B.Comp. Dissertation

Analytics Systems for Anti-Money Laundering

By

Du Linxiang

Department of Computer Science
School of Computing
National University of Singapore

2010/2011

B.Comp. Dissertation

Analytics Systems for Anti-Money Laundering

By

Du Linxiang

Department of Computer Science
School of Computing
National University of Singapore

2010/2011

Project No: H114290

Supervisor: Prof Limsoon Wong

Deliverable: Report 1 Volume

Abstract

In this paper, we introduce the implementation of a data mining system we developed to search for patterns from large sets of relational databases. In the database that we obtained from a bank, there is large amount of information of the bank's clients and companies that the clients are related to, as well as relations among the clients and companies (the sensitive and personal information is anonymized for privacy issues). The patterns that our system searches for are user-defined. The definition and limitations of the patterns will be discussed