermediate storage of relational data.

5.4  Monet DB as a back-end

The Monet DB [CWI] is used with Proximity, mainly for performance reasons. SQL databases assume that data should be accessed in a certain way, but for the queries generated by Proximity, this is not optimal and therefore the Monet DB is needed. A traditional row-oriented relational database access every attribute/column in a table when a SQL query is executed, even though we only need some of the columns. A vertical column-oriented database, however, access only the columns which were defined in the query [Wiki4]. A vertical database stores all the values of an attribute in a separate file with unique identifiers to match other attribute values in other files, thus we can get the same type of information stored in row-oriented databases [Wiki4].

Vertical databases are more suited for query intensive applications—reading data—whereas row oriented databases are more suited for transaction purposes—writing data [Wiki4]. Column data which are of the same type is much easier to compress compared to row data [Wiki4]. That is why the actual disk reads required to read the columns defined in a query may be far less than the size of the raw data in these columns [Wiki4]. Many network graph queries are ad hoc and therefore hard to optimize with row oriented databases. Therefore column oriented databases are more suited since they provide faster read capabilities.

5.5  Generating a unified graph from a Proximity subgraph set

QGraph generates subgraph sets that match a query’s structure, but these subgraph sets are not connected in any way. We need a connected graph that does not contain any duplicate information. Figure 5.4 shows subgraph sets in Proximity (to the left) and the final type of output that we would like to have (to the right).
To construct connected graphs we need a container which can map a unique node to a list of neighbors connected to it. The proposed container—the linked list—has been discussed in section 3.1.4 with its advantages and disadvantages. The subgraph set can be exported to an XML format which is similar to the main format for importing data into Proximity. An example of the XML format is shown below:

```
<ITEM SUBG-ID="0" ITEM-ID="2" ITEM-TYPE="O" NAME="Vertex2"/>
<ITEM SUBG-ID="0" ITEM-ID="3" ITEM-TYPE="Q" NAME="Vertex3"/>
<ITEM SUBG-ID="0" ITEM-ID="3" ITEM-TYPE="L" NAME="Edge1"/>
<ITEM SUBG-ID="0" ITEM-ID="5" ITEM-TYPE="L" NAME="Edge3"/>
<ITEM SUBG-ID="1" ITEM-ID="12" ITEM-TYPE="O" NAME="Vertex12"/>
<ITEM SUBG-ID="1" ITEM-ID="7" ITEM-TYPE="O" NAME="Vertex7"/>
<ITEM SUBG-ID="1" ITEM-ID="4" ITEM-TYPE="L" NAME="Edge4"/>
<ITEM SUBG-ID="1" ITEM-ID="2" ITEM-TYPE="L" NAME="Edge2"/>
...
<ATTRIBUTES>
  ...
</ATTRIBUTES>
...

<LINK ID="2" O1-ID="1" O2-ID="7"/>
<LINK ID="3" O1-ID="1" O2-ID="2"/>
<LINK ID="4" O1-ID="1" O2-ID="12"/>
<LINK ID="5" O1-ID="1" O2-ID="3"/>
...
```

From this we can store each link and object uniquely in our linked list container. We store references to items in hash tables which are used for matching the right objects with the right links and vice versa, therefore we can construct a connected graph.
5.6. FINDING NEW NETWORK GRAPH NODES WITH THE BFS ALGORITHM

Finding new network graph nodes with the BFS algorithm

Many times when we study network data we would like to discover additional information, that is, we would like to discover new nodes that are somehow related to the existing nodes—the subgraph. There are different ways this can be done but we choose the BFS algorithm—Breadth First Search—since it is well-documented and is easy to modify. Our solution can be designed with the DFS algorithm—Depth First Search—as well, or other graph search algorithms, but we have chosen the BFS algorithm mainly for its simplicity and optimal performance on unweighted graphs.

The BFS algorithm has a time complexity of $O(|V| + |E|)$, where $V$ is the vertex (node) set and $E$ is the edge (link) set. The below pseudo code shows a general BFS algorithm:

Let $G$ be a graph

$Q \leftarrow$ QUEUE

$\forall \nu$ vertex, $\nu \in G$

Paint $\nu$ white

Paint start vertex $\phi$ gray, $\phi \in G$

$Q \leftarrow Q + \phi$

WHILE $Q \neq \emptyset$

$\nu \leftarrow \text{first}(Q)$

$\forall \delta$ white neighbors of $\nu$

Paint $\delta$ gray

$Q \leftarrow Q + \delta$

Paint $\nu$ black

The BFS algorithm allows us to traverse through the entire network graph and find new nodes but there need to be conditions that must be fulfilled in order to expand the subgraph set or else the data set would grow enormous due to expansion. We can call these conditions “predicates”, that is, a predicate—or rule—has to be fulfilled if a new undiscovered node is to be added to the subgraph set. Figure 5.5 shows an example of this, a subgraph and new nodes that can be discovered by some predicate.
5.7 Forming predicates and modifying the BFS algorithm

