Vrije Universiteit Brussel – Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes – France

and

Monash University – Australia

2004

Data Mining e-Resources in the LEOPARD Platform

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Marc-Emile Vanbrabant-Cattoor

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoters: Dr. Annya Réquilé (École des Mines de Nantes)
Supervisors: Prof. Christine Mingins and Prof. Judy Sheard (Monash University)

Abstract

Repositories have a tremendous amount of data stored often making navigation or information retrieval for users a hard task to accomplish. Furthermore, repository administrators want to stay up to date with the latest usage patterns of their repositories and are looking for an easy way to extract knowledge from this data.

We propose a Business Intelligent Architecture as part of the LEOPARD project, to aid the mining of data in repositories. The architecture will allow for different algorithms to be plugged in and tasks can be performed by these algorithms upon selection by the administrator via the user interface.

The prototype of the Business Intelligent Architecture has been designed and implemented. It has a naïve algorithm implemented as a proof-of-concept. The prototype was built on the existing agent infrastructure and agent framework developed as part of the LEOPARD project.

Acknowledgements

I owe my utmost thanks to Judy Sheard who has supervised me in a very professional way and always keeps her office door open for someone to drop in.

I am also grateful to Christine Mingins who shared many ideas during the meetings and initially created this project together with Annya Réquilé.

Also, thanks to Joe Zhou for proposing his algorithm to us and sharing some meetings to explain the application of his algorithm in our platform.

And again, thanks to Jan Miller and Hugo Leroux for introducing me to the Aussie culture, supporting my work and keeping me in touch with the world.

Table of contents

Chapter 1: State-of-the-Art

1 LEOPARD

1.1 Introduction

Repositories have always been a necessity in computing, be it a floppy, a hard-disk, a database, an XML schema or a simple text log. Although a huge volume of data is stored all over the world in many different data types, users have difficulty finding or accessing this data. We do not argue that all data should be made publicly available, but for open sites such as tourism, educational, flight companies there is a need to let the user know *what* is available, *where* it is and *how* to access it. In the end the user has to be *guided* to this data in the most efficient way.

To gather all this information and aid the user in its navigation through repositories we have created the LEOPARD (Learning EnvirOnment Platform for Agent-based Repository Discovery) project. In section 1.2 we briefly discuss the layered LEOPARD architecture.

Originally, the purpose of LEOPARD was to assist students in the navigation of educational repositories and provide a feedback to administrators about the usage of the repositories. The introduction of learning objects has enabled a way to analyze resources usage and to guide users into using the right resources. Users have to be guided in a non-intrusive way, yet in the right way. For example, some approaches for tutorials in e-learning educational software might recommend the user to read the lecture resource first, before starting on the exercises. Another approach could also enable access to a resource to be prohibited if other resources need to be accessed first. Different groups of users, for example, students with no experience in e-learning might prefer a lot of assistance when navigating the resources. Other groups with previous experience in e-leaning and interaction with the system might prefer access to the resources in a minimal amount of steps. An agent system in .NET was developed to accommodate tracking of users and resources for a repository.

Recent changes to LEOPARD have made the platform more generic so it could be applied to any kind of repository. The abstraction of a resource inside the application allows it to be used for any kind of repository and thus not only for e-learning. One of these changes includes the replacement of the pre-processor by a Web service (section 1.3.1). The hype of Web services allows for easy Business to Business (B2B) communications due to the use of universal standards. If LEOPARD is to be successful for a myriad of repositories, it needs a uniform way to exchange data with them. Web services allow this type of uniform communication and can abstract the pre-processor from, for example, a log parser to a *user* and *resource* parser.

To generate knowledge from the data tracked by the agents, a Business Intelligence Architecture was proposed but still unimplemented. Section 1.3.2 gives a snapshot of the Business Intelligence Architecture which is currently being researched as part of an MSc. thesis in 2004. Its associated research field, data-mining, is discussed in section 2 of this chapter.

1.2 Architecture

The LEOPARD architecture is an agent-based platform, consisting of three layers: the agent infrastructure, the agent framework and the LEOPARD application. These layers will be described in the following sections and we refer the reader to [1] for an in-depth discussion.

1.2.1 Agent infrastructure

At the lowest level of LEOPARD, there is the agent infrastructure. Since no agent system was available in .NET, we created our own .NET agent platform. In short, the agent infrastructure consists of three main services:

(1) The Message Transporter (MT)
(2) The Name Server (NS)
(3) The Directory Facilitator (DF)

The *Message Transporter* (MT) delivers messages between agents. This is on done by utilizing the Microsoft Message Queue (MSMQ) component in Windows .NET Server. The *Name Server* (NS) takes care of resolution of an agent identifier to the host address of the agent. The host address of the agent is the machine the agent currently lives on. The MT communicates closely with the NS to find out the location of a specific agent when sending a message. The *Directory Facilitator* (DF) acts like a Yellow Pages and agents can request a particular service from it, for example, a Database Manager. The DF returns the identifier of the agent running the service. MTs have an instance on all the agents while the NS and the DF run only on one master host. This means that requests passed to a slave NS will be redirected to the master host. The master host is the first agent launched in the system. The redirection is done in .NET by a technique called proxy remoting. The ensemble of the MT, NS and DF make up the agent infrastructure.

1.2.2 Agent framework

On a higher level we had to specify the agent framework. From an outside perspective, the agent, represented by a unique identifier, is a stand-alone entity able to send and receive messages. The inner structure was inspired by the DIMA [3] design and consists of a main thread, a message queue and activities. The main thread "holds" the life of the agent; when the main thread terminates, the agent dies. However, it is desirable to execute other tasks asynchronously inside the agent; therefore, *activities* each run using their own thread and are referenced by the main thread with a weak reference. When the main thread of the agent dies, the references are used to terminate all the depending activities of the thread. The message queue is implemented using an *IList*, on which a *Message* object can be enqueued and dequeued. An *IList* is similar to a Collection in Java. Additionally, an array of *MessageCategory* objects can be passed to the message queue which will act as a filter. Messages posted to the message queue of an agent belonging to a specific category can then be used to execute a specific activity. We also introduced *Conversation* objects at this point. While agent communication is typically asynchronous and uses message passing, it could be desirable to have inter-activity

communication between different agents. Using conversations, activities can communicate synchronously with activities from other agents.

1.2.3 LEOPARD architecture

The prototype version of LEOPARD consisted of several components, all agents: a *LogMonitor Agent* (LMA), *User Agents* (UA), *Node Agents* (NA), a *Proxy* Agent for UAs and one for NAs, a UA and a NA *Maker* Agent, a *Database Manager Agent* (DBA) and a *Business Intelligent Generator* (BIG). The starting point of the data processing is the LMA, which monitors a log file. The log file is in W3C [4] format, although additional types can easily be created from the abstract Log class. The log file is scanned for new entries in a five second interval unless it is already processing. If new entries are found the log items are pre-processed and the user id, resource id and timestamp are extracted. Another task of the pre-processor is to filter out unwanted resources. This is done by matching the type of the resource and can be specified in the XML file CleaningCriteria.xml. In this implementation an IP address identifies a user and the URL identifies a resource. Although this is a naïve implementation, these fields have been typed as string and could also contain session identifiers from the web server or other (see section 1.3). The LMA sends off a message to the UA Proxy informing it of the user's navigation sequence. The UA Proxy holds a table of all UAs known by LEOPARD and forwards the message to the UA if it exists. If no UA exists in the system a "creation" request is send to the UA Maker which will notify the UA Proxy once creation is completed. Any incoming messages for the UA will be buffered on the UA Proxy while the UA is created and will be forwarded to the UA's MQ once UA creation is complete. UAs consist of user profiles (UP), user profile links (UPLink) and user profile nodes (UPNode). The UP remembers when the user was last seen on the system. The UPNode keeps track of all resources accessed by the UP. Finally the UPLink specifies the link between two UPNodes with its occurrence, last usage and read time. The read time is the total time spent on the *target* node. The NAs also use the NA Proxy and NA Maker for agent creation and contain similar profiles for storing NA information. When UAs and NAs have been idle for a period of time (30 minutes), they terminate to save resources and send the profiles to the DBA which will save them in a relational database. The agent is restored from the database when it receives a navigation activity or when the application is initially started. We can see the database as long term memory of the agents. A schema of the database and an explanation of its tables is available in Appendix A and B respectively. Finally, after each navigation activity, a request is send to the BIA which will generate recommendations to the user. Repositories administrators can also generate feedback of navigational patterns inside the system. The BIA is currently being researched and part of an MSc. thesis in 2004.

1.3 Recent work

1.3.1 Genericity

The application as proposed in [1] was designed using a web site as a repository. Indeed, the LMA can only parse log files albeit of different types. However, this defeats the purpose of analyzing other repositories, for example, accesses to a database or file system. In [2] we introduce the use of a Web service instead of the

LMA to solve the Genericity problem. The Web service is a .NET web service listening for SOAP requests and passing them accordingly to Proxy Agents as mentioned above. We argue that the knowledge of *what is a resource* and *what is a user (or session)* is not up to our application to decide. Therefore we transfer the pre-processing part to the repository administrators, outside the LEOPARD application. Repository administrators have more knowledge about the organization of their repository and their contents and know better how to define a *user* or a *resource* for their repository. Hence, writing the correct pre-processing stub to extract session information and resources is their responsibility. The advantage of this model is that the repository administrator can write his stub in any programming language and can even split up large processing in multiple heterogeneous concurrent client stubs. The only condition to be satisfied is a valid SOAP request to the LEOPARD Web service. The collection of a repository and its client stubs are called a repository domain. Now, multiple repository domains can interact in a generic way with the LEOPARD platform; however, we need a uniform way inside the application to distinguish resources from different repository domains. A solution is to use a URL [5] for resource storage. We are quite familiar with HTTP and FTP URLs, but this could be extended with a DB scheme or an XML scheme. For example, an access to a database repository located at oracle1.csse.monash.edu.au for table "students" and field "name" could result in the following resource URL:

db://oracle1.csse.monash.edu.au/students/name

Access to a file system located at cheetah.csse.monash.edu.au for file c:\documents\LEOPARD.DOC could results in following resource URL:

file://cheetah.csse.monash.edu.au/C:/Documents/LEOPARD.DOC

A scheme (the protocol identifier) for each type of repository can be defined in such a way to guarantee a unique resource representation.

Likewise, the repository administrator can choose how he wishes to define a user, be it IP address based or authentication based. User identifiers can be stored in user@repository domain [6] notation:

mvanbrab@oracle1.csse.monash.edu.au

would identify user mvanbrab accessing the oracle1 database repository in case of an authenticated user. Guest users as defined in [18] could be identified by following user identifier:

@oracle1.csse.monash.edu.au

Users defined by their IP address need to be specified in a different way. This is a client from 193.198.29.1 which is accessing the oracle1 repository:

193.198.29.1@oracle1.csse.monash.edu.au

Further work could be done on the web service to achieve this kind of functionality and some syntax checking should be build in to check for the validity of the URL and @ notation format.

Furthermore, the introduction of the web service offers a better real-time (RT) interaction than the LMA. This is especially due to the fact that no file seeking for new records needs to be done anymore in potentially huge log files.

1.3.2 Business Intelligence Architecture

The prototype application LEOPARD was able to track User Profiles (UP) and Node Profiles (NP) using agents in .NET. The advantage of using agents is clear, extended scalability, and mobile agents can allow UPs to live closer to other depending agents for processing information. However, the prototype LEOPARD did not have a business architecture, which is the focus of this literature review and MSc. thesis.

First we explain the purpose of the business intelligent component and propose an overall Business Architecture. In section 2.1 we give an overview the different types of data-mining and explain which types are relevant to LEOPARD. In section 2.2 we give some common definitions used in data-mining and compare them with the ones defined in LEOPARD. Finally section 2.3 gives an overview of popular algorithms used for mining user navigation behavior and the ones used in LEOPARD. We now proceed with the introduction of the Business Intelligent Architecture.

The idea of the Business Intelligent Architecture (BIA) is that already developed .NET (algorithm) components are encapsulated inside an agent. This creates a kind of prototype for other agents to spawn off. Multiple components could also be placed inside the agent, this allows for a close interaction inside the agent, but a transparent usage outside the agent. Communications between agents is asynchronous, thus the underlying algorithms are indirectly asynchronous. Wrapping the algorithm in an agent also gives it the opportunity to go mobile and for example migrate to a high-end server for CPU intensive computation.

The Business Intelligent Generator (BIG) is the core of analyzing the user behavior and of providing feedback to repository administrators. Inside the business domain we find the BIG itself, user algorithm agents and group algorithm agents. As explained briefly in section 1.2, the BIG receives a request from the UAs after a user has navigated. User navigation is defined as a user moving from one resource to another resource. The definition of “a user” is stated in section 2.2.1. The UA responsible for the current user will send a *BIRequest* message to the BIG, having the current UP as its content. The asynchronous nature of inter-agent communication allows the user to continue navigation around the webpage without waiting for the result of the request. When the BIG receives the request, it removes it from the queue and checks the message against its message filters. Invalid messages are discarded by the filter and depending on the type of message the corresponding agent algorithm is invoked.

There are various possible applications of the BIG. For example, we can recommend a user certain resources based on the resources he has previously accessed. Hence, after the BIG receives a *BIRequest* message, a *user algorithm agent* for recommending the user is started. This autonomous entity will then collect the necessary information

from the UP to recommend the user. A separate recommender algorithm agent is launched for each active user agent navigating in the system. More complicated recommender algorithms agents could be introduced in the application. The only steps to follow are to wrap the algorithm in an agent and add an extra message filter. We could see the BIG as a proxy server between the UA and its associated algorithms.

The recommendation might also be done by grouping the user with other users of same interest. In that case *group algorithm agents* will be spawned which will take care of the recommendation for groups of users.

When an administrator wishes to mine the environment to get feedback on, for example the most popular navigation paths in the system, we use a *group algorithm agent*. The difference between user and group algorithm agents is that group algorithm agents work on bigger sets, take longer to compute and are execute less frequently than user algorithm agents. Also, unlike user algorithm agents, group algorithms agents terminate when they finished processing the data and the results have been transmitted. Results could be visualized on a GUI, dumped to a log file for further processing or stored in a database. Typically, there will be only one instance of the group algorithm agent; in other words the algorithm is not "cloned" because more than one user exists in the system.

A figure is included below, note that the other agents (Proxy, Maker, User agents and Node agents) have been left out for simplicity.

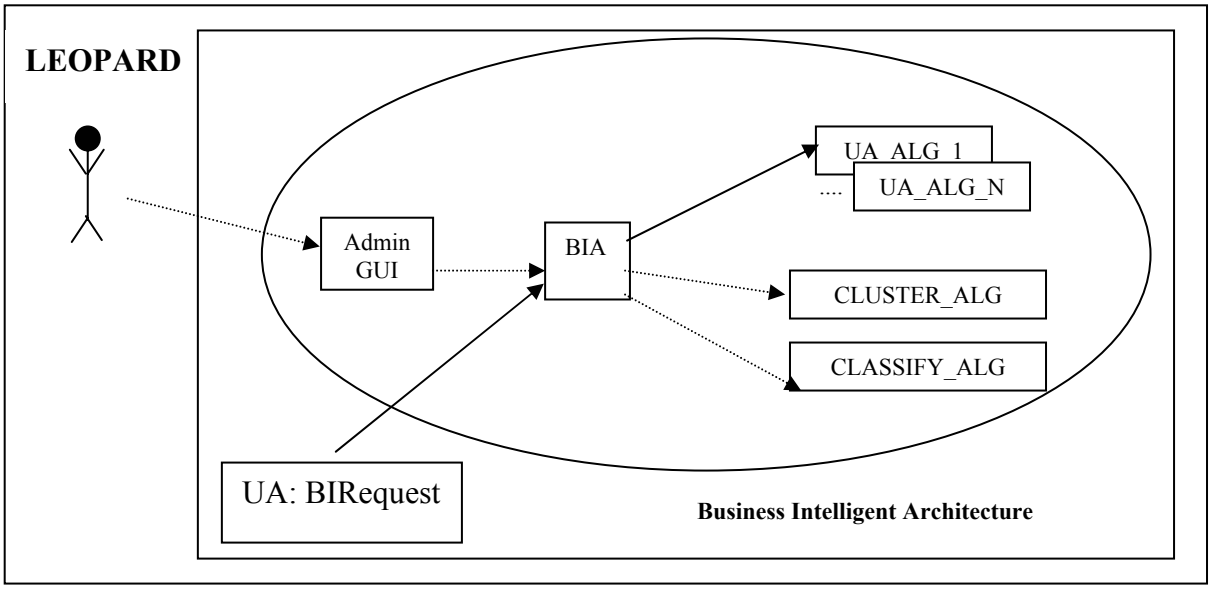


Figure 1: A close-up of the Business Intelligent Architecture

2 Data-mining

2.1 Mining areas

It is important to remember the goal when mining your resources. Are we searching our article repository for resources that belong to a certain news category, or are we more interested in how the resources relate to each other and the paths between those resources? Maybe we want to see which resources are most frequently accessed by a certain group of users to improve our marketing campaigns?

The types of data-mining mentioned above may seem all alike, however they all belong to a different area and they have been classified as: content mining, structure mining and usage mining respectively.

2.1.1 Content mining

Content mining, of which Web content mining is an instance, focuses on the content of a resource. The Web is one of the biggest resource repositories, billions of (Web) documents are spread around the world located on heterogeneous types of servers and network links. Most documents are in the form of static HTML pages, which can be considered semi-structured documents. The HTML tags provide structure in the way that they define, for example, a table for rendering purposes, but do not define the content of that table. This is exactly the problem for content mining. Instead of being able to search for the album titles, we need to mine the resource containing insignificant HTML tags to find the album titles. The rise of hypermedia (hyperlink navigation using graphical objects) gives this mining area another disadvantage. Hypermedia is often inserted in Web resources instead of text, which makes it very hard to know the content of an image by just looking at the IMG tag. However, different properties inside that tag (the ALT property for example) *might* provide us with additional information on the content of the image. Possibly, analyzing the image itself for patterns is a better solution but much harder and more time-consuming to accomplish. This could be applied to Web browsers for blind people. For example, the browser could read images off the current page, analyze them, compare them to patterns from already analyzed images and then inform the user of what the image displays.

Content mining is closely related to structure mining. Good content mining is improved by a good structure in the resource. The structure and content problem has been widely recognized in the past years and has led to the development of a new technology called XML (eXtensible Markup Language) [7]. XML defines the structure of the document rather than the layout of a document. This makes processing of XML content ***meaningful.*** It is meaningful in the way that the XML tag exactly defines what the content of the tag is. The rendering or layout is done by a separate "engine", the most popular being CSS [8], and the XSL family [9] including: XSLT [10], XPath [11] and XSL-FO [12]. After the introduction of XML, we saw a huge XML hype ranging from Web Services using SOAP [13] – an XML based B2B language – to Microsoft adopting XML as the *de facto* standard, for example, for Word formatting.

This being said, other people are working on the introduction of the semantic Web [14] which will facilitate mining Web resources, however some older resources might never be converted to this new technology leaving this research area a necessity on the Web.

2.1.2 Structure mining

Whenever we are reading a research paper or surfing on the web, we find references to other papers or pages. From the initial resource, we navigate away to other resources. However, the average number of linked resources increases exponentially with the navigation. Some resources might link back to the initial resources, some to the resources accessed in between.

This mining area is interested in studying the relation between resources. The first step in this process is to find all the references for the research paper, or the hyperlinks in case of Web pages. This is where structure mining is closely related to content mining. After all links have been found for the current resource, one of the next resources is chosen and the process is repeated recursively on all the found links.

A nice example is Googlebot [15] which uses Web content mining for extracting the hyperlinks of a page. Possibly it also deduces candidate categories to link the page in Google's Directory. It then continues crawling the rest of the Web using the links it learned from the current resource. Using this technique, Google is able to let users do advanced searches like: "search all pages that link to page XYZ".

2.1.3 Usage mining

This mining area is interested in the automatic detection of access patterns on repositories. Web usage mining studies user access patterns to Web servers. The most common data sources for web usage mining are log files, containing various data items. The most important ones are the timestamp, the current resource, the referrer and the originating IP. These items allow us to detect, for example, which users favor which resources, the average time spent on a resource, to which group of users a user belongs to according to certain access pattern characteristics (classifying).

LEOPARD concentrates on Web usage mining using the log file (or the Web service, section 1.3.1) as source of the data. Different algorithms can be plugged into the BIA to produce different types of knowledge.

2.2 Definitions

In this section we give a very brief overview of the diversity of the definitions for *user* and *session* proposed by authors in research papers.

2.2.1 Definition of a user

Due to the heterogeneous nature of the Internet, identifying a user has become troublesome. Apart from the fingerprint the user is using to type on his keyboard,

there is no unique identifier to define a user on the Web. Constant [1] splits the mining up in client-side profiling and server-side profiling. Client-side profiling has the advantage of enabling us to determine exactly who the client is, for example, by implanting a unique user identifier when installing the client side program. Client-side profiling can also track off-site activity [17]. However this technique is considered to be intrusive [18]. Hence, most other mining programs use server-side profiling. Server-profiling is also used more often because it is easier. There is no need to install specific software on the client's computer to enable tracking.

An easy way to define a unique user with server-side profiling is authentication. Zaïane [21] specifies that users in many e-learning applications are easier to identify because authentication is mandatory. Chen and Cooper [18] simply exclude guests; users that only log onto the system to look around.

Constant [1], Schechter, Krishman and Smith [16] and Kosola and Blockeel [22] propose using the IP address of the client to uniquely identify a user. While this is a reasonable attempt to identify the user, [16] argues that proxy servers mask the requests from many different users under one IP address. We think the impact of proxy server is limited because most of them specify a HTTP header with the originating IP (HTTP_X_FORWARDED_FOR). Even if this IP address is private as specified in [19], we can still deduce the "real" user for this, i.e. IP 192.168.10.1 *via* proxy server 195.10.19.8. We agree however, that users using other types of masquerading to use the Internet (such as Network Address Translation [20]) will remain affected.

The current prototype of LEOPARD uses IP addresses to differentiate between users. However, since the agents and database are both defined as a "String" type, any identifier could be used inside the application. We found out that most of the techniques mentioned above, can be used with LEOPARD. For this project, we have used the system successfully with IP addresses, usernames from authentication, and GUIDs [23] generated by Microsoft.

2.2.2 Definition of a session

Once we have successfully identified the *user* we still have another difficulty: How to establish when the user is finished with a session and when is he starting a new session.

Berendt and Spiliopoulou [24] define a session as a sequence of page accesses performed by a user to accomplish a task. The task could be defined as for example just "visiting a site" or "purchasing a product".

Schechter, Krishman and Smith [16] and Berendt and Spiliopoulou [24] adopted the heuristic that any two HTTP requests separated by more than thirty minutes are *not* part of the same user session. The time interval is based on the X value of 25.5 minutes proposed in [26]. Zaïane [21] notes that this idle time is not large enough in e-learning applications, where the session could span several hours or even days.

Similar to the definition of a user, authentication is an easy way to identify a session. When a user logs in, the previous session is terminated and a new session is started.

This can be facilitated by the use of cookies or SIDs (Server IDs). Cookies are small pieces of data stored by the browser on the client to ensure state as proposed in [27]. The state problem has existed since the introduction of the HTTP, which is *stateless*. For each page, a separate connection to the server is established, which does not really allow identification of a user. After the transfer of the page is done, the connection is lost and no notion of which user accessed the server is saved. The introduction of cookies, by putting a cookie on the client side, was a limited solution to the state problem. However, since cookies are set by the browser, state is lost when a user changes browsers, when he flushes the cookies (or disables them) or changes to another public computer. Also, the topic of cookies and privacy has been highly debated on the Web since they may be used to track user's behavior by advertising companies. The PHP [28] and ASP.NET [29] Web programming languages have built-in state management support using SIDs.

To define the beginning of a session and the end of the session, sometimes we need to introduce "start" and "stop" pages. This has also been done by [16], [25], [30] and [31]. In LEOPARD also, we have introduced a start page. For the testing of the WIER [61] database we introduced a "NULL" resource, i.e. an agent resource tracking all the "NULL" resources would define start of the sessions.

2.3 User navigation analysis

A wide variety of algorithms have been used by researches to analyze navigation of users through a system. The system's data source can range from a Web server log file [16], [25], [31], [32], a Web browser's history [17], a library system session log [18] to hypermedia navigation logs [33], [34]. In the case of LEOPARD, the generic design allows the system to be any of those.

There are various methods used to analyze the navigation data these which will be discussed in the next sections. In section 2.3.1 and 2.3.2 we discuss clustering. Clustering is a method to find groups of data based on similarity of attributes describing the data. For example, clustering could find which groups of users are closely related and might find differences in navigation behavior between novice and expert users of the system. In section 2.3.3 we discuss reduct (reduction) algorithms. These are used to determine which attributes in a data set are important and which attributes are not relevant to the definition of the data set. Section 2.3.4 discusses tree algorithms, focusing on sequence analysis. These are used to find common sequences for users accessing resources on a repository.

2.3.1 Hierarchical Clustering

One of the first papers on clustering dates back from 1963 and introduces a popular method called Ward's method [35]. Known as a hierarchical clustering technique, it creates a hierarchy of mutually exclusive subsets. The algorithm starts off with one group (cluster) of data items and then proceeds to divide them sequentially by splitting them into more clusters. However, we loose information after each ungrouping. As the number of clusters increases we get smaller groups of items, this new group yields in a less accurate group. In each iteration, the method takes the

Euclidian[1] distance between two items and groups them accordingly. If the distance between two items is small, those items will be grouped together and form a cluster. However, using this method, there is a chance that some items are erroneously grouped. These wrong groups will remain throughout the iteration process.

El-Hamdouchi and Willett [36] developed a new set of algorithms based on Ward's method for clustering documents. They also propose new strategies to increase efficiency and clustering. Other methods to identify the distance between two items include:

- The *nearest neighbor method* (or minimum method, single linkage) finds the closest neighbor with the minimum distance and merges those into a new cluster. This method tends to lead to too few large clusters and heterogeneous clusters due to chaining.

- The *furthest neighbor method* (or maximum method, complete linkage) clusters two items which are furthest from one another with maximum distance. This produces strong homogeneous clusters but results in dilatation and might produce too many clusters [37].

- The *centroid method* uses the distance to the centroid of the cluster to merge similar groups. The centroid is defined as the mean value of the objects contained in the cluster for each variable. This method is frequently used in biology.

These methods can be used in agglomerative clustering as well as divisive clustering. In agglomerative clustering we work "bottom-up", i.e. each item belongs to its own cluster and will be combined using one of the methods mentioned above into a new cluster. On the other hand, divisive clustering takes a "top-down" approach which starts with all the objects in one cluster. At each level the cluster is divided into more clusters resulting in each item belonging to its "own" cluster.

Finally, hierarchical methods are computationally expensive. Also, more accurate methods like Ward's method or the centroid method are more complex than the nearest neighbor method, which has led to the development of another clustering approach: non-hierarchical clustering.

2.3.2 Non-hierarchical Clustering

K-means algorithm

Non-hierarchical clustering or flat clustering is more efficient and faster than the popular hierarchical methods when *K*, the number of clusters, is small. However, while hierarchical clustering is more appropriate for data-analysis, flat clustering is preferable when efficiency is mandatory or the data sets are very large. The most common used algorithm for non-hierarchical clustering is *K-means* [38] and thus comes in many flavors and optimizations.

[1] The Euclidian distance is the shortest distance between two points in Euclidian space. In other words this is a straight line from point A to point B.

The steps of the algorithm are as follows:

(1) partition the dataset into *K* clusters by randomly assigning the data points to a cluster
(2) for each data point:
    - calculate the distance from the data point to each cluster
    - if the distance is closer to another cluster, move the data point to that cluster, otherwise leave it in the current cluster
(3) repeat the above step until the cluster distribution is stable, i.e. no more data points are moving between clusters

The reason this algorithm is popular is because it is easy to implement and quick to run. For the algorithm to execute, it needs two inputs: a set of vectors and *K*. One of the disadvantages of this algorithm is that it is difficult to guess *K* for an "unknown" data set. This is why the K-means algorithm is sometimes preceded by a hierarchical clustering technique as in [31]. For example, Giudici [31] uses hierarchical clustering with Ward's method as a preliminary step to find the optimal number of clusters as input for non-hierarchical clustering. To find the optimal number of clusters and the quality of the clustering, they measured the R^2 and the SPRSQ (semi partial R-square) after each iteration of Ward's method. While R^2 should be minimal, the SPRSQ should be maximal. On the basis of these heuristics, we can find the optimal number of clusters and the items inside the cluster.

Another drawback of this algorithm lies in the fixed size of *K* throughout the execution which means that some data items might end up in the wrong cluster due to local solutions and thus decreasing the quality of the clusters. Also, the final clusters are not consistent throughout different executions of the algorithm on the same data set. This is due to the random assignment to the *K* clusters in the initialization of the data set. Different random assignments in the initialization will lead to other distances between clusters and data points, resulting into other inter-cluster displacements of the data points.

However the advantages are easy, fast implementation and quick runtimes. Instead of preceding the non-hierarchical method with a hierarchical method, it is also possible to run the algorithm with different values of *K* to find an optimal cluster size.

EM algorithm

The EM (Expectation-Maximization) algorithm [39] follows the same idea as the *K-means* procedure in that a set of parameters are re-computed until a certain convergence is achieved. EM is a widely used statistical model using the finite Gaussian mixtures models. Data-mining tools like Weka [40] use this algorithm for cluster analysis.

The EM algorithm uses five parameters in each iteration to assign data items to a cluster. These five parameters for $N=2$, N being the number probability distributions

each representing a cluster, are: the mean (μ) and standard deviation (σ) for cluster 1, the mean and standard deviation for cluster 2 and the sampling probability *P* for cluster 1.

The process of the EM algorithm is as follows:

(1) assign initial values to the five parameters
(2) while cluster quality increases:

- calculate the cluster probability for each instance using the probability density function for a normal distribution. For a single independent variable with mean and standard deviation the function is:

$$f(x) = \frac{1}{(\sqrt{2\pi}\sigma)e^{\frac{-(x-\mu)^2}{2\sigma^2}}} \quad [44]$$

- re-estimate the five parameters using the probability score

A way to measure the quality of the cluster is to look at the probability that the data comes from the dataset determined by the clustering. If the probability does not increase anymore, the execution of the algorithm stops.

The advantages of this simple statistical algorithm are easy implementation and small memory requirements. However, convergence of the algorithm is slow which is why more complicated but faster algorithms have been developed to increase the speed of the convergence of the algorithm: [41], [42], [43]. EM also allows omitting the numbers of clusters and will automatically produce the "optimal" number of clusters. Weka uses the default value of -1 to specify the automatic creation of clusters.

2.3.3 Attribute reduction

While the methods discussed previously focus on finding clusters of data, some other methods use reduction to see which variables are representative enough for a data set. For example, if the data set is a log file of Web server used for online teaching, of which some variables are the student's gender, student's grades and the resources accessed (Web pages), we could be able to find out that the gender attribute is of less relevance to the data set than the other two attributes, which are more relevant to define the data set. Therefore it allows us to remove the attribute without loosing too much knowledge of the data. The reduced attribute set can then be used as an input for cluster analysis.

Reducts

The approach taken in [18] led us to a method proposed by Zhou [47] which uses a relation matrix to give us the near-optimal attributes reduct. The algorithm finds its roots in rough set theory [50] and addresses several limitations of that theory. The main problems that can be approached in this theory include: data reduction (purging

of redundant data), discovery of data dependencies, estimation of data significance, generation of decision (control) algorithms from data, approximate classification of data and discovery of patterns in data.

As both papers [47], [48] contain some small mistakes, we will illustrate the algorithm with a small corrected example used in [47] and initially provided by Skowron and Stepaniuk [51].

	Height (H)	Weight (W)	Hair (R)	Eyes (E)	D
1	Short	Light	Dark	Blue	1
2	Tall	Heavy	Dark	Blue	1
3	Tall	Heavy	Dark	Brown	1
4	Tall	Heavy	Red	Blue	2
5	Short	Light	Blond	Blue	2
6	Tall	Heavy	Blond	Brown	1
7	Tall	Heavy	Blond	Blue	2
8	Short	Light	Blond	Brown	1

Table 1: Decision table of a tolerance information system

This represents an information system *S* with users U={1, 2, 3, 4, 5, 6, 7, 8} and attribute set A={H, W, R, E} and decision attribute *D*.