There can be many types of predicates depending on the study, but we choose three predicates only as an example. Before we mention them we explain the general outline of the new algorithm that we introduce in order to discover new nodes. This new algorithm is a modification of the BFS algorithm which was discussed in the previous section. The following pseudo code shows the general outline of the new algorithm:

Let $G$ and $S$ be graphs, $S \subset G$
Let $\rho$ be a predicate
$Q \leftarrow$ QUEUE

\[
\forall \nu \text{ vertex, } \nu \in G \\
\quad \text{Paint } \nu \text{ white}
\]

\[
\forall \phi \text{ vertex, } \phi \in S \\
\quad \text{Paint } \phi \text{ gray} \\
\quad Q \leftarrow Q + \phi
\]

WHILE $Q \neq \emptyset$
\[ \nu \leftarrow first(Q) \]
\[ \forall \delta \text{ neighbours of } \nu \\
\quad \text{IF } evaluate(\rho, \delta) \text{ AND } \delta \text{ is white} \\
\quad \text{Paint } \delta \text{ gray} \\
\quad Q \leftarrow Q + \delta \\
\quad S \leftarrow S + \delta \\
\quad \text{Paint } \nu \text{ black} \]

RETURN $S$
We will now discuss why this algorithm works. To start with this algorithm requires that we have two graphs $G$ and $S$, $S \subseteq G$ ($S$ has been generated by Proximity). $G$’s vertices are painted white according to the BFS algorithm. We put all vertices of $S$ in the queue and paint them gray since the $S$ graph already exists and we want to expand it. Then we look for all nodes and try to evaluate the predicate on them. If the predicate evaluates true and the node is white, then the node is processed like the BFS algorithm and is added to the subgraph set. This will make sure that the predicate criteria is fulfilled on each new node that we want to discover, hence we can discover new nodes based on predicates. Finally we return the expanded subgraph.

The time complexity for this new algorithm is still $O(|V| + |E|)$. If the predicate evaluation method requires heavy processing then the time complexity will increase.

**Predicate 1: “Links between the subgraph nodes”**

This predicate returns true if there are edges between subgraph nodes which are not included in the default subgraph generated by Proximity.

**Predicate 2: “Distance from a subgraph node to a root graph node”**

This predicate returns true if a node $\nu$, where $\nu \in G \setminus S$, is on a distance $x$ from $S$. That is, if the distance from a node in $S$ to $\nu$ is $\leq x$, $x \in \mathbb{Z}$ and $x \geq 0$. If $x$ is 1, then all undiscovered nodes are valid if they are not present in $S$ and are on a distance $\leq 1$ from $S$.

**Predicate 3: “Links from a root graph node to the subgraph”**

This predicate returns true if a node $\nu$, where $\nu \in G \setminus S$, has $x$ links to nodes in $S$, $x \in \mathbb{Z}$ and $x \geq 0$. If $x$ is 2, then all undiscovered nodes are valid if the number of links from $\nu$ to nodes in $S$ are $\geq 2$.

**Strengths and weaknesses with the modified BFS algorithm**

The main strength of the modified BFS algorithm is that it follows the search standards of the BFS. This will ensure that each node put into the queue will be processed, therefore modification is rather trivial. The main weakness of the modified BFS algorithm is the evaluation part. This evaluation is dependent on the data that needs to be collected for the predicates. For small number of predicates only a small amount of data needs to be collected as we iterate, but for many predicates we would perhaps have to gather a large amount of data.

For future work we could consider evaluating complex logical expressions that involve multiple predicates, but in this thesis we only consider single executions of predicates.
5.8 Sharing graphs

Networks provide a great source of knowledge in understanding relations between different entities. The graph data that is produced should easily be exported to formats that can be interpreted by other programs. In this thesis we consider two of these formats, JUNG [MDT] and GraphML [GPG].

We have chosen JUNG as an internal format for representation of graphs since it is widely used in many visualizing applications. We choose the GraphML format as an external format for graph representation for the same reasons, this format is considered a standard format for graphs.

5.9 JUNG

The JUNG—Java Universal Network/Graph Framework—format has been developed by a research team for the purpose to provide “a general, flexible and powerful API for manipulating, analyzing and visualizing graphs and networks in Java” [MDT].

There are many other applications/libraries similar to JUNG, but we chose JUNG mainly for its strong algorithm support and the ability to be easily integrated with other existing tools.