The first step is to order the table by the decision attribute *D*, the first column with record numbers can be left out since it is irrelevant to the end result. Indeed, we want a reduct, not which items are in it. We left the intermediate table out to gain some space.

The second step is to make a tolerance relation (TR) matrix for each attribute in *A*.

H	1	2	3	4	5	6	7	8
1	1	0	0	0	1	0	1	0
2	0	1	1	1	0	1	0	1
3	0	1	1	1	0	1	0	1
4	0	1	1	1	0	1	0	1
5	1	0	0	0	1	0	1	0
6	0	1	1	1	0	1	0	1
7	1	0	0	0	1	0	1	0
8	0	1	1	1	0	1	0	1

Table 2: TR Matrix for Height

W	1	2	3	4	5	6	7	8
1	1	0	0	0	1	0	1	0
2	0	1	1	1	0	1	0	1
3	0	1	1	1	0	1	0	1
4	0	1	1	1	0	1	0	1
5	1	0	0	0	1	0	1	0
6	0	1	1	1	0	1	0	1
7	1	0	0	0	1	0	1	0
8	0	1	1	1	0	1	0	1

Table 3: TR Matrix for Width

R	1	2	3	4	5	6	7	8
1	1	1	1	0	0	0	0	0
2	1	1	1	0	0	0	0	0
3	1	1	1	0	0	0	0	0
4	0	0	0	1	1	0	1	1
5	0	0	0	1	1	0	1	1
6	0	0	0	0	0	1	0	0
7	0	0	0	1	1	0	1	1
8	0	0	0	1	1	0	1	1

Table 4: TR Matrix for Hair

E	1	2	3	4	5	6	7	8
1	1	1	0	0	0	1	1	1
2	1	1	0	0	0	1	1	1
3	0	0	1	1	1	0	0	0
4	0	0	1	1	1	0	0	0
5	0	0	1	1	1	0	0	0
6	1	1	0	0	0	1	1	1
7	1	1	0	0	0	1	1	1
8	1	1	0	0	0	1	1	1

Table 5: TR Matrix for Eyes

We have added the relative numbers of the rows and columns to reference them more easily. To calculate a value in a cell we take the number of the column *C* and the

number of the row *R* and compare the rows *C* and *R* in the sorted decision table from Table 1 for that attribute. In other words, does item 3 and item 4 in *S* have the same value on attribute *H* (height)? If so, the attribute is relevant and we place a 1 in the TR matrix for attribute H, otherwise we place a 0. The vertical line throughout the table shows the symmetrical axis, comparing item 3 with item 3 will always deliver a relevant attribute; hence all values in the cell along the vertical line are relevant 1. Another way to view this is that all self reflecting items are always relevant. The bold lines split up the tables in positive areas: [1:1]-[5:5] and [6:6]-[8:8] and negative areas: [1:6]-[5:8] and [6:1]-[8:5] which will be explained later in this example.
After the TR matrix for all attributes is finished, we must make a TR matrix I_A of all attributes in *S* (Table 6).

I_A	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	1	0
2	0	1	1	0	0	1	0	1
3	0	1	1	1	0	0	0	0
4	0	0	1	1	0	0	0	1
5	0	0	0	0	1	0	1	0
6	0	1	0	0	0	1	0	1
7	1	0	0	0	1	0	1	0
8	0	1	0	1	0	1	0	1

Table 6: TR of tolerance relation I_A

To calculate the value in a cell of this matrix, we need to look at the relevance of all attributes in the tolerance relation. Column 1, row 2 compares all attributes from item 1 with all attributes of item 2 from the sorted table of Table 1. If two or more attributes between the items are different, then the relation is not relevant (false, thus 0) else, the relation is relevant (1). We needed to choose an upper limit for the number of attributes that are relevant; in this example, we chose 2 for the upper limit. In the sorted table of Table 1, item 1 and 2 have Height and Weight different, the comparison for these items can terminate prematurely. There is no need to check for Hair and Eyes since 2 attributes were already non-relevant. Item 1 and item 7 have only 1 attribute different (Hair), thus they have strong relevance over most of the attributes (Height, Weight and Eyes), hence the relation is true.

After this, we continue with the calculation of β, φ and α for all attributes. β counts all the relevant relations (all ones) in the positive area. φ counts all the false relations, the zeros in the negative areas. The importance of the decision attribute α is defined by:

$$\alpha(a) = \beta(a)/\varphi(a)$$

	β (= 1)	φ (= 0)	α
H	18/34	16/30	0.9926
W	18/34	16/30	0.9926
R	18/34	8/30	1.9853
E	22/34	12/30	1.6176

Table 7: Calculations of β, φ and α for all attributes

Since attribute *R* has the most relevance (the highest value of α), we use this attribute to start the algorithm which will map the I_A matrix to the *R* matrix using the XOR operator. The algorithm continues adding the most relevant attributes (*E* secondly and

H or *W* thirdly) until the loop condition is satisfied. The loop condition is defined as: while the "0"s of the $M(I_A)$ and $M(I_R)$ do not correspond, add a new attribute. After the loop condition was met, the resulting reduct was *P*={R, E, H}.

In short, this algorithm will choose the most likely attribute to be a reduct member and incrementally add an attribute until a reduct is reached. Because of the nature of the matrix (binary) and the inexpensive calculations (binary calculations) this algorithm has been proven to perform extremely well. The time complexity of this algorithm in the worst case is *O(card(A) x (card(U))2)*. Moreover, this algorithm is well suited for parallel computing due to matrix division as explained in [48].

Hybrid approaches

A similar approach was taken in [18] which analyzes usage patterns in a library system. Twenty five base session variables were extracted from the Web based library catalogue and from these base variables, 47 variables were derived by combining variables using formulas. For example: average time between page requests = length of a session / number of pages in a session. This vector of 47 variables represented a user session; however this data set is too large for data analysis and it is very probable that some of the variables are correlated. The paper proceeds by lowering the dimension of the vector by calculating the *eigenvalue* of each variable. The variables with the highest *eigenvalues* are considered of great relevance. Using this method they retain the first sixteen variables (with eigenvalue >= 1 or proportion >= 1/20).

The remaining sixteen variables were representative enough, even with the reduction to one-third of the variables, 76% of the variance in the data was explained. After this reduction, traditional non-hierarchical and hierarchical clustering – using the SAS procedure FASTCLUS [45] and the SAS procedure CLUSTER with Ward's method [46] respectively – were applied. For the flat clustering, the value of *K* (number of clusters) was chosen to be 100. The FASTCLUS procedure allowed them remove sixteen clusters with too few observations, feeding the remaining 84 clusters into the hierarchical clustering. After the clustering was finished, six clusters remained, each identifying a particular group of users. The six groups were classified as: knowledge users with advanced usage, novice users, interactive users with good search results, users searching known items, help-intensive searches and fairly unsuccessful users of the system.

2.3.4 Tree based analysis

Trees and binary trees have been proven widely useful in data compression [54], expression parsers and file-systems [55] [56]. Another useful feature of trees is it allows for easy searching. Trees are most useful to store sequences and have therefore been used to store navigation patterns and usage patterns. Trees can be built to store the navigation paths of users throughout a system in which the nodes can contain several pieces of data. The tree can then easily be searched for patterns by use of an algorithm. The tree could be used, for example, for recommendation or for page prediction. In the case of recommendation, the tree will be searched for popular access patterns. These patterns could originate from a general tree generated from a set of users as a result of a clustering; or they could come from a personalized tree for one user. This allows several types of recommendations towards the user. On the other

hand, the tree could be used for adaptive Web design or page prediction. The tree is then used to predict the probability that a user will navigate from a specific resource to another resource. This allows, for example, the Web server to pre-generate a Web page for a particular user. Another application is adaptive Web sites, which will automatically change their content depending on frequent access patterns inside the tree. Example of such tree algorithms are given in the next sections.

Traditional date structures as stacks and queues are linear because each item is preceded or followed by exactly one item. Trees are non-linear and hierarchical data structures of which each node may have parent nodes and child nodes. A special node in the tree is the root node, the parent of all nodes. Nodes on the same level are called siblings. Trees can have any number of child nodes, sometimes called the width of the tree. Binary trees can only have 2 child nodes. They are sometimes specified as the left and right node. To traverse a binary tree, we can use pre-order traversal, post-order traversal or in-order traversal. The process for a pre-order traversal is to visit the root node, recursively traverse the left subtree and recursively traverse the right subtree. Pre-order traversal can be seen as a top-down traversal of the tree. Post-order traversal will visit the root node last, so it will recursively traverse the left subtree, then recursively traverse the right subtree and finally visit the root node. Post-order traversal is thus considered a bottom-up traversal of the tree [58].

WAP-Tree

Pei, Han, Mortazavi-asl and Zhu [32] use the notion of a WAP-tree (Web Access Pattern). The algorithm starts out with a pre-processing part by making a database of access sequence, by user session (User ID in their paper). The access sequences can be of different length and expresses which resources were accessed for a user in a session. By scanning the access sequence database twice, the WAP-tree can be efficiently constructed. On the first pass the frequent events are found and on the second pass the WAP-tree is constructed over the set of frequent events. After this is completed, we can discard the access sequence database and all further operations act on the WAP-tree. The recursive WAP-mine algorithm – their second achievement and proposition – then applies a conditional search and looks for patterns with the same suffix[2] and frequent counts of prefixes with respect to the suffix. So, only patterns with enough *support* will be considered. Also, there is no need to generate large candidate sets, which solves the problem of candidate set explosion.

Finally the algorithm returns a complete set of access patterns without redundancy and was proven to perform better than the GSP (generalized sequential pattern) algorithm [52]

PathTree algorithm

A slightly simpler algorithm was proposed by Schechter, Krishman and Smith [16], which discovers frequent paths by using a path tree. They argued that one tree could hold together all path trees instead of using multiple trees as in [53]. The *PathTree* algorithm iterates until a stable tree is achieved. This is typically after no more than

[2] e.g. if *b* is a frequent event in the set of prefixes w.r.t. *ac*, then *bac* is considered an access pattern (with suffix *ac*)

fifteen iterations. Each node stores a node occurrence count and a label (the identifier of the resource). They use a * to label the root node and they place the threshold *T* in the occurrence field of the root node. While more nodes are added to the tree, the number of occurrences for the nodes increases. If the occurrence count of a node being added is greater than *T*, another iteration inside the first iteration will add all paths to that node from which this node is a predecessor. For path [a,b,a,c,d] in a user session, when parsing the third node |a|, this means we have one root node * with occurrence 2 (*T)* and 2 leaf nodes: |a| with occurrence 2 and |b| with occurrence 1. Since |a| >= *T*, all paths where |a| is a predecessor are now relevant. The algorithm as proposed will only detect the paths where |a| is the predecessor from the current point to the end. In other words only [a,c] will be found. This is why the algorithm needs to be executed an extra time (with the occurrence counts set to zero) to detect the missing paths, in this case [a,b]. At the end of the algorithm the tree is stable and the example of path [a,b,a,c,d] is shown in Figure 2.

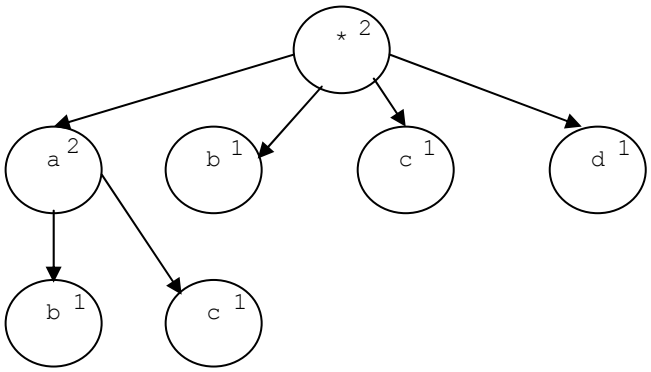


Figure 2: The stable tree for path [1,2,1,3,4] with *T*=2

The application of the algorithm was to do page prediction; however, it could easily be used for resource analysis and user analysis.

Relational Markov Models

While trees are fast, their structure is quite rigid and sometimes a burden. Moving away from trees we find structures as networks, sequences and probabilistic Markov models (PMMs).

A most interesting approach was taken by [30] using RMMs (Relational Markov Models) to allow the personalization of web pages. While a PMM will have a node for each state (page) on the site and arrows will specify the relations between the states (the hyperlinks), RMMs will generalize a set of states in a RMM groups which groups pages of the same type into relations. For example the:

```
ProductPage(Product, StockLevel)
```

relation, can group the following pages:

```
iMac_instock.html, dimension4100_instock.html, sunfirev210_
instock.html, poweredge2600_order.html
```

Other relations are:

```
MainEntryPage() and CheckoutPage()
```

Holding following pages respectively:

```
main.html and checkout.html
```

This means that all pages belong to a generalized relation *ProductPage*, i.e. they are all of the type *ProductPage*, and the type is a product combined with a stock level. A *product* in this case is a tree of products with an *AllProducts* root node with leafs as *AllComputers, AllDesktops, AppleDesktops* … The same tree structure can be achieved for the *StockLevel* relation.

```
Products hierarchy:        AllProducts(…,
                                  AllComputers(…,
                                        AllServers(…,
                                              DellServers(…,
                                                    poweredge2600,
                                              …),
                                              SunServers(…,
                                                    sunfirev210,
                                              …),
                                        …),
                                        AllDesktops(…,
                                              DellDesktops(…,
                                                    dimension4100,
                                              …),
                                              AppleDesktops(…,
                                                    iMac,
                                              …),
                                        …),
                                  …),
                           …)
```

```
StockLevel hierarchy:      AllStockLevels(
                                  instock,
                                  order
                           )
```

PMMs only work reasonably well with training data, RMMs will work more efficiently thanks to the generalization into relations thus solving one of the problems of PMM. Using a probability estimation tree (PET) on the hierarchy mentioned above, one can calculate the probability for all relations in each node of the hierarchy. In our example, the probability for accessing a page in the *MainEntryPage, ProductPage* and *CheckoutPage* relations are calculated for each node in the hierarchy. So, these probabilities are calculated for the *AllProducts* node, the *AllComputers* node, the *AllServers* node, *AllServers_Instock* node, *DellServers* node, etc.

Finally, this scheme allows us to predict the probability from any page by generalizing it to the relation it belongs to and looking at the most probable relation the user will navigate to.

g-sequences

In [24] the authors point out that frequent paths are not the best way for recommendations. Frequent paths recommend paths that have been accessed

frequently, to new visitors in the system. However, if a large number of visitors follow the same path, the system will keep suggesting this path to new users, so this path will always stay the frequent path in the system. They propose g-sequences (generalized sequences) for their WUM (Web Utilization Miner). A g-sequence is nothing more than sequences with added wildcard operators. The wildcard operators introduced are the Kleene star * and the [min;max] operator. A special value for "max" is allowed, being +∞ For example when having the following sequences:

```
(a,1)(b,1)
(a,1)(b,1)(c,1)(d,1)(e,1)
(a,1)(b,1)(d,1)(f,1)(h,1)(e,1)
```

then:

```
(a,1)*(b,1)[2;4](e,1)
```

is a g-sequence for the second and third sequence. The * denotes zero or more occurrences of any sequence (in this case both zero) and the [2;4] allows two up to four occurrences of any sequence. The sequences are then put in a tree, with the sequence identifier (a,1) and the number of occurrences of this sequence. MINT is an SQL-like querying language which works on this tree to find all sequences for a specific g-sequence. MINT can also handle *support* modifiers and *confidence* modifiers. A sample query could be:

```
SELECT t
FROM NODE AS x y z
       TEMPLATE x * y [2;4] z AS t
WHERE x.support >= 85
AND (y.support / x.support ) >= 0.8
AND (z.support / x.support ) >= 0.4
```

which will find all g-sequences in the tree where the support levels for the nodes are satisfied. The authors tested their WUM on the SchulWeb site to improve the site's conformance to its users' intentions. Their first results have been rewarding and they claim to have gained more insights in how the site is used by its visitors.

Chapter 2: Contribution

The team working on the Business Intelligent Architecture (BIA) included Prof. Christine Mingins (first supervisor), Judy Sheard (second supervisor), PhD candidate Joe Zhou and myself. Several weekly meetings where held with Prof. C. Minings, J. Sheard and myself to discuss the progress of the BIA. Sometimes J. Zhou was part of the meeting when discussing algorithm specific functions of the BIA.