5.10 GraphML

The GraphML format is an XML based format which is a successor to the GML format [Infosun]. The GraphML format stores objects and links as well as corresponding attributes. It is easy to use and provides complete information about the graph. We can see an example of the GraphML format on the next page, an undirected graph without edge attributes.
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <graph edgedefault="undirected">
    <!-- data schema -->
    <key id="name" for="node" attr.name="name" attr.type="string"/>
    <key id="gender" for="node" attr.name="gender" attr.type="string"/>

    <!-- nodes -->
    <node id="1">
      <data key="name">Jeff</data>
      <data key="gender">M</data>
    </node>
    <node id="2">
      <data key="name">Ed</data>
      <data key="gender">M</data>
    </node>
    ...

    <!-- edges -->
    <edge source="1" target="2"></edge>
    <edge source="1" target="3"></edge>
    ...
  </graph>
</graphml>
Chapter 6

Implementing a Middleware for Social Networks

In this chapter we discuss the implementation decisions and the effects they had. The softwares that were used for this Master’s thesis were Denodo Virtual DataPort 4.0 [DT], Proximity 4.2 [KDL], Monet DB 4.6.2 [CWI], MySQL 5.0.41 [MP] and Java 1.6 [ST].

6.1 Implementation approach

Our approach was mainly to build the functionalities discussed in the previous part into a middleware. The middleware was constructed with Java [ST] since Java code can be executed on other operating systems and there are also many open source libraries for graph visualization available for Java.

To start with we developed the object model class, see section 4.2.2. Later we built a class for supporting the model processing, see section 4.2.2. The model processing class also created the XML database—network data—which was used by Proximity to generate subgraphs. We needed a connection between relational data and data processed by the middleware so therefore we built a back-end to support basic SQL commands. In our study we used MySQL [MP] as a DBMS but other DBMS’s could be used since our back-end followed an abstract class.

Furthermore, the modified BFS algorithm with predicate support needed to be incorporated in Proximity and therefore a section of Proximity was changed in order to support this function. A network could be reduced by filtering mechanisms in Proximity, a subgraph could be expanded by node discovery features and finally the results could be exported to GraphML.

6.2 Proximity modifications

The modifications made to Proximity consisted of changing the QGraph classes to support newer versions of Java. This was done in collaboration with the Knowledge
Discovery Laboratory [KDL], the creators of Proximity. Furthermore the GUI was changed to support the node discovery functionalities which can be seen in figure 6.1. Also Proximity’s exporting functionality was changed to generate subgraphs with additional attribute information.

6.3 Graphical user interfaces

We needed a general interface to access both Proximity and also to manage relational data, that is, to define objects and links. This general interface can be seen in figure 6.2. Figure 7.3 shows the interface for defining objects and links.
6.4 Obstacles

There were many obstacles during development. The first obstacle was to identify the problem with Proximity not being able to run queries on newer Java versions. Fortunately we could solve this problem with some professional help from the crew at KDL. Another obstacle was the use of Denodo Virtual Dataport as a back-end without any intermediate storage. This idea had to be discarded since our design was heavily dependent on the SQL language. The major obstacle during development was retrieving the different graph sets $G$ and $S$ in Proximity. They were needed to implement the modified BFS algorithm as we can recall from section 5.7. We had to modify Proximity once more to make sure that we could retrieve both the generated subgraphs—which were stored as “containers” in Proximity—and also the main/root graph. Fortunately we could export both graphs to an XML format after some modifications and then read them again into internal structures—linked lists.
6.5 Implementation summary

Programs developed (see table 6.1):

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNA Filter Manager</td>
<td>Interface to connect to a MySQL DB, Monet DB and load Monet data, see figure 6.3</td>
</tr>
<tr>
<td>Proximity Manager</td>
<td>Map relational data to the “nodes and edges” concept and create a Proximity DB, see figure 6.3</td>
</tr>
<tr>
<td>Subgraph Manager</td>
<td>Generate GraphML subgraphs based on predicates, see figure 6.3</td>
</tr>
</tbody>
</table>

![Figure 6.3. Programs developed.](image)

Modifications made (see table 6.2):

<table>
<thead>
<tr>
<th>Name</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modification 1</td>
<td>Support to run queries in newer Java versions</td>
</tr>
<tr>
<td>Modification 2</td>
<td>Support to export subgraphs and their link/attribute information</td>
</tr>
<tr>
<td>Modification 3</td>
<td>Graph menu added to the GUI to include the subgraph manager, see figure 6.1</td>
</tr>
</tbody>
</table>
Chapter 7

Testing with Different Data Sets

In this chapter we show tests of our middleware. The tests were carried out using different data sets and they were made from a user perspective using the graphical user interface. We also defined use-cases for all interesting scenarios involving our middleware.

7.1 Data sets

For testing we chose two different data sets and we started with a unique Proximity query for each data set. The following data sets were chosen:

- Enron (tested with a simple Proximity query)
- Books about US politics (tested with an advanced Proximity query)

The Enron data set contains massive amounts of e-mails that have been sent back and forth between different people in the organization, edges represent e-mails sent between persons [Cohen]. “Books about US politics” [Newm] contains a network of books about US politics that have been sold by Amazon.com. Edges between books represent frequent co-purchasing of books by the same buyers.