1 Approach

Most of the recommender systems around allow powerful querying of a repository, however most of them restrict to one algorithm, or even one purpose. It has always been a requirement of the LEOPARD platform to be generic and that no assumptions are made of the nature of the repository. The BIA follows the same pattern and tries to achieve a generic implementation that supports:

(1) different algorithms
(2) different algorithm tasks (for example administrator feedback)
(3) independence from repository

Different algorithms allow a heterogeneous environment in which each algorithm has a specific objective. This is because an algorithm is developed to, for example, do a clustering. However, different clustering algorithms can be added to the system; so several algorithms may be used for a particular task. It is up to the administrator to choose the most suitable clustering algorithm to handle the clustering task.

Several algorithms could be deployed for example for the reduction of attribute sets. (1) does not rule out (2), which allows an algorithm to be used for other tasks (if it was designed generic enough).

The design and implementation should be repository independent. However, this does not impose a big problem because the majority of the querying of the data will be done by inter-agent communication and not by querying the existing repository. Once the data has entered the LEOPARD application, the original data in the repository is redundant.

2 Expected outcomes

The BIA needs to be able to make recommendations to users based on resources they are accessing during a session. Sometimes we will want more complex recommendations – maybe based on the type of user. This should also be made possible in LEOPARD.

The most important goal at this stage is the design of the Business Intelligence Architecture for the recommendation of users based on previously accessed resources. Also needed is an administrator module to query repository usage and to optimize it when possible. The module should be designed as an agent.

3 Proposed solution

The BIA should allow an existing .NET algorithm component to be plugged in and used. This has led to the idea of encapsulating the algorithm inside an agent. This algorithm agent (AA), having its own message queue can then receive messages from other components. The AA can of course discard unknown messages thanks to the message filters in the message queue.

Valid requests trigger a new activity inside the AA. Each activity is implemented in a new thread, which gets handled by the algorithm. While agent communication is typically asynchronous, communication between different components inside the agent could be synchronous. Of course, Conversations in LEOPARD remain synchronous. So, it is possible to have two .NET algorithm components encapsulated inside the agent, which can communicate synchronously – using normal object oriented messaging – while the agent itself talks asynchronously to the other agents using messages.

Also, we distinguish between user algorithm agents and group algorithms agents. While user algorithm agents have an instance for each user and are, for example, triggered by a navigation activity, group algorithms are typically executed by the administrator to be applied to group sets of data found in the repository. Group algorithms mostly have one instance of them inside the system (unless the administrator launches more then one).
The common applications of UserAgent algorithms are user path analysis and recommendation. Typical applications of group algorithms include classifying, clustering of users and reduction of attributes in data sets.

4 Conclusion

The LEOPARD project had a great need for a BIA as this was currently not implemented in the system. The main focus of this work was to devise a generic design for the BIA and a proof-of-concept by either using an existing algorithm or writing a simple algorithm to put into the BIA.

Chapter 3: Business Intelligence Design

Since one of the purposes of LEOPARD was to allow the mining of data, with several algorithms plugged into the application, I decided that a decent Business Architecture needed to be designed to accommodate the algorithms.

1 Architecture overview

The original design consisted of a BIGenerator Agent which was not implemented. The initial architecture can be seen in Figure 3.

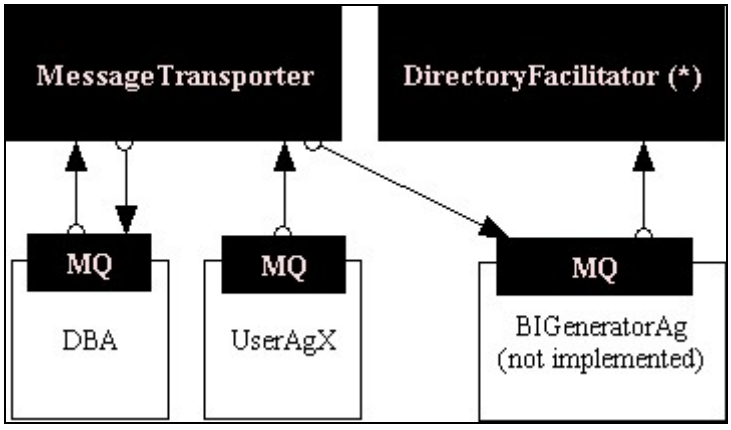


Figure 3: The original BI Generator

This figure shows the Database Agent (DBA), the Message Transporter (MT), the Directory Facilitator (DF), a User Agent (UA) and the BIGenerator Agent (BIG).

Each white box, that represents an agent, has its own MQ to receive messages from other agents. The arrows represent messages being sent between agents. In some figures, messages have been omitted to keep the figure clean. Finally, we note an arrow from the BIG to the DF. This is the registration of the agent with the DF which is done for every agent but not shown in the picture (*).

In the LEOPARD prototype, the UA would generate a message containing the User Profile (UP) of a user and send it to the BIG. All messages pass over the MT and are dispatched to the corresponding recipient. The recipient in this case is the BIG, san empty agent and thus not implemented. Also, the BIG is part of the LEOPARD architecture. Actually, all agents shown in Figure 3 are part of the same layer. In the new LEOPARD prototype, there is an added Business Intelligence Architecture (BIA). The new design of the Business Intelligence Architecture is shown in Figure 4. The BIA is shown in gray box with a dashed border.

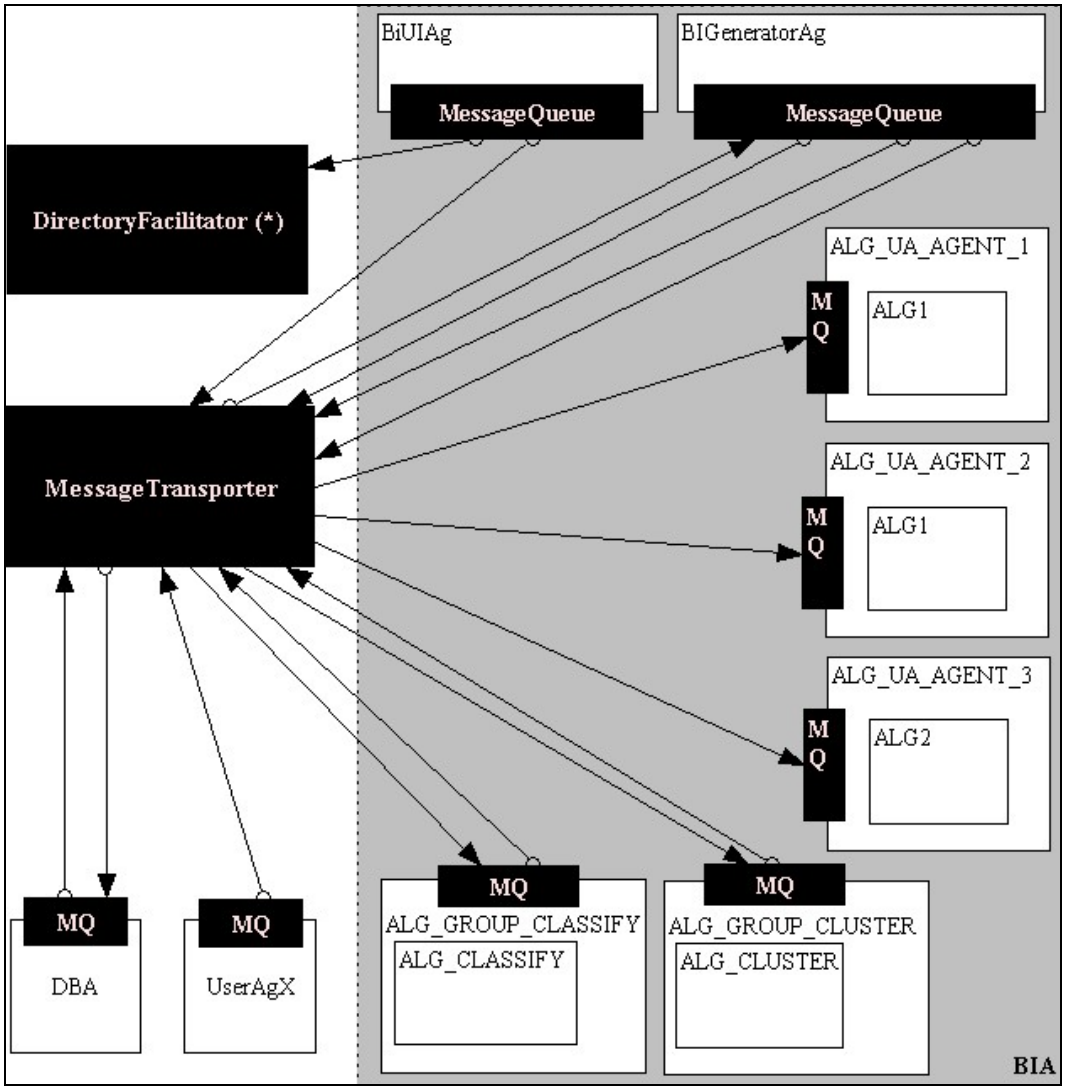


Figure 4: Overview of the Business Intelligence Architecture

This figure still shows several components such as the DF and the MT from underlying layers. Also the DBA is not really part of the Business Architecture, but has been included for better understanding. Some messages have been omitted to keep the figure clean. Similarly to the previous figure, we note a registration from the BiUI (Business Intelligence User Interface) to the DF. This is not shown for all agents in the system (*). The figure also shows the two types of algorithm agents: user algorithm agents and group algorithm agents. We will now break down the diagram to further explain their purpose.

2 User algorithm agents

As explained briefly in Chapter 1, section 1.3.2, user algorithm agents serve the sole purpose of generating knowledge for a particular UA. The request for data generation was already coded in the LEOPARD application and was the starting point for designing the architecture. To illustrate this, in Figure 5 a UA (UserAgent2) creates a new message with a BIGeneration subject. This is a request for Business Intelligence Generation. The creation of this message is done after a user has navigated to another resource. In the old LEOPARD application this means: when the Log Monitor Agent

(LMA) parsed a pair of resource records for the same user. This might be real-time (RT), but does not necessarily have to be. For example, when the LMA is parsing a huge set of new log entries, requests might be delayed because the LMA will not rescan the log file for new entries when it is already processing. With the LEOPARD Web service, chances that the user receives RT recommendation are much higher because we get a near-RT request to the Web service. The request is considered near-RT because of small networking delays and processing delays to generate the recommendation. All messages pass over the MT, which has been shown at this level for clarity. Other services like the DF and the Name Server (NS) which are also used at this level are left out. The BIGeneration arrives at the BIG which works more or less as a proxy server for user algorithm agents. In short, the BIG holds a hashtable of user algorithm agents that are currently executing in the BIA.

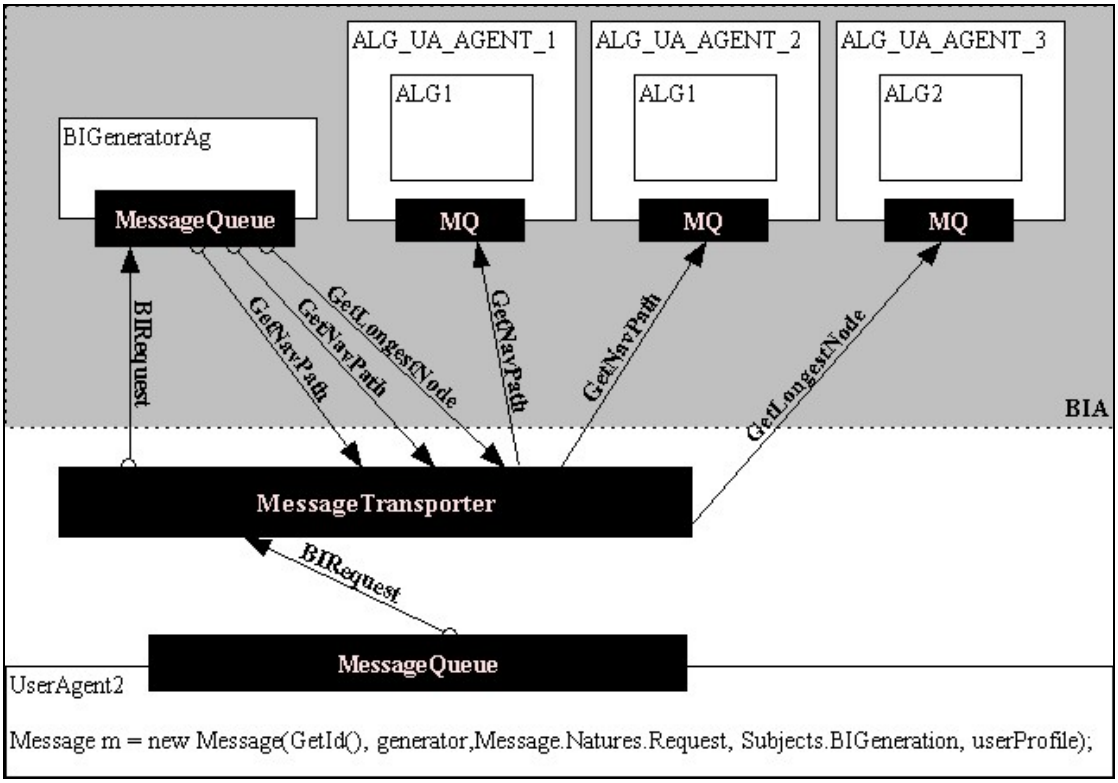


Figure 5: User algorithm agents architecture

The BIG validates the message category against its message queue filters and creates a new activity. A new activity means a new thread inside the BIG, which unblocks the main thread. Thanks to the unblocking of the main thread, the BIG can now go back to parsing messages arriving from other UAs while the activity running in a separate thread takes care of the original UA request.

At this point the activity checks the hashtable to see if a recommender algorithm for this UA is already running and creates one if none has been found. When the recommender algorithm finishes, it terminates and the entry in hashtable of the BIG is removed.

The T3RA (Top 3 Recommender Algorithm Agent) is a simple and naïve recommender algorithm developed by me to demonstrate the workings of a user

algorithm agent. After this algorithm agent for a UA is created, it will receive a message from the BIG. This message holds the same content as the message sent from the UA to the BIG. The content of the message is the UP of the UA. When the T3RA receives this message, it will extract the UP from its content and query the UP for the last node accessed. After this it will retrieve all the links from the last node and make a top three based on the number of occurrences of the links. The user is now recommended through this algorithm agent and the user algorithm agent, serving no more purpose, terminates.

Caution must be used when a new BI generation request is received for the same UA. In other words, the BIG receives a new BI generation request for the same user, while a user algorithm agent for that user is already processing. Several solutions exist:

(1) We can choose to ignore the new request and wait for the older request to finish processing and send its feedback to the user. While this is an easy solution, this technique is not desirable. This would mean that a recommendation of a newer resource would have to wait until the recommendation of a previous accessed resource is finished. The user has no interest anymore in a recommendation from a previous accessed resource; he has already made up his mind and moved on to what he choose to be the most important resource. We must try to avoid this technique and advise the user as soon as possible.

(2) Another, more feasible method is to allow the generation of the newer resource anyway, while the old resource is still processing. This will create an overhead on the system but will at least recommend the user in a smaller time frame than the technique explained above. Unfortunately, we are vulnerable to a potential DoS (Denial of Service) when a UA repeatedly starts sending generation requests in short intervals to the BIG. Recommender algorithms for the UA will keep spawning inside the system, clogging up the resources. Eventually recommendations will become slow or impossible.

(3) A further solution is to terminate the older recommendation algorithm so resources are freed for the execution of the new recommender algorithm. This will allow a RT response to the user while resources are kept minimal for a UA. However the implementation of this method is a bit more complicated as we will have to keep references to the threads that take care of the recommender algorithm. A similar solution as in the UA Proxy can be used, by keeping weak references to the involved threads.

At the moment only one user algorithm agent has been developed, the T3RA, but the architecture allows for easy deployment of new user algorithm agents. A second recommender algorithm could easily be deployed. Several suggestions for implementations of recommender algorithms:

(1) Top N resources, based on the occurrences of the resources links from a resource. We could see this as a forward recommendation.

(2) Top M resources that link to the current resource. In other words, resources from which a user frequently navigates to the current resource, based on the

number of occurrences of the resource links. This would be the opposite of (1), we could call it backwards recommendation.

(3) More advanced algorithms using statistical methods instead of the number of occurrences of the links.

3 Group algorithm agents

In contrast, group algorithms agents work on larger sets and have typically one instance running. They are not tied to UAs as user algorithm agents are, but are instead started by an administrator to get feedback on repositories. The sample architecture is show in Figure 6.

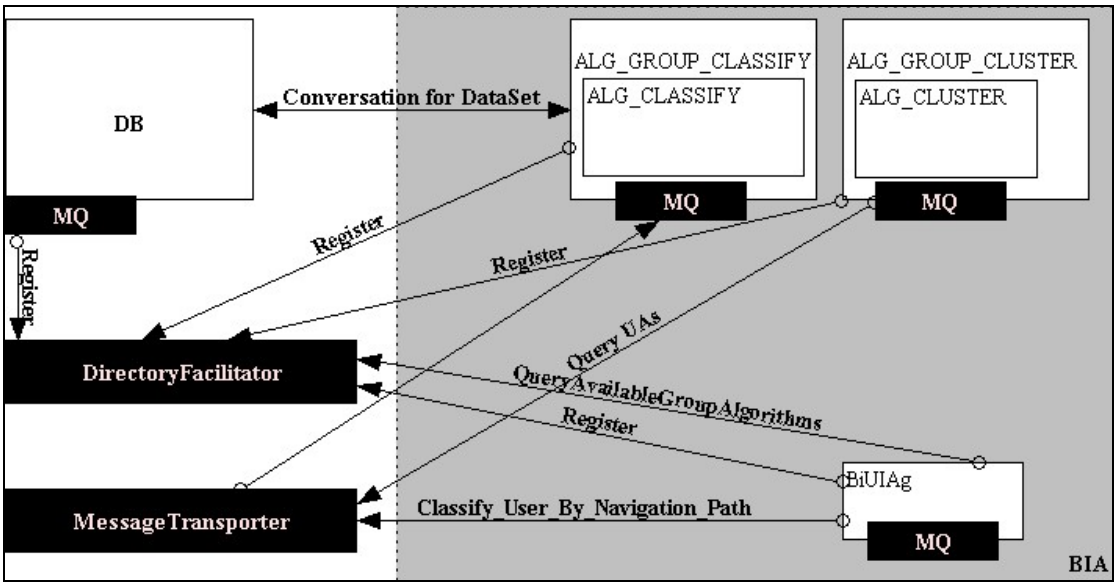


Figure 6: Group algorithm agents architecture

One of the differences between the user algorithm architecture and the group algorithm architecture is the "removal" of the BIG. It has been is replaced with a BiUI. As the name might suggest, this is a Business Intelligence User Interface Agent which the administrator uses to query the underlying agent infrastructure. The first step for adding a group algorithm agent is the same as for user algorithm agents. Encapsulate the algorithm in an agent and define its filters and activities. Then, once the algorithm is implemented, it can be plugged in. When an agent initializes, it registers itself with the DF. Using the BiUI, the administrator can request a particular service, for example, a clustering algorithm we want to use. If the agents did not register with the DF, this would not be possible. The UI could contain a list of algorithms that registered with the DF and a list of "tasks" an administrator wants to perform on the agents. The UI could also show the algorithms that are currently running in the system. This would show the group algorithms and maybe even user algorithms currently executing. Furthermore the UI could be used as an overall administration panel to, for example, stop the LEOPARD application, save the agents to database, kill algorithms that are stuck and at a further stage: code generation for dynamic agents (see Chapter 5, section 3).

When developing a group algorithm agent, which is much more complicated than a user algorithm agent, we have to keep in mind one very important factor. As the original LEOPARD specification proposes, UAs should terminate after an idle period and save their state to a database. While this has not yet been implemented in the LEOPARD prototype this poses an added complexity to the mining of the data stored in LEOPARD. Several solutions to this problem are possible:

(1) We only consider running group algorithm agents to query active user agents and node agents. This means that the generated knowledge will impact users or nodes that have accessed a repository in the last 30 minutes (the termination timeout). In this case, the knowledge we derive will always be in the form of: "Between 16:00 and 16:30 on the 7th of July 2004, 4 groups of users were found in the LEOPARD application". This is however not satisfactory as it imposes us to run the simulation every *X* minutes to achieve knowledge about bigger time spans.

(2) We use the database as the source of our mining and have all knowledge generation taken place from the database. This technique allows us to investigate bigger time spans and has the added advantage that all the data is centralized. Also, SQL queries are a fast and optimized way to query the data. However, the database is only updated when a user agent or node agent terminates or the application terminates, so on average the results will not be 100% RT and accurate.

(3) A solution I have implemented includes an auto-save execution every *X* seconds/minutes. This will trigger the update method of the DBA in an interval of 10 seconds. In other words, the DBA saves itself with all agents' state to the database. This method has the added advantage that the application is making a backup of itself. If the LEOPARD application crashes, the state will always be up-to-date until 10 seconds before the crash. Querying is done on the DBA *and* the agents. It is also possible to just query the DBA at this point. Data inside the DBA will be 10 seconds delayed.

(4) As in networking, we can implement a kind of broadcast message. A broadcast message is received by all agents, instead of just one agent and thus wakes up all agents. All agents are then restored from the database and the group algorithms can then use the agent infrastructure to query the agents. However, the reloading of all agents from the database will without doubt take a lot of resources and clog up the system. I am sure this technique will not perform well at all, but I mention it for the sake of completeness.

(5) Never have the agents timeout and only save their state to the database when the application terminates. While this will take a lot of resources, it just might be a solution for a test system. Maybe the addition of mobile agents will allow "idle" agents to migrate to a low-end server where they can live in a state of "hibernation" and can be quickly recovered. In other words, the agents do not completely terminate as in the first scheme, but they live in a state in between, limiting the amount of resources they use. For example, this could mean blocking certain threads to keep processing as low as possible and cleaning up unused threads.

My personal experience with the LEOPARD application favors the third solution. The auto-saving DBA assures a backup solution for the LEOPARD application and always keeps agent memories intact. However, it defeats the reason of having an agent infrastructure in the first place. If, for example, a clustering algorithm is used for clustering the resources accessed by users in the system, why would we have a whole agent system in the first place when we can just query the DBA? If we want to use the agent infrastructure for the grouping algorithms, *all* agents need to be active, only leaving solution (3) or solution (4) left to implement. A hybrid solution exists by querying the DBA as well as the agents. Querying can be done in parallel and will increase performance for the data exchange. However, this approach is complex and might take a long time to implement.

4 Implementation Process

The process to plug-in a user algorithm agent or a group algorithm agent is the same. The process has been described below and is recommended for successful usage of the BIA. The following instructions are applicable at design-time. Solutions for run-time loading of algorithm agents is discussed in Chapter 5, section 3.

(1) Open the LEOPARD solution in Visual.NET
(2) Load the assembly of the algorithm component in Visual.NET

This is done by adding a Reference to the component. The component is usually has a .exe, .ocx, .dll extension for COM component, .NET components mostly have a .dll extension

(3) In the Architecture.AlgArch namespace, create a new C# Class, giving it a sensible name:

I suggest a naming convention in the form of: *AlgorithmNameAlgAg.cs*. A recommender algorithm would then be called *RecommenderAlgAg.cs*, a classify algorithm *ClassifyAlgAg.cs*.

(4) Open up the class and extend the class from the Agent class. Just extending this will lead into a compilation error, because you need to add some references to the *MAP* namespace. This can be done by adding:

```
using MAP;
```

(5) Compiling this piece of code will still not work because the Agent class implements an abstract method `Execute()`. This method is implemented by adding:

```
protected override void Execute()
{

}
```

This code will compile, however when instantiating this agent, it will terminate immediately. Also, all agents need a service name and need to register themselves with the DF.

(6) We add a service name to the agent:

```
public static readonly string SERVICE =
"Business_intelligence_recommender_algorithm";
```

This is a class variable, so it needs to be inside your class and outside the constructor and class methods. You need to make sure you do not pick existing service names. Use the "business_intelligence" prefix and an "algorithm" suffix. I suggest the following naming scheme:

```
SERVICE = "Business_intelligence_AlgorithmName_algorithm";
```

(7) Make the agent register itself with the DF.

For this we need to modify the constructor a bit. First of all we need to call the `base()` method to ensure the upper constructors from the Agent class are executed. After this, we register the agent with his service name and his ID to the DF. A modified constructor looks like this:

```
public TestAlgAg():base()
{
DirectoryFacilitator.GetDF().RegisterProvider(SERVICE, GetId());
}
```

(8) The agent will now register, but still terminate directly. To blow some life into the agent we add an infinite loop. This infinite loop is known as a "busy while" and is generally not a good practice because it keeps the CPU busy, making the LEOPARD terribly slow. Therefore, I put in a `Thread.Sleep(SLEEP_TIME)` in each agent. The `SLEEP_TIME` is defined in the Agent class and is set to 10 ms. This will allow the CPU to switch to other threads in the system making the "busy while" less busy for a short period of time. To use the Thread class we need to reference it by adding the following to the class:

```
using System.Threading;
```

(9) Inside the infinite loop we typically place some code for reading the incoming messages. Since we will have multiple messages arriving in the agent, we add another iteration which will read the messages. The `Execute()` method now looks like:

```
protected override void Execute()
{
       while(true)
       {
              while (msgQueue.IsEmpty)
                     Thread.Sleep(SLEEP_TIME);
              Message m = msgQueue.Dequeue();
       }
}
```

The *m* object holds a Message object which was just popped from the MQ.

If we want to use filters on the MQ, continue with step 10, otherwise continue to step 11 to add activities. If you do not need either, this is the end of the process.

(10) To use a message filter, you need to define an object inside the agent for each message category. Then, when the message has been dequeued, you can validate the message against the message categories you defined. The following is an example from the BIG, which defines the `BIRequest` object sent after each resource navigation, from a UA to the BIG.

```
private static readonly MessageCategory BIRequest = new
MessageCategory(MAP.Message.Natures.Request, Subjects.BIGeneration);
```

After the message has been dequeued we can test the message on the category by using an if-else structure or a case statement. An example:

```
if(BIRequest.IsCategoryOf(m)) { }
else { }
```

(11) It is a good idea to use activities inside agents since they create a new thread for executing CPU intensive tasks. Meanwhile the main thread of the agent is blocked and no message dequeuing can be done. To use an activity we add a private method to the agent, using the Message object *m* as a parameter. From there we instantiate the activity. The activity itself can be coded in the same class file. The activity must be extended from the *Activity* class which takes care of the creation of a new thread for the activity. The implementation of the activity is put in an `Execute()` method, similar to the `Execute()` method for the agent. The method inside the agent:

```
private void RunAnActivity(Message m)
{
       new AnActivity(this, m);
}
```

And the activity itself:

```
class AnActivity : Activity {

private Message m;

public AnActivity(Agent _agent, Message _m) : base(_agent)
{
       m = _m;
}

public override void Execute()
{
       Console.WriteLine("The Activity");
}
```

Of course it is possible to pass more than two arguments to the activity, by using an appropriate constructor.

Chapter 4: Implementation

1 Implementation strategy

To illustrate the usage of the Business Intelligence Architecture, I needed to implement different algorithm agents. After discussing the architecture with my supervisors, I was told to implement as much of the architecture as possible and a .NET algorithm component under development would be supplied to me. Depending on its purpose, this .NET component would serve as a user algorithm agent or a group algorithm agent inside the BIA. I had planned some contingency in the project plan because I did not want to depend completely on the .NET component. I started to implement the BIG, one of the agents inside the BIA which were part of the original design but not yet completed. The implementation of the .NET component was delayed and meanwhile I started working on other aspects of the thesis. This is where a big part of the literature review was completed. After the completion of the literature review I decided the project had reached a critical point and started working on the T3RA. The T3RA is a simple recommender algorithm agent for suggesting popular navigation paths to users. The BIA now has a working BIG and the T3RA (Top 3 Recommender Algorithm Agent) as a proof-of-concept. Some of the parts that still need to be implemented are written down in Chapter 5, Future Work. Source code listings are provided for the agents that I implemented. They can be found in Appendix C. The project plan shows the activities taken throughout the project, it can be found in Appendix D.

2 BI Generator Agent (BigGeneratorAg.cs)

The BI Generator Agent (BIG) is the heart of the user algorithm agents because of its proxy behavior. Its working is closely related to the Proxy agent from UAs and NAs. However, the difference between the Proxy agent from the UAs and the BIG is that no Maker agent is currently required because the current implementation is quite simple and no buffering of messages is required when a user algorithm is created as done in the Maker agent. The message passing could be seen as a Fire-and-Forget pattern.

The BIG will need to be extended with one of the schemes mentioned in Chapter 3, section 2 to handle RT recommendation when duplicate messages from the same UA arrives. At the moment, duplicate messages are ignored and recommendation will only happen after the current recommendation agent for that UA has terminated.

The BIG class re-uses and extends from the Agent class, which implements its own thread and provides an abstract implementation of an Agent. The abstract class Agent also provides the message queue and the agent id. In the constructor of the BIG we register the agent with the DF and register the filters to the message queue. The service name of the BIG was chosen to be "Business_intelligence_generation". The abstract class also forces us to override the `Execute()` method. This is where we dequeue messages from the message queue and match them to the filters. Currently only one message category has been added to the filter, the *BIRequest* message category, which will execute a new activity inside the BIG. Finally, this activity launches a new T3RA if no algorithms for that UA are processing.

The behavior of the main thread of the BIG is described in Figure 7 in a Statechart diagram.

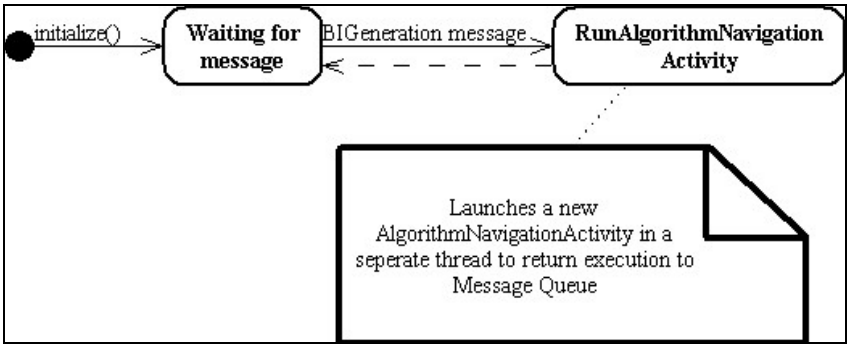


Figure 7: Statechart diagram of the BIGeneratorAg

Notice the dashed lines in the diagram that represent possible concurrent behavior due to the creation of a new thread for the activity. The execution of the activity itself is straightforward and has been shown in the sequence diagram in Figure 8.

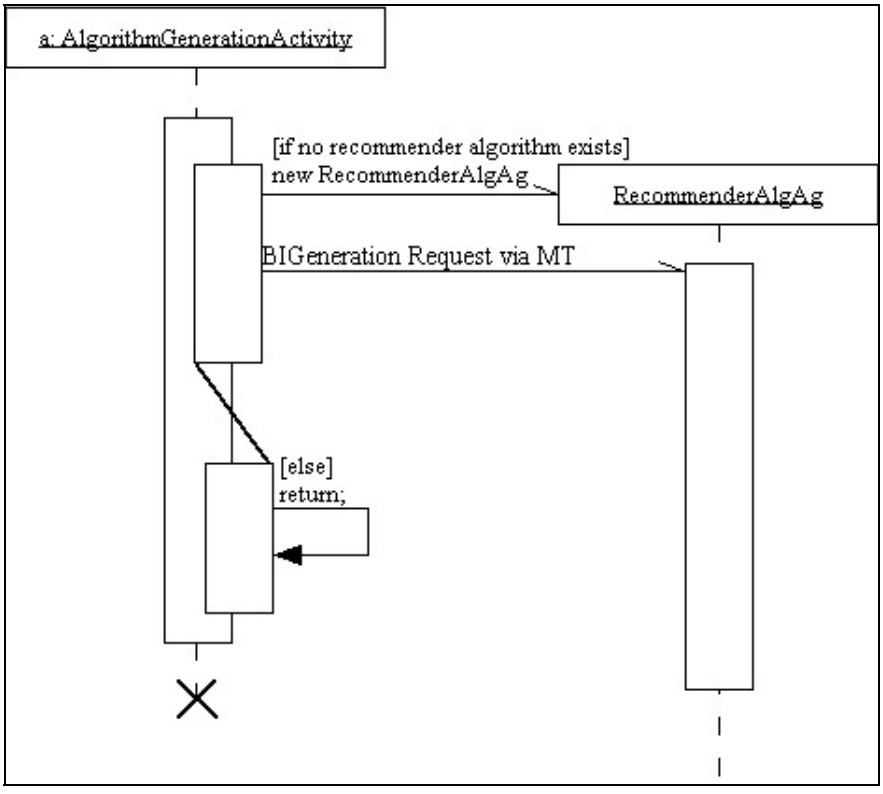


Figure 8: Sequence diagram of the Algorithm Generation Activity

The important thing to note in this diagram is that the T3RA stays alive after the activity thread terminates. Indeed, the only two reasons for having the activity are:

(1) To unblock the main thread from the BIG to continue parsing the message queue.