We tested all use-cases for each data set and for the last use-case (see section 7.7) we tested each predicate (see section 5.7) on the data set. We also tested a default predicate which only returned the subgraph without any new node discovery features.
7.2 Use-cases

The use-cases in this section demonstrate how the functionalities that have been developed should be used.

7.2.1 Connecting to a MySQL and to a Monet database

Table 7.1. Use-case 1: Connecting to a MySQL database.

<table>
<thead>
<tr>
<th>Use-case 1: Connecting to a MySQL database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: the user connects to a MySQL database, see figure 7.1</td>
</tr>
<tr>
<td>Input: the user types in login information</td>
</tr>
<tr>
<td>Expected output: a connection</td>
</tr>
<tr>
<td>Constraints: a MySQL database must be running</td>
</tr>
</tbody>
</table>

Figure 7.1. The SNA Filter Manager—Proximity Data.
### Table 7.2. Use-case 2: Connecting to a Monet database.

<table>
<thead>
<tr>
<th>Use-case 2: Connecting to a Monet database</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> the user connects to a Monet database, see figure 7.2</td>
</tr>
<tr>
<td><strong>Input:</strong> the user types in login information</td>
</tr>
<tr>
<td><strong>Expected output:</strong> a connection</td>
</tr>
<tr>
<td><strong>Constraints:</strong> a Monet database must be running</td>
</tr>
</tbody>
</table>

**Figure 7.2.** The SNA Filter Manager—Proximity Client.
CHAPTER 7. TESTING WITH DIFFERENT DATA SETS

7.2.2 O/R mapping and creating a Proximity database

Table 7.3. Use-case 3: O/R mapping.

<table>
<thead>
<tr>
<th>Use-case 3: O/R mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: the user defines nodes and edges using the “Proximity Manager”, these are summarized in an object model along with selected attributes, see figure 7.3</td>
</tr>
<tr>
<td>Input: the user selects object table(s), link table(s) and corresponding identifiers and attributes</td>
</tr>
<tr>
<td>Expected output: the object model</td>
</tr>
<tr>
<td>Constraints: tables must have identifiers</td>
</tr>
</tbody>
</table>

![Figure 7.3. The Proximity Manager.](image-url)
7.2. USE-CASES

Table 7.4. Use-case 4: Generate a Proximity DB.

<table>
<thead>
<tr>
<th>Use-case 4: Generate a Proximity DB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> the user generates a Proximity DB, see figure 7.3.</td>
</tr>
<tr>
<td><strong>Input:</strong> the object model</td>
</tr>
<tr>
<td><strong>Expected output:</strong> XML database containing the Proximity DB</td>
</tr>
<tr>
<td><strong>Constraints:</strong> none</td>
</tr>
</tbody>
</table>

7.2.3 Loading data into a Monet DB

Table 7.5. Use-case 5: Load a Proximity DB into Proximity.

<table>
<thead>
<tr>
<th>Use-case 5: Load a Proximity DB into Proximity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> the user chooses a Proximity DB and loads it into Proximity, see figure 7.2 (“Load new Monet data”).</td>
</tr>
<tr>
<td><strong>Input:</strong> a Proximity DB</td>
</tr>
<tr>
<td><strong>Expected output:</strong> a root container</td>
</tr>
<tr>
<td><strong>Constraints:</strong> none</td>
</tr>
</tbody>
</table>

7.2.4 Generating a subgraph based on filtered data with predicates

Table 7.6. Use-case 6: Generating a subgraph with QGraph.

<table>
<thead>
<tr>
<th>Use-case 6: Generating a subgraph with QGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> the user starts QGraph and creates a query, see figure 7.4.</td>
</tr>
<tr>
<td><strong>Input:</strong> a query</td>
</tr>
<tr>
<td><strong>Expected output:</strong> a container with sets of the matched query</td>
</tr>
<tr>
<td><strong>Constraints:</strong> a root container must exist</td>
</tr>
</tbody>
</table>
CHAPTER 7. TESTING WITH DIFFERENT DATA SETS

Figure 7.4. A query in Proximity QGraph with condition “id=3” on Vertex1.

Table 7.7. Use-case 7: Generate a filtered subgraph.

<table>
<thead>
<tr>
<th>Description</th>
<th>Generate a filtered subgraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>a container, a predicate</td>
</tr>
<tr>
<td>Expected output</td>
<td>GraphML file containing the processed subgraph</td>
</tr>
<tr>
<td>Constraints</td>
<td>a container must exist (not the root container)</td>
</tr>
</tbody>
</table>

Figure 7.5. The Subgraph Manager.
7.3 Results

The tests were carried out in the following computer environment: Intel P4 3 GHz, 1 Gbyte RAM, Microsoft Windows XP SP2, Monet DB 4.6.2 [CWI], MySQL 5.0.41 [MP] and Java 1.6 [ST].

Result information
Default predicate: “subgraph generated by Proximity”
Predicate 1: “Links between the subgraph nodes”, see section 5.7
Predicate 2: “Distance from a subgraph node to a root graph node”, see section 5.7
Predicate 3: “Links from a root graph node to the subgraph”, see section 5.7

Nodes are indicated as squares and edges are indicated as circles in the visualized network graphs (not in the QGraph). The visualized nodes have unique identifiers which are integers and should not be mistaken for Proximity’s identifiers.

The following results were retrieved:

Enron
Use case 1: success
Use case 2: success
Use case 3: success

Object(s): “people” (id = “personid”), “recipients” (id = “recipientid”)
Link(s): “mailgraph” (from-object = “people”, from-id = “senderid”, to-object = “recipients”, to-id = “recipientid”)

Use case 4: success
Use case 5: success
Use case 6: success, see figure 7.6
Use case 7: success
- Default predicate: success, see figure 7.7
- Predicate 1: success, see figure 7.8
- Predicate 2 with restriction input x = 1: success, see figure 7.9
- Predicate 2 with restriction input x = 3: success, see figure 7.10
- Predicate 3 with restriction input x = 2: success, see figure 7.11
- Predicate 3 with restriction input x = 7: success, see figure 7.12
Figure 7.6. Enron: QGraph query.

Figure 7.6 shows how we defined a query which only had one condition on one node, “email = craig.dean@enron.com”. This gave us all e-mails sent from that node (a person) to other nodes (recipients with no conditions).

Figure 7.7. Enron: Default predicate.

Figure 7.7 shows the default subgraph based on the query (see figure 7.6). Nodes: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (center node).

Figure 7.8. Enron: Predicate 1.

Figure 7.8 shows the default subgraph after predicate 1 has been run. Predicate 1 discovers links between subgraph nodes. But the result clearly shows that no such links were found, hence the subgraph did not change.
Figure 7.9 shows the default subgraph after predicate 2 has been run. Predicate 2 included all undiscovered nodes that were on a distance $\leq x$. In this case $x$ was 1. We can see that new nodes were discovered and their corresponding links to the default subgraph. For example, nodes 21, 19 and 20 were not in the default subgraph, but node 4 was and they were on a distance equal to 1 from 4, hence they were included.
Figure 7.10 shows the default subgraph after predicate 2 has been run. In this case we put $x = 3$ which clearly expanded our subgraph even further. This result could perhaps be of interest to a user when comparing graph 7.10 with 7.9 and 7.7.
7.3. RESULTS

Figure 7.11 shows the default subgraph after predicate 3 has been run. Predicate 3 included all undiscovered nodes which had $x$ links or more to the default subgraph. In this case we tested with $x = 2$ and we can see that the new nodes that were discovered had at least 2 edges to the default subgraph. We can see that, e.g., nodes 13 and 14 (to the right) were not in the default subgraph but they had $\geq 2$ links to the default subgraph, hence they were included.
Figure 7.12. Enron: Predicate 3, input 7.

Figure 7.12 shows the default subgraph after predicate 3 has been run. In this case we tested with $x = 7$ which clearly reduced the number of nodes if we compare 7.12 with 7.11. For example, we see that node 12 was not in the default subgraph but it had 7 links to the default subgraph, hence it was included.
7.3. RESULTS

**Books about US politics (BAUP)**

Use case 1: success  
Use case 2: success  
Use case 3: success  

Object(s): “polbooks_node” (id = “id”)  
Link(s): “polbooks_edge” (from-object = “polbooks_node”, from-id = “source”, to-object = “polbooks_node”, to-id = “target”)  

Use case 4: success  
Use case 5: success  
Use case 6: success, see figure 7.13  
Use case 7: success  
- Default predicate: success, see figure 7.14  
- Predicate 1: success, see figure 7.15  
- Predicate 2 with restriction input x = 1: success, see figure 7.16  
- Predicate 2 with restriction input x = 15: success, see figure 7.17  
- Predicate 3 with restriction input x = 3: success, see figure 7.18  
- Predicate 3 with restriction input x = 5: success, see figure 7.19

![Figure 7.13. BAUP: QGraph query.](image)

Figure 7.13 shows how we defined an advanced query with condition “id = 14” set on one node and using combinations of directed and undirected edges.
Figure 7.14. BAUP: Default predicate.

Figure 7.14 shows the default subgraph produced. Nodes: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

Figure 7.15. BAUP: Predicate 1.

Figure 7.15 shows the subgraph after Predicate 1 has been run. We see that links have been discovered between the existing subgraph nodes, these links were not included in the default subgraph if we compare 7.15 with 7.14.
Figure 7.16. BAUP: Predicate 2, input 1.