(2) To check the existence of a recommender algorithm for that UA and create a new one if not so.

The T3RA terminates when the recommendation to the user is completed. The BIG has been made part of a new namespace: `Architecture.AlgArch`.

3 T3RA (RecommenderAlgAg.cs)

The Top 3 Recommender Algorithm Agent (T3RA) is a simple agent executed by the RunAlgorithmGeneration activity in the BIG and recommends to a specific UA some resources. The recommendation is based on the number of occurrences the user has previously accessed the resources, so it is not affected by other users. The agent is also part of the `Architecture.AlgArch` namespace and uses `"Business_intelligence_recommender_algorithm"` as a service name. The agent holds a read-only variable called "TOP" which is currently set to "3". This is the number of resources that will be recommended to the user. In future implementations it could be possible to remove the read-only modifier and to provide the agent with a number of resources to recommend, for example specified by the administrator in the BiUI. As all other agents, the recommender agent registers itself with the DF in the constructor. The sequence diagram of the recommender algorithm agent is shown in Figure 9.

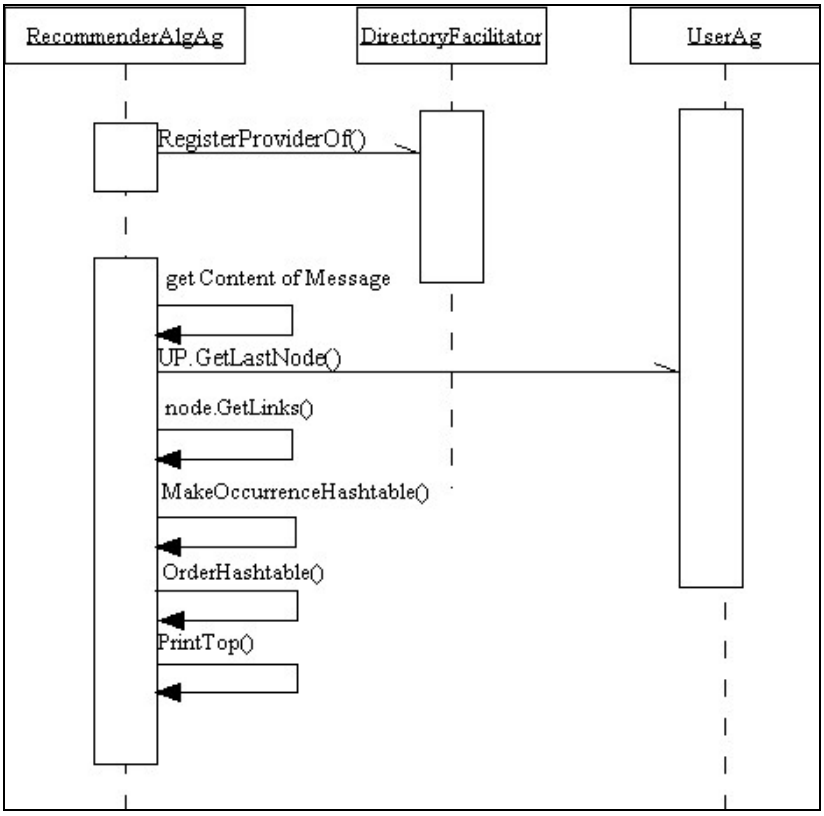


Figure 9: Sequence Diagram

First, the contents of the message is extracted, which still holds the original message sent from the UA to the BIG. This holds the UP of the UA. Using this we can get the last node accessed for that user. We then fetch all the links from that last node and count the occurrences of each link. The occurrences are the keys in a hashtable for fast access, this is done by the `MakeOccurrenceHashtable()` method. The values of the

hashtable are an ArrayList of resources. After this the hashtable is ordered and reversed (the biggest occurrences come first). This is done by the `OrderHashtable()` method. Finally, the top *X* resources are printed by the *PrintTop()* method.

4 Testing classes (Testing123.cs)

Some utility methods have been created to aid me in small tasks. They are all part of the main method in the Testing123 class.

The first big method was the one to parse the WIER [61] log file to fill the database with sample data. The log file contained approximately 235,000 records and it took about 1 hour to fill the database. The old implementation of the LEOPARD application had the database saved when the application was terminated, which would take another hour or more to save all the agents to the database. This is why I added a public method `SaveUserDB()` to the DBA, so I could save the database every 10,000 records. The implementation of the `DataSet` in .NET seems to have a buffer that remembers which records have already been saved and thus it only saves new objects to the database that have not been saved already, speeding up the saving process. The process of the method is simple. It steps through all the records in the log file sequentially and generates the UP depending on the user sessions. If a new session is started, the "NULL" resource is used to specify the start of a new session.

The second biggest method was used to test the BIG and the Recommender algorithm using the WIER data. At first, the database is initialized and the BIG initialized. At this stage there was a problem because the amount of data in the database was just too large to create all agents on the system. This is why the UP objects are only created in memory, the UAs themselves are never created at this point. This allowed me to create only three agents on which I wanted to test with. I create them myself in the test method explicitly and then use messages to communicate between those. Having to create all the agents would have been too time consuming and pointless for the test. Finally, it will show a top three recommendation of resources, generated by the recommender algorithm agent for that UA.

The code listings for these methods are available in Appendix C.

Chapter 5: Future Work

1 Web service completion

The Web service for preprocessing, introduced in section Chapter 1, Section 1.3.1 needs to be integrated and fully tested.

While the old platform which uses the LMA was started and instantiated from the command line, the new platform is started by the first HTTP request to the Web service. So, the Web service holds all the references to the Database Manager agent and Proxy agents. While this is fine for a test setup, the Web service should just be a server stub for receiving the SOAP request, then passing them on to the LEOPARD application using Proxy Remoting.

Also, the Web service will allow any input for a resource or a user ID to be forwarded to the LEOPARD application. Some test methods need to be developed on the Web service to test the validity of the input (see Chapter 1 section 1.3.1), unless we completely trust the source of the data (the repositories).

In the latter case some extra fields are required in the Web service (and the database) to authenticate the client repository. Authentication is merely used to check the validity of the source of the data, it has no further meaning to the rest of the data-mining.

2 Administration agent

There is a need for the development of an Administration agent, as proposed in Chapter 3, which is able to execute tasks with certain algorithms. This agent should also allow the loading of new algorithms. This could be done by assembly loading or by using the Reflection package in .NET. At one point the UI could also be merged with the BIG. This would leave 1 agent on which all Business Intelligence communication passes through. The communication could come from the administrator (UI) as well as the UAs (BIRequests).

3 Code generation for dynamic agents

The *MessageQueue* for the agents are quite rigid and burden the flexibility of the agents. Agents are autonomous systems and it could make sense if the agents would be able to expand themselves with new instructions. Related to the above topic, agents should be able to generate some new pieces of code inside the agent, based on a message posted to the agent. This would allow for more dynamic agents, limiting the need to decommission an "outdated" algorithm agent and merely updating an agent with new code. This process could be thought similar to updating the firmware of a router, a DVD player or a computer BIOS. Maybe, a starting point is to enhance the original Agent framework with a code generation component.

4 Test algorithms and testing

The algorithm developed as proof-of-concept in this thesis was a naïve implementation for proof-of-concept. Other algorithms should be ported and tested on the BIA.

Testing the integrity of agents should also be possible. Test methods could be developed and coded inside each agent. These methods execute when receiving a special "test" message, for example, from the BiUI. They should use test data stored in the agent to check the overall integrity of the agent. The test methods could be made mandatory by putting them in the abstract Agent class. This enforces the programmer to override the original test method.

Also, a TestingAgent should be designed and implemented. The purpose of this agent would be to supervise all the agents in the system, policing agents and maintaining a certain QoS. The Quality of Service consists of several parameters defined by the administrator of LEOPARD. For example, the administrator could tell the TestingAgent to maintain a specific number of UAs in the system and to prematurely terminate to disk when this amount is exceeded. Other applications of the TestingAgent could be to load balance agents inside the system and to request an agent to go somewhere else when the system is overloaded (mobile agents).

5 Debugging

The current debugging is mainly implemented with `Debug.Writeline()` and `Console.Writeline()` statements which is, in my opinion, not the best solution. A better solution would be to write a small Debug agent that allows any message type to be posted to it and handles the contents as debugging information. The Debug agent could be set to write to the console, or to save debugging records to the DBA, or to send them to the BiUI for presentation to the administrator. Either way, the implementation needs to allow the adding of other logging mechanisms quite easily. This is the same implementation as the LMA, in reverse workings. Instead of the reading the log file, messages need to be written to a log file. The LMA makes a suitable agent to reuse as a Debug agent because it has a solid abstraction build into its Log class.

Conclusion

Since the early 1990's, universities and companies have become connected to the Internet and have increasingly recognized e-learning applications as one of the easier, cheaper and effective ways for online and distance education. With the gargantuan set of resources the Web has to offer, there is a need for knowledge generation so that users can be guided to the right resources and administrators can analyze dependencies between resources. Neoteric repositories need a Business Intelligence Layer that generates this kind of knowledge.

LEOPARD was built with the purpose of analyzing usage of resources in, and to guide the users through, e-learning applications. Repository owners are able to extract different kinds of knowledge from the data and can use it to improve the recommendation to the users and provision of resources.

The LEOPARD application was implemented using an agent platform in .NET and tracks resources and users in repositories. A Business Intelligence Architecture needed to be designed and implemented. The design and implementation of the Business Intelligence Architecture was part of thesis. The crucial part was to design a way to plug in any .NET algorithm component into the architecture. We found out that algorithms can be split up in user algorithms which generate knowledge for a particular User Agent and group algorithms which generate knowledge for a set of data. This set of data could be a group of resources or users. This lead to the design of user algorithm agents and group algorithm agents in the BIA. At this point, these algorithms still needed a way to communicate with the other agents. We concluded that wrapping up an algorithm inside an agent gave the algorithm component "a life". Using messages, the algorithm agent can now communicate with other agents and using conversation, it can communicate with activities inside other agents. The message queue of the algorithm agent is used to push messages or data towards the algorithm.

A Business Intelligence Algorithm was designed to make full use of the user algorithm agents and the group algorithm agents. A BiUI was also added to the design. This will allow the administrator to interact with LEOPARD and to manipulate the group algorithms. In future implementations it could be used to generate code at run-time, to add new algorithms without needing to stop the application for a compilation. The implementation of a simple recommender algorithm (T3RA) shows the workings of a user algorithm agent and gives an example of how the BIG can be used to dispatch messages to other user algorithm agents.

The most significant achievement of this thesis is the design of the Business Intelligence Algorithm and the implementation of the BIG and the recommender algorithm agent. The research has given me multiple ideas to improve the project. The areas I am keen to explore are: code generation for dynamic agents, the implementation of the Web service and the administration agent to fully control LEOPARD.

Glossary

AA: Algorithm Agent, this is an algorithm encapsulated in an agent, the algorithm will have its own message queue and activities.

B2B: Business to Business, is an acronym commonly used to indicate communication between business partners.

BIA: Business Intelligence Architecture, the combination of the BIG, user algorithm agents, group algorithm agents and the BiUI makes up this architecture.

BIG: Business Intelligence Generator, generates knowledge for UAs that request any kind of Business Intelligence generation.

BiUI: Business Intelligence UI, a UI driven by an agent, which can be used to interact with the LEOPARD application and to manipulate and interact with AAs inside the LEOPARD application.

DBA: Database Manager Agent, the agent that takes care of storing agent data to the underlying relation database.

DF: Directory Facilitator, acts as a Yellow Pages to lookup services offered by the LEOPARD application.

LEOPARD: Learning envirOnment Platform for Agent-based Repository Discovery, the application which mines e-resources.

LMA: Log Monitor Agent, the first-cut agent that parsed Web server log files as input of the data-mining.

MT: Message Transporter, the agent that dispatches messages to the right agents, a MT lives on every agent and MTs on agents communicate with each other.

MQ: Message Queue, where new messages for an agent arrive and are queued for processing by the agent.

NA: Node Agent, the agent that represents a specific resource in the application.

NS: Name Server, resolves an agent identifier to a hostname of a computer. The master NS lives on the first agent launched in the system, all other NS' are slave NS' which redirect requests to the master NS.

QoS: Quality of Service, a device that guarantee some kind of performance such as traffic delivery priority, speed, latency [59].

RT: Real-time, relating to computer systems that update information at the same rate they receive information [57].

SAS: Statistical Analysis System, a software package for the manipulation and statistical analysis of data.

SOAP: Simple Object Access Protocol, a lightweight XML based protocol for exchanging structured information in a decentralized, distributed environment [13].

T3RA: The top three recommender algorithm agent, a simple, lightweight agent that recommends to users based on the number of occurrences that a link has been traversed.

TR: Tolerance Relation, often a matrix which will specify if some attributes are relevant to each other or not. Relevance could be when, for example, two attributes are the same.

UA: User Agent, the agent that represents and tracks users inside the LEOPARD application.

UI: User Interface, a program that controls a display for a user (often a computer monitor) and that allows the user to interact with the system [57].

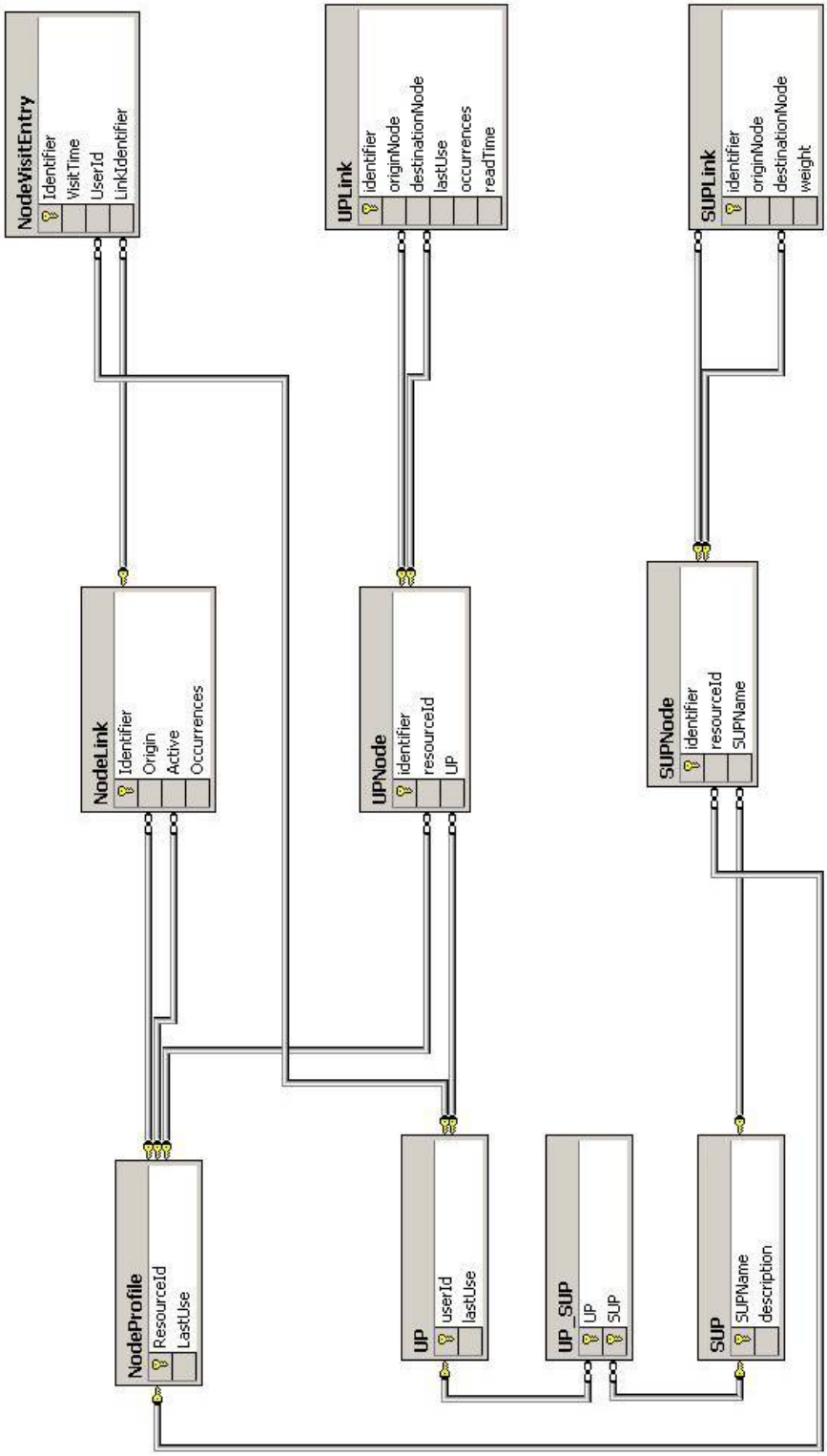
UP: User Profile, an object inside the UA which tracks the usage of the user throughout the system.

XML: Extensible Markup Language, a simple, very flexible text format derived from SGML to meet the challenges of electronic publishing. XML is getting more and more popular as a medium for exchanging a wide variety of data on the Web and elsewhere [7].

W3C: World Wide Web Consortium, develops interoperable technologies for the Web so it can reach its full potential [60].

WIER: Web Industrial Experience Resources, a collection of resources on the available to guide students through different phases of their industrial placement.

Appendix A: Database Schema



Appendix B: SQL tables explained

Table	Column	Usage
NodeProfile		**This table stores all the resources visited (the URL e.g.) with its last visit**
NodeProfile	ResourceId	A resource, in this case we use the URL of a Web server log
NodeProfile	LastUse	The DateTime it was last used
NodeLink		**This table makes a link between two resources (NodeProfiles). This represents a traversal of a user from one resource to another.**
NodeLink	Identifier	A GUID (a Unique ID that servers as primary key)
NodeLink	Origin	From which resource the link goes
NodeLink	Active	To which resource the link goes
NodeLink	Occurrences	How many times the link was traversed
NodeVisitEntry		**This table is an 'instance' of a NodeLink, in other words they store information about all the NodeLinks that occurred.**
NodeVisitEntry	Identifier	A GUID
NodeVisitEntry	VisitTime	The time of the traversal for a specific link
NodeVisitEntry	UserId	Which user was traversing this link (look at UP table)
NodeVisitEntry	LinkIdentifier	To which NodeLink this entry relates to

Table 8: Tables for Node Profiling

UP		**This table stores all the users in the system**
UP	userId	The unique user ID in the system (we can use a static IP address, or use a userId from a website with authentication)
UP	lastUse	The Date Time the user was last seen
UPNode		**This table stores which resources were accessed by a user**
UPNode	Identifier	A GUID
UPNode	resourceId	The resource (foreign key to NodeProfile (the URL)
UPNode	UP	The user that accessed the resource (Foreign key to UP)
UPLink		**This table stores how much times a link between two resources occurred and how long users spend on it (regardless of the user)**
UPLink	Identifier	A GUID
UPLink	originNode	The source of the link
UPLink	destinationNode	The target of the link
UPLink	lastUse	When this link was last used

UPLink	Occurrences	How many times this link was traversed
UPLink	readTime	The total time spend on the target resource
SUP		**Defines standard user profiles (e.g. expert users)**
SUP	SUPName	A name for the 'expert user'
SUP	Description	What this 'standard user represents'
UP_SUP		**Stores which UP could serve as a SUP**
UP_SUP	UP	The userID of the UP table
UP_SUP	SUP	The SUPName of the SUP table

Table 9: Table for User Profiling

SUPNode		**Acts the same as the UPNode**
SUPLink	**Identifier**	**A GUID**
SUPLink	originNode	The source of the link
SUPLink	destinationNode	The target of the link
SUPLink	Weight	A certain weighting of this link

Table 10: Table for Standard User Profiling

Appendix C: Code

1 BIGenerator.cs

```
// Architecture - BIGeneratorAg.cs
using System;
using MAP;
using Architecture.AlgArch;
using System.Collections;
using System.Threading;
using System.Diagnostics;
using System.Windows.Forms;

namespace Architecture.AlgArch
{
	/// <summary>
	/// Business Intelligence Generator.
	/// </summary>
	public class BIGeneratorAg : Agent
	{
		public static readonly string SERVICE =
"Business_intelligence_generation";

		// Message Filter
		private static readonly MessageCategory BIRequest = new
MessageCategory(
			MAP.Message.Natures.Request, Subjects.BIGeneration);
		private static readonly MessageCategory ClusterRequest = new
MessageCategory(
			MAP.Message.Natures.Request, Subjects.BIGeneration);

		private Hashtable algTable;

		public BIGeneratorAg() : base()
		{
			DirectoryFacilitator.GetDF().RegisterProvider(SERVICE,
GetId());
			algTable = Hashtable.Synchronized(new Hashtable());
			msgQueue.AddFilters(new MessageCategory[] {BIRequest});
		}

		protected override void Execute()
		{

			while(true)
			{
			Step1: //wait for birequest
				while (msgQueue.IsEmpty)
				{
					Thread.Sleep(SLEEP_TIME);
				}
				MAP.Message m = msgQueue.Dequeue();
				if(BIRequest.IsCategoryOf(m)) goto Step2;
				else goto Step3;

			Step2:
				Console.WriteLine("GENERATE ALG ACTIVITY");
				 RunRecommenderActivity(m);
				goto Step1;

			Step3:
				Console.WriteLine("TERMINATION ACT");
				goto Step1;
			}
		}
```

```
		private void RunRecommenderActivity(MAP.Message m)
		{
			new RecommenderActivity(this, algTable, m).Start();
		}

		public void ToChangePostMessage(MAP.Message m)
		{
			if (BIRequest.IsCategoryOf(m))
				Console.WriteLine("{0} requested for {1}", this,
SERVICE);
			else
				MessageNotUnderstood(m);
		}
	}


	class RecommenderActivity : Activity
	{
		private MAP.Message m;
		private Hashtable algTable;

		public RecommenderActivity(Agent _agent, Hashtable _algTable,
MAP.Message _m) : base(_agent) {
			algTable = _algTable;
			m = _m;
		}

		public override void Execute()
		{
			AgentId AgentAlgorithm = (AgentId) algTable[m.Sender];
			if(AgentAlgorithm == null)
			{
				Agent a = new RecommenderAlgAg();
				MAP.Message message_to_alg = new
MAP.Message(m.Recipient, a.GetId(), MAP.Message.Natures.Request,
					Subjects.BIGeneration, m);

	MessageTransporter.GetMT().PostMessage(message_to_alg);
				algTable.Add(m.Sender,a.GetId());
			}
			else
			{
				//NEW REQUEST FOR RUNNING ALG
				Console.WriteLine("ALG RUNNING");
			}
		}
	}
}
```

2 RecommenderAlgAg.cs

```
using System;
using System.Collections;
using System.Threading;
using System.Diagnostics;
using Architecture.Profiles;
using MAP;


namespace Architecture.AlgArch
{
	/// <summary>
	/// Summary description for RecommenderAlgAg.
	/// </summary>
	public class RecommenderAlgAg : Agent
	{
```

```
            public      static      readonly      string      SERVICE      =
"Business_intelligence_recommender_algorithm";
            private static readonly int TOP = 3;

            public RecommenderAlgAg() : base()
            {
                  DirectoryFacilitator.GetDF().RegisterProvider(SERVICE,
GetId());
            }

            protected override void Execute()
            {
                  while (true)
                  {
                        while (msgQueue.IsEmpty)
                              Thread.Sleep(10);
                        Message m = msgQueue.Dequeue();

                        //Extract the original message at this point
                        Message msg_UA_to_BIG = (Message) m.Content;
                        Console.WriteLine(msg_UA_to_BIG.ToString());

                        //Fetch the UP from the message
                        UP up = (UP) msg_UA_to_BIG.Content;

                        //Get the last accessed Node
                        INode n = up.GetLastNode();
                        Console.WriteLine("Last Node is: " + n.Resource);

                        //Get all the links to that node
                        ILink[] links = ((UPNode) n).GetLinks();

                        //Make    an    Hashtable    with    key=>value    =
occurence=>resource
                        Hashtable                OccHash                =
MakeOccurencesHashtableFor(links);

                        //Order the hashtable by the number of occurrences
                        int[] i = OrderHashtable(OccHash);

                        //Print out the top TOP number of resources from
the array
                        PrintTopOf(i, OccHash);

                  }
            }

            private Hashtable MakeOccurencesHashtableFor(ILink[] links)
            {

                  Hashtable occ = new Hashtable();

                  foreach(UPLink l in links)
                  {

                        Console.WriteLine("       Targets:       "       +
l.Destination.Resource);
                        Console.WriteLine(" occ: " + l.Occurrences);
                        ArrayList a = new ArrayList();
                        if(occ[l.Occurrences] == null)
                        {
                              a.Add(l.Destination.Resource);
                              occ.Add(l.Occurrences, a);
                        }
                        else
                        {
                              a = (ArrayList) occ[l.Occurrences];
```

```
                                a.Add(l.Destination.Resource);
                        }
                        a = null;
                }

                return occ;
        }

        private int[] OrderHashtable(Hashtable occ)
        {
                int[] i = new int[occ.Count];
                occ.Keys.CopyTo(i, 0);
                Array.Sort(i);
                Array.Reverse(i);

                return i;
        }

        private void PrintTopOf(int[] i, Hashtable OccHash) {
                Console.WriteLine("Top: " + TOP + " targets: ");
                int k = 0;

                for(int j = 0; j < i.Length && k < TOP ; j++)
                {
                        ArrayList a = new ArrayList();
                        a = (ArrayList) OccHash[i[j]];
                        for(int x = 0; x < a.Count; x++)
                        {
                                Console.WriteLine("*** " + a[x]);
                        }
                        k++;
                        a = null;
                }
        }

    }
}
```

3 Testing123.cs

```
// Architecture - Testing123.cs
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
using System.Text.RegularExpressions;
using System.Collections;
using System.Threading;
using System.Diagnostics;

using MAP;
using Architecture.Profiles;
using Architecture.NodeArch.Profiles;
using Architecture.AlgArch;

namespace Architecture
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Testing123
    {
        static void Main(string[] args)
        {

    Console.WriteLine("##################################################
##############################");
```

```
                    Console.WriteLine("1. Clean DB");
                    Console.WriteLine("2. Fill Sample Data");
                    Console.WriteLine("3. Parse the hardcoded logfile!");
                    Console.WriteLine("4. Test the BIGenerator");

      Console.WriteLine("##################################################
############################");
                    string choice = Console.ReadLine();

                    if(choice == "1")
                          CleanDB();
                    else if(choice == "2")
                          FillSampleData();
                    else if(choice == "3")
                          ParseWIERLogFile();
                    else if(choice == "4")
                          TestBIG();
                    Console.WriteLine("####   END   OF   TEST   -   Enter   to
continue");
                    Console.ReadLine();

             }

             private static void CleanDB()
             {
                    Middleware.Install();
                    DBManagerAg dbA = new DBManagerAg();
                    NodeArch.DBManagerAg dbB = new NodeArch.DBManagerAg();
                    Console.WriteLine("####  END  OF  CLEANING  -  Enter  to
continue");
                    Console.ReadLine();
                    Middleware.Uninstall();
             }


             private static void FillSampleData()
             {
                    Middleware.Install();
                    DBManagerAg dbA = new DBManagerAg();
                    NodeArch.DBManagerAg dbB = new NodeArch.DBManagerAg();

                    // Nodes
                    Thread.Sleep(5000);

                    ResourceId              a              =              new
ResourceId("au.edu.monash.csse.html.1");
                    ResourceId              b              =              new
ResourceId("au.edu.monash.csse.html.2");
                    ResourceId              c              =              new
ResourceId("au.edu.monash.csse.html.3");
                    ResourceId              d              =              new
ResourceId("au.edu.monash.csse.html.4");
                    ResourceId              e              =              new
ResourceId("au.edu.monash.csse.html.5");

                    // NodeProfiles
                    NodeProfile nPa = NodeProfile.CreateNodeProfile(a);
                    NodeProfile nPb = NodeProfile.CreateNodeProfile(b);
                    NodeProfile nPc = NodeProfile.CreateNodeProfile(c);
                    NodeProfile nPd = NodeProfile.CreateNodeProfile(d);
                    NodeProfile nPe = NodeProfile.CreateNodeProfile(e);

                    //NodeLinks
                    NodeLink aTob = nPa.AddLinkFrom(b);
                    NodeLink aToc = nPa.AddLinkFrom(c);
                    NodeLink aTod = nPa.AddLinkFrom(d);

                    // Create a SUP
```

```
                    string supName = "My SUP";
                    SUP sup = SUP.GetSUP(supName);
                    if (sup == null) sup = SUP.CreateSUP(supName, "A dummy
description");
                    sup.AddLinkBetween(a, b, new DateTime(2004, 1, 1));
                    sup.AddLinkBetween(b, c, new DateTime(2004, 1, 1));
                    sup.AddLinkBetween(d, b, new DateTime(2004, 1, 1));
                    sup.AddLinkBetween(b, d, new DateTime(2004, 1, 1));

                    // Create an UP
                    UP up = UP.CreateUP(new UserId("1.1.1.1"), new string[]
{supName});
                    up.AddLinkBetween(a, b, new DateTime(2004, 1, 1));
                    up.AddLinkBetween(b, c, new DateTime(2004, 1, 1));
                    up.AddLinkBetween(c, b, new DateTime(2004, 1, 1));
                    UPLink link = up.GetLinkBetween(b, c) as UPLink;
                    link.Traverse();
                    link.AddReadTime((long)1500);


                    DateTime                    timestamp                    =
NavigationActContent.ConvertToDateTime((string)"27/03/2004 00:00:00");
                    Console.WriteLine(timestamp);
                    NodeVisitEntry                  entry                    =
NodeVisitEntry.CreateVisitEntry(timestamp, up.User, aTob.Identifier);

                    Console.WriteLine("#### END OF LOADING DATA - Enter to
continue");
                    Console.ReadLine();

                    Middleware.Uninstall();
             }


             private static void ParseWIERLogFile()
             {
                    Middleware.Install();
                    DBManagerAg dbA = new DBManagerAg();
                    NodeArch.DBManagerAg dbB = new NodeArch.DBManagerAg();

                    Thread.Sleep(5000);

                    string FILE_NAME = "WIER2001.txt";
                    //string FILE_NAME = "ex020721.log";
                    if (!File.Exists(FILE_NAME))
                    {
                           Console.WriteLine("{0}    does    not    exist.",
FILE_NAME);
                           return;
                    }
                    StreamReader sr = File.OpenText(FILE_NAME);
                    String input;

                    int records = 0;
                    String prevRes = "NULL";
                    String prevSessid  = "0";
                    while ((input=sr.ReadLine())!=null)
                    {
                           records++;
                           if(records % 1000 == 0)
                                  Console.WriteLine("Records    done:    "    +
records);
                           if(records % 10000 == 0)
                           {
                                  dbA.SaveUserDB();
                                  dbB.SaveNodeDB();
                           }
```

```
                        if(input.Trim() != "" && input.Substring(0,1) !=
"#")
                        {
                                //Console.WriteLine(input);
                                String[]              fields              =
input.Split("\t".ToCharArray());
                                String userid     = fields[1].ToString();
                                String sessid     = fields[2].ToString();
                                String resourceid = fields[7].ToString();
                                //String prevRes    = fields[7].ToString();

                                //String   host                           =
fields[10].ToString();
                                String host         =
"wier.csse.monash.edu.au";
                                String[]   date                           =
fields[11].ToString().Split("/".ToCharArray());
                                String[]   time                           =
fields[12].ToString().Split(":".ToCharArray());

                                if(prevSessid != sessid) //NEW SESSION SO
PREVIOUS RESOURCE IS "NOWHERE"
                                {
                                        prevRes = "NULL";
                                }

                                resourceid = "http://" + host + resourceid;
                                //Console.WriteLine("SessionId: " + sessid
+ ", PrevSessionId: " + prevSessid);
                                //Console.WriteLine("ResourceId:     "     +
resourceid + ", " + ", UserID: " + userid + ", Previous: " + prevRes +
",time: " + time);
                                //Console.ReadLine();

                                ResourceId r = new ResourceId(resourceid);
                                ResourceId p = new ResourceId(prevRes);
                                NodeProfile              nP              =
NodeProfile.CreateNodeProfile(r);
                                UserId     u = new UserId(userid);

                                UP up = UP.GetUP(u);

                                if(up == null)
                                        up = UP.CreateUP(u);

                                //Console.WriteLine("From: " + prevRes + "
To: " + resourceid);
                                //Console.ReadLine();
                                if(prevRes != "-")
                                {
                                        NodeLink l = nP.AddLinkFrom(p);

                                        //System.Globalization.CultureInfo
info =
                                        //     new
System.Globalization.CultureInfo("en-US", false);

                                        //System.Globalization.Calendar
calendar = info.Calendar;

                                        DateTime     timestamp     =     new
DateTime(Int32.Parse(date[2]),Int32.Parse(date[0]),Int32.Parse(date[1]),Int3
2.Parse(time[0]),Int32.Parse(time[1]),Int32.Parse(time[2])/*, calendar*/);

                                        NodeVisitEntry        entry        =
NodeVisitEntry.CreateVisitEntry(timestamp, up.User, l.Identifier);
                                        UPLink ul = null;
```

```
                                        ul = up.GetLinkBetween(p, r) as
UPLink;
                                        if(ul == null)
                                               ul = up.AddLinkBetween(p, r,
timestamp) as UPLink;
                                        else
                                        {
                                               ul.Traverse(timestamp);
                                               ul.AddReadTime(123);
                                        }
                                 }
                                 prevRes = resourceid;
                                 prevSessid = sessid;
                                 //Console.ReadLine();
                          }
                          else
                          {
                                 Console.WriteLine("Skipped line");
                          }
                   }
                   Console.WriteLine ("The end of the stream has been
reached.");
                   sr.Close();
                   Middleware.Uninstall();
            }


            private static void TestBIG()
            {
                   Middleware.Install();
                   new BIGeneratorAg();
                   new DBManagerAg();
                   new NodeArch.DBManagerAg();


      Console.WriteLine("#####################################################
##############################");
                   Console.WriteLine("WAIT FOR DB TO INITALIZE");

      Console.WriteLine("#####################################################
##############################");
                   string foo = Console.ReadLine();

                   UP up1 = UP.GetUP(new UserId("1041"));
                   UP up2 = UP.GetUP(new UserId("1061"));
                   UP up3 = UP.GetUP(new UserId("1081"));

                   NodeProfile np1 = NodeProfile.GetNodeProfile(new
ResourceId("http://wier.csse.monash.edu.au/live/admin/index.php"));

                   AgentId generator =
DirectoryFacilitator.GetDF().FindProviderOf(
                          BIGeneratorAg.SERVICE);
                   Debug.Assert(generator != null);

                   Message m1 = new Message(AgentId.NewId(), generator,
Message.Natures.Request,
                          Subjects.BIGeneration, up1);
                   MessageTransporter.GetMT().PostMessage(m1);

                   Thread.Sleep(1000);

                   //Message m2 = new Message(AgentId.NewId(), generator,
Message.Natures.Request,
                   //     Subjects.BIGeneration, up2);
                   //MessageTransporter.GetMT().PostMessage(m2);

                   Thread.Sleep(1000);
```

```
                    //Message m3 = new Message(AgentId.NewId(), generator,
Message.Natures.Request,
                    //     Subjects.BIGeneration, up2);
                    //MessageTransporter.GetMT().PostMessage(m2);

                    //Thread.Sleep(1000);

                    //Message m4 = new Message(AgentId.NewId(), generator,
Message.Natures.Request,
                    //     Subjects.BIGeneration, np1);
                    //MessageTransporter.GetMT().PostMessage(m4);

                    Console.ReadLine();

      Console.WriteLine(DirectoryFacilitator.GetDF().ToString());
                    Middleware.Uninstall();
              }
       }
}
```

Appendix D: Project Management

A rough and minimal ad-hoc project plan was setup before starting the project. The project plan as defined at the project start is shown in Table 11.

Period	Activity
15th March	Project start
April	Read data-mining papers / Literature Review
8th May – End May	Planned holiday break
June	BIA design and implementation / Literature Review
July	Write thesis
August	Last minute changes / Print and bundle thesis
20th August	Project finish

Table 11: Project Plan

The actual project plan was a bit shuffled due to unforeseen events. The actual actions taken throughout the project were recorded in a spreadsheet and are presented in Table 12

Period	Activity	
15th March	Project start	
17th March	Meeting Christine & Judy: LEOPARD and possible projects	READING PAPERS
21st March	Meeting Judy: data-mining, clustering, classification	
28th March	Meeting Christine & Judy: - Assessment of using Joe's algorithm in LEOPARD - Database tables need clarification - We need real life data in the database	
30th March	Development of method for loading data into database (source: WIER [61] database), Appendix C	
31st March	Meeting Christine, Judy & Joe: The datasets in the database are too small, creation of DataViews in SQL as algorithm data source	LITERATURE
2nd April	- Produced an explanation of the tables in LEOPARD (Appendix B) - Produced a database schema for a better overview of the data (Appendix A)	
14th April	Proposed the Business Intelligence Architecture, User Algorithm Agents and Group Algorithm Agents	
5th May	Demonstration of Joe's algorithm	
8th May- 21st May	Holiday break (Los Angeles/Belgium)	

21st May- 30th May	Meeting Christine & Judy: Project assessment, proposed to start on implementation parts that do not need Joe's algorithm	
30th May – 12th June	Implementation of BIGenerator Meeting Christine & Judy: Proposed to implement own algorithm	
12th June – 30th June	Implementation of Recommender Agent Meeting Judy: Received papers of Joe's algorithm and updated literature review Meeting Christine & Judy: Assessment of current BIA and Literature Review	R E V I E W
1st July	Start on thesis report and merge literature review	
31st July	End thesis report	
1st August – 14th August	Commit changes to thesis Setup conference room Print thesis and bundle Prepare presentation for thesis defense	
19th August	Flight to Europe	
26th August	Thesis Defense	

Table 12: Actual Project Plan Actions

References

[1] Olivier Constant, *An Agent-Based Platform for Assisting Repository Navigation and Administration* (MSc. Thesis), Vrije Universiteit Brussel, Belgium

[2] Marc Vanbrabant-Cattoor, Specialization Training, *Genericity Checking*, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France

[3] Zahia Guessoum, Jean-Pierre Briot, *DIMA, From Active Objects to Autonomous Agents*. IEEE Concurrency, vol. 7(3), pp. 68-76, 1999.

[4] Phillip M. Hallam-Baker, Brian Behlendorf, *Extended Log File Format*, http://www.w3.org/TR/WD-logfile.html

[5] Tim Berners-Lee, *Uniform Resource Locators*, RFC1738, CERN, http://rfc.net/rfc1738.html

[6] *Standard for the format of Arpa Internet Text Messages*, RFC822, Revised by David H. Crocker, Dept. of Electrical Engineering, University of Delaware, Newark, DE 19711, http://rfc.net/rfc822.html

[7] *Extensible Markup Language*, XML, http://www.w3.org/XML/

[8] *Cascading Style Sheets*, CSS, http://www.w3.org/Style/CSS/

[9] *The Extensible Stylesheet Language Family*, XSL, http://www.w3.org/Style/XSL/

[10] *XSL Transformations*, XSLT, http://www.w3.org/TR/xslt

[11] *XML Path Language*, XPath, http://www.w3.org/TR/xpath

[12] *XSL Formatting Objects*, XSL-FO, http://www.w3.org/TR/xsl/

[13] *Simple Object Access Protocol*, SOAP 1.1, http://www.w3.org/TR/soap/

[14] Tim Berners-Lee, *A roadmap to the Semantic Web*, Sep 1998, http://www.w3.org/DesignIssues/Semantic.html

[15] Google, Googlebot, Google's Web Crawler, http://www.google.com/bot.html

[16] S. Schechter (Harvard University, 29, Oxford St., Cambridge, MA 02139, USA), M. Krishman (Microsoft, One Microsoft Way, Redmond, WA 98052, USA), M. D. Smith (Harvard University, 29, Oxford St., Cambridge, MA 02139, USA), *Using path profiles to predict HTTP requests*

[17] Xiabbin Fu, Jay Budzik, Kristian J. Hammond, *Mining Navigation History for Recommendation*, Infolab, Northwestern University, 1890 Maple Avenue, Evanston, IL 60201, USA

[18] Hui-Min Chen, M. D. Cooper, *Using clustering techniques to detect usage patterns in a web-based information system*, School of Information Management and Systems, University of California at Berkeley, Berkeley, CA 94720-4600, USA

[19] *Address Allocation for Private Internets*, RFC1918, http://rfc.net/rfc1918.html

[20] *The IP Network Address Translator (NAT)*, RFC1631, http://rfc.net/rfc1631.html

[21] Osmar R. Zaïane, *Web Usage Mining for a Better Web-Based Learning Environment*, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada

[22] Raymond Kosola, Hendrik Blockeel, *Web Mining Research: A survey*, Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

[23] Microsoft Development Network, *The GUID Structure*, http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemguidclasstopic.asp

[24] Bettina Berendt (Institute of Pedagogy and Informatics, Faculty of Philosophy IV, Humboldt University Berlin, 10117 Berlin, Germany), Myra Spiliopoulou (Institute of Information Systems, Faculty of Economics, Humboldt University Berlin, 10178, Berlin, Germany), *Analysis of navigation behavior in web sites integrating multiple information systems*

[25] José Borges, Mark Levene, *Data Mining of User Navigation Patterns*, Department of Computer Science University College London, Gower Street, London WC1E 6BT, UK

[26] Lara D. Catledge, James E. Pitkow, *Characterizing browsing strategies in the world wide web*, Computer Networks and ISDN systems, April 1995, vol. 27(6), pp. 1065-1073

[27] *HTTP State Management Mechanism*, RFC2109, http://rfc.net/rfc2109.html

[28] *PHP and session management*, http://au.php.net/function.session-start

[29] .NET Framework Developer's Guide, Session State, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsessionstate.asp

[30] Corin R. Anderson, Pedro Domingos, Daniel S. Weld, *Relational Markov Models and their Application to Adaptive Web Navigation*, Dept. of Comp. Sci. & Eng. University of Washington, Seattle, WA, USA

[31] Paolo Giudici, *Applied Data Mining*, John Wiley & Sons, Ltd ISDN 0-470-84679-8, Chapter 8; Web Clickstream Analysis

[32] Jian Pei, Jiawei Han, Behzad Mortazavi-asl, Hua Zhu, *Mining Access Patterns Efficiently from Web Logs*, School of Computing Science, Simon Fraser University, Canada

[33] Sasha A. Barab, Brett R. Fajen, Jonna M. Kulikowich, Michael F. Young, *Assessing hypermedia navigation through pathfinder: prospects and limitations*, University of Connecticut, Journal of educational computing research, vol. 15(3), pp. 185-205, 1996

[34] Sasha A. Barab, Bruce E. Bowdish, Kimberly A. Lawless, *Hypermedia Navigation: Profiles of Hypermedia Users*, ETR&D, vol. 45(3), pp. 23-41; 1997

[35] Joe H. Ward, *Hierarchical Grouping to optimize an objective function*, Journal of American Statistical Association, 58(301), 236-244, 1963

[36] A. El-Hamdouchi, P. Willett, *Hierarchic Document Clustering Using Ward's Method*, Sheffield University, Western Bank, Sheffield, S10 2TN, UK

[37] ZA Spring Seminar2002: Cluster Analysis, Chapter 4, http://www.soziologie.wiso.uni-erlangen.de/koeln/

[38] J. MacQueen, *Some methods for classification and analysis of multivariate observation*, Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, Vol. 1; pp. 281-297, 1967, University of California Press, Berkeley.

[39] A. P. Dempster, N. M. Laird, D. B. Rubin, *Maximum likelihood from incomplete data via the EM algorithm*, Journal of the Royal Statistical Society Series B, vol. 39(1), pp. 1-38, 1977.

[40] Weka, Waikato Environment Knowledge Analysis, University of Waikato, http://www.cs.waikato.ac.nz/~ml/index.html

[41] Shiro Ikeda, *Acceleration of the EM algorithm*, The Institute of Physical and Chemical Research (RIKEN), Saitama, 351-01 Japan

[42] T.A. Louis, *Finding the observed information matrix when using the EM algorithm*, Journal of the Royal Statistical Society B 44, 226-233, 1982

[43] Kenneth Lange, *A quasi-Newton acceleration of the EM algorithm*, University of Michigan, Statistica Sinica 5, pp. 1-18, 1995, http://www.stat.sinica.edu.tw/statistica/oldpdf/A5n11.pdf

[44] GrbTS, Gamma Ray Burst Tool Shed, *The EM Algorithm for Unsupervised Clustering*, Minnesota State University, Mankato, http://grb.mnsu.edu/grbts/doc/manual/Expectation_Maximization_EM.html#sec:em

[45] The FASTCLUS procedure, http://www.id.unizh.ch/software/unix/statmath/sas/sasdoc/stat/chap27/

[46] The CLUSTER procedure, http://www.id.unizh.ch/software/unix/statmath/sas/sasdoc/stat/chap23/

[47] P.L. Zhou, School of Computer Science and Software Engineering, Monash University, Melbourne Australia, Z.H. Wang, School of Computing and Mathematics, Deakin University, Australia, C. Mingins, School of Computer Science and Software Engineering, Monash University, Melbourne Australia, *Attribute Reducts of Tolerance Information System*

[48] P.L. Zhou, C. Mingins, *An effective parallel attribute reduct algorithm based on relation matrix*, School of Computer Science and Software Engineering, Monash University, Melbourne, Australia

[49] Zdzislaw Pawlak, Warsaw University of Technology, Warsaw, Poland, Jerzy Grzymala-Busse, University of Kansas, Lawrence, Roman Slowinski, Poznan University of Technology, Poznan, Poland, Wojciech Ziarko, University of Regina, Sask., Canada, *Rough Sets*, Communications of the ACM, vol. 38(11), pp. 88-95, 1995

[50] Z. Pawlak, *Rough sets*, lnternational Journal of Computing lnformation Science 11, pp. 341-356, 1982

[51] A. Skowron, J. Stepaniuk, *Generalized approximation spaces*, In: Lind T Y ed. Conference Proceedings of the Third Internation Workshop on Rough Sets and Soft Computing (RSSC 94), San Jose, California, USA, 1994, pp. 156-163

[52] R. Agrawal, R. Srikant, *Mining Sequential patterns*, In Proc. 1995, International Conference of Data Engineering, pp. 3-14, Taipei, Taiwan

[53] R. Young, *Path based compilation*, Ph.D. thesis, Division of Engineering and Applied sciences, Harvard University, 1997

[54] Denis V. Ivanov, Dr. Eugene P. Kuzmin, Dr. Sergey V. Burtsev, *Progressive Image Compression Using Binary Trees*, Mathematics and Mechanics Dept., Moscow State University, Vorobyovy Gory, Moscow, Russia, 119899,
http://www.cgg.ru/PROGRESSIVE/PROGRESSIVE.htm

[55] XFS: A High Performance Journaling System, SGI,
http://oss.sgi.com/projects/xfs/

[56] ReiserFS: Journaling File System for Linux based on balanced tree algorithms,
http://www.namesys.com/v4/v4.html#tree_design

[57] Wordnet, Princeton University, http://www.cogsci.princeton.edu/~wn/

[58] Tree concepts: Binary Trees, http://www.cs.aucegypt.edu/~mudawwar/csci210/slides/trees.pdf

[59] Puredata.com, http://www.puredata.com/manual/backboneswiches/appendix/glossary.html

[60] W3C, http://www.w3c.org/

[61] J. Ceddia, S. Tucker, C. Clemence, A. Cambrell, 2001. *WIER – Implementing artifact reuse in an educational environment with real products.* Proceedings of the Thirty-first Annual Frontiers in Education Conference, Reno, Nevada.