Figure 7.16 shows the subgraph after Predicate 2 has been run with input 1. New nodes have been included which were on a distance \( \leq 1 \). We can, e.g., see that nodes 42 and 44 (to the left) were not in the default subgraph but were later included since they were on a distance equal to 1 from their neighbor 10 which was in the default subgraph.
Figure 7.17. BAUP: Predicate 2, input 15.

Figure 7.17 shows the subgraph after Predicate 2 has been run with input 15. The difference between different distances can clearly be observed.
Figure 7.18 shows the subgraph after Predicate 3 has been run with input 3. We can observe that the number of links are at least 3 or more for nodes not in the default subgraph. We can see that, e.g., node 24 (to the left) was not in the default subgraph but was included since it had 3 links to the default subgraph.
Figure 7.19. BAUP: Predicate 3, input 5.

Figure 7.19 shows the subgraph after Predicate 3 has been run with input 5. We can observe that fewer nodes matched the predicate criteria, hence a smaller subgraph expansion. We can, e.g., observe that node 17 was not included in the default subgraph, but after the predicate was run it was added since it had 5 links.
7.4 Comments

The tests were completed successfully and the retrieved results were promising, network data could be created. We could filter data sets in various ways and retrieve interesting results by experimenting with different queries and predicates. From an analyst perspective the variation of different filtering techniques could lead to knowledge discovery in network graphs generated by the middleware.

We could also observe different limitations in the computer memory which are discussed in appendix A followed by a test in appendix B.
Chapter 8

Conclusions

In this chapter we present different conclusions about our middleware based on the tests. We also make conclusions about the whole system environment involving our middleware.

8.1 Object mapping and data transformation

Object mapping could be done in its simplest form described in section 4.2.2. We propose this model because we think it is the most intuitive way of mapping relational data to the concept involving nodes and edges. We do not conclude that it is always the best way, other mapping techniques could perhaps be of better use in some cases.

The data transformation process mainly involved transformation of relational data to Proximity’s import format. This was done as described in section 4.2.2 in two steps—transforming the relational data to Proximity formatted text files and then transforming the text files to a Proximity formatted XML version. The latter step should be considered as the final format (XML based format). In retrospect we must conclude that the transformation process could be reduced if we would transform relational data directly to the XML format used by Proximity instead of first transforming the data to text files and then transforming the text files to the XML format, however this was not applied in practice.

8.2 Node discovery features

The node discovery algorithm—the modified BFS algorithm—proved to work. We conclude that the BFS algorithm is an intuitive algorithm for discovering close-by nodes, which in many cases are of interest when analyzing social networks. The predicate support improved the search variety performed, hence different interesting results could be retrieved (see chapter 7). We can also conclude that the output formats—GraphML and JUNG—for the network graphs proved to be efficient when we tried to visualize the graph.
8.3 Proximity & Monet DB

In this Master’s project we used Proximity 4.2 as a filtering tool along with Monet DB 4.6.2. We can conclude that Proximity was a good choice for filtering data sets. The main reason was because of performance issues and the strong query support feature, but we could also modify Proximity to generate interesting graphs for external usage. We did not find other open source tools that provided the same functionalities as Proximity, although we found Graph-Tool which was similar but lacked the fast and strong query support which we needed. Proximity provided great knowledge discovery features but the tool itself was not flexible enough for integration with other modules. We had to make modifications for exporting data to other formats. Proximity also lacked an intuitive way to store subgraphs, it stored each matched data as an independent subgraph set. Therefore we needed to create whole connected graphs of the subgraph sets before we could implement any algorithm.

Monet DB was used mainly because of Proximity. The main reason was that a vertical database was more suited—in a performance sense—for the type of queries generated by Proximity. A Monet DB server instance could not dynamically access different databases, each started Monet DB server instance had to point to a specific database when started. This limited the way we would normally operate a DBMS. However, our design supported accessing different Monet DB server instances. This proved useful until we found out that Proximity lacked distributed access of Monet DB server instances. We found out that the only way we could access all functionalities of Proximity was to run a localhost instance of Monet DB. Since Proximity was heavily dependent on Monet DB we could not use other similar DBMS’s.

8.4 Denodo Virtual DataPort

At first our aim was to skip the intermediate storage of data in a MySQL database because DVDP could for most parts be treated as a normal DBMS, however this proved to be impossible since our design was heavily dependent on the SQL language. DVDP did not support many features of the SQL language that we needed in our design and was therefore excluded. We did not examine other database mediators due to project restrictions.
8.5. FUTURE IMPROVEMENTS

8.5 Future improvements

The following future improvements of the middleware are suggested:

- An automated function to detect “node- and edge-semantics” in relational data, perhaps by using primary/foreign key relations.

- Transforming relational data directly to Proximity XML without organizing the data in text files first and then transforming them.

- More predicate support.

- A logical engine to form different compositions of predicates using “AND”, “OR” and so on.

- Rewriting some parts of Proximity to support distributed access to the Monet DB server instances or check if other vertical databases could be used, either way the Proximity code needs to be rewritten.

- Memory optimizations using different storage techniques as well as reading and writing data on the fly.

- More intuitive user interfaces to access all functionalities.
Bibliography


**Web resources**

http://monetdb.cwi.nl/

http://www.cs.cmu.edu/~enron/

http://www.denodo.com/english/virtual_dataport.html

http://graphml.graphdrawing.org/

http://www.infosun.fim.uni-passau.de/Graphlet/GML/

http://kdl.cs.umass.edu/proximity/

http://jung.sourceforge.net/

http://www.mysql.com/

http://www-personal.umich.edu/~mejn/netdata/

http://projects.forked.de/graph-tool/

http://java.sun.com/

http://www.w3.org/XML/

http://www.w3.org/2007/OWL/

http://www.w3.org/RDF/

http://en.wikipedia.org/wiki/Social_network

http://en.wikipedia.org/wiki/Social_structure

http://en.wikipedia.org/wiki/Database_management_system

http://en.wikipedia.org/wiki/Column-oriented_DBMS
Appendix A

SNA Middleware Memory Issues

Problem

The heap size can result in an overflow if too much data (depending on the memory size of the computer and the allowed amount of memory that each process can use in the OS) are read from a database (relational or XML). This problem occurs when data is loaded into internal structures which are used to “simplify” the transformation process of importing and exporting data.

Import of data

Relational data need to be transformed to an XML format which is used by Proximity. The format consists of three sets (object, link and attribute) which eases the task of loading the data into the vertical database Monet DB. The Proximity guidelines for data import suggest storing the data into a textfile format and then using their Perl script (which we converted to Java) to transform the data to the XML format (Proximity XML).

Transforming relational data to textfiles

Basically the following steps are performed:

- Read each row from a relational table (node or edge table)
- Generate unique identifiers
- Map an old id to a new id in a java.util.Hashtable (old-ID → new-ID)
- Store each row in a java.util.ArrayList (list representing the row that was read from the database including the generated id) and store the list in another ArrayList (matrix representation)
- Read the matrix in a separate method and store its content as textfiles
Transforming textfiles to Proximity XML

Basically the following steps are performed:

- Three sets are constructed: object-, link- and attribute-set
- Each set is constructed by reading a line (with java.io.BufferedReader) from a textfile containing the table data

Memory overflow

The heap size overflow occurs if the matrix in the first transformation step becomes too large.

Proposed solution

Two actions are considered: keeping and not keeping the textfiles.

Keeping the textfiles

We could skip the internal storage structure (the matrix) and write directly to the secondary memory. For each row that we read we could at the same time write that line to a file, along with its generated identifier. Look-ups in the hashtables (used when generating the links) could be done simultaneously. This would reduce the heap size usage since each row that is read is stored directly in the secondary memory.

Although the heap size would be reduced we would still have to refer to enormous hashtables mapping an old object id to a new (generated) id. This could also be stored in the secondary memory but then the look-ups would normally take longer time, which would probably mean that the user would have to wait a very long time before the transformation process would be complete.

Not keeping the textfiles

We could choose to ignore transforming relational data to textfiles, but instead transforming directly to Proximity XML. Along with the outline of the Proximity XML format, this would require that for each object table we would have to read its data to store its old id’s along with the generated id’s. At the same time we would have to write that information. Then for each link table we would have to perform look-ups (which involves getting the newly generated object identifiers). We would also have to generate a link identifier and creating a key which maps two references (two columns in the relational table) to the generated link id.

This is needed because every single link table may not have a unique identifier. For example, if a row in a link table has the generated identifier 5 and the column references (foreign keys) 3 and 1 (to object tables), then the mapping key could be “31” (“31” → 5). Note that if the table references contain duplicate information,
then we would have to change our key. We could, e.g., create an auto-increment database identifier and then map like we would map object table identifiers.

Finally, processing the attributes requires that for each attribute we check which tables that have that attribute. Then for each of those tables we read the corresponding rows and we retrieve the attribute values and identifiers. When processing an attribute we perform look-ups to identify the nodes and edges that have that attribute and we write the data at the same time.

This solution requires more interaction with the database and also that we store the link identifiers in hashtables. Basically we would have two large tables to perform look-ups in. Note that this could be stored in the secondary memory, but that would result in a transformation process that would normally take longer time to execute.

Solution summary

Heap size usage could be reduced by storing most of the relational data in the secondary memory, “except” for the look-up tables used to map old identifiers to new ones.

Export of data

The data stored in Monet DB need to be transformed to GraphML (an XML based format) which is, e.g., used for visualization tasks. The network data in Proximity (although physically stored in Monet DB) need to be exported (both the root container and the generated subgraph container).

Modified BFS algorithm

To simplify an implementation of the BFS algorithm the containers (root and subgraph) are exported using Proximity functions to Proximity XML (the XML file containing the subgraph differs a little from the root XML file) and then read into linked lists.

Memory overflow

Heap size overflow occurs if too much data is added to the linked lists containing the network graphs (root container and subgraph container).

Proposed solution

The proposed solution is that we could store a graph structure on the secondary memory in such a way that the maximum time complexity to access all neighbors of a node would be $O(n)$. For linear access we could store the neighbors of a node in a separate table. We could also store each attribute type along with its values.
in a separate file. Then to match an attribute with an element (node or edge) we could store the corresponding identifier next to the values.

If we think more about this, we realize that this type of storage is basically how a vertical database stores its data (e.g., Monet DB), so the storage solution really doesn’t have to be recreated again. We could perform our modified breadth-first search algorithm directly on the database in Monet and store the results in the same way as above. When creating the GraphML file the results could be processed row by row in the same way as importing data when keeping the textfiles.

This solution would work but it is not that intuitive since we would normally like to have internal structures which are more easier to work with when implementing graph algorithms. Also, going through a database in the secondary memory would be more time consuming than accessing the primary memory. We must also bear in mind that the generated GraphML files can not be too large since a program (e.g., a visualization program) which uses them often has to read the structure into the primary memory, thus the problem has been dislocated.

In that situation, storing the data in the secondary memory and then accessing it would be extremely slow since many visualization layout algorithms require data access on the fly. Also note that many times all data (such as attribute information) do not need to be present when visualizing. In a visualization tool we could choose to store the attributes on the secondary memory but store the connections needed to visualize the graph on the primary memory, thus reducing the amount of data. However, the main problem about limited memory would still be present, basically we have only normalized a proportion of the data.

**Solution summary**

Heap size usage could be reduced if the process of creating GraphML files would involve accessing the Monet DB directly and storing the result in the same format (column-oriented format which is used by Monet DB). The results could then be read from the vertical database and stored in the GraphML format. These procedures would probably be very time consuming since the secondary memory is accessed frequently.

**Comments**

Depending on how much data we would like to acquire, different solutions could be selected. The export scenario, accessing Monet DB tables directly, requires massive changes in the middleware if it is to be implemented (at present). The import scenario of keeping the textfiles could be implemented (at present) since it doesn’t require a lot of changes compared to the scenario of not keeping the textfiles. The problem is that either way, the memory issue is dislocated to exporting the graphs, thus a single implementation here would not prove to be sufficient if we look at the whole data processing flow.
A total re-design of the middleware with the focus on large scale data storage solutions would have to be considered if we would like to handle extremely large graphs. That is, we would have to optimize memory usage from the moment of data import to data export. If we would like to visualize extremely large network graphs (e.g., with a visualization tool) we would have to consider two main limitations:

- Memory limitations when reading the GraphML file into the primary memory (which mostly can not be done in practice)
- Visual limitations (hard to see anything interesting on the screen) if we would manage to display extremely large network graphs (also depends on the layout complexity)

Note that the large scale graphs can be created for other purposes than just visualization, e.g., calculation of network measurements, but the limitations are considered when testing on average computers with average settings for visualization purposes.
Appendix B

SNA Middleware Memory Error
Dislocation

Test scenario

The test that was performed using the middleware (at present) shows that large graphs may in fact cause out of memory exceptions in visualization programs, even though the graphs could be generated by the SNA Middleware. These are the resources used to perform the test:

Computer:

- CPU: 3 GHz Intel Pentium 4
- RAM: 2 GByte
- OS: Microsoft Windows Server 2003 SP2

Data source:

- Enron (subset)
- 1,734,698 objects
- 362,151 links

Programs:

- SNA Middleware
- SNA Tool

Java engine:

- JVM 1.6.0_01 (build 1.6.0_01-b06)
The flow of data:

- Enron → Filter 1 → Filter 2 → Filter 3 → {GraphML file} → SNA Tool

Results

These are the results that were retrieved:

Filter 1:
- Proximity Manager
- 20,000 objects
- 7,892 links

Filter 2:
- QGraph
- 282 subgraphs
- 266 objects (unique)
- 282 links (unique)

Filter 3:
- Subgraph Manager
- Predicate: distance to outside nodes ≤ 5
- 1,805 objects
- 4,574 links

SNA Tool:
- ERROR: Stack overflow
- CAUSE: Oversized container in Prefuse (3rd party visualization library)
Stack memory increased:

- SNA Tool (visualization tool)
- See figure B.1

Figure B.1. The generated graph which caused stack overflow without increased stack memory.

Note that the graph (see figure B.1) was extremely slow to view with the visualization tool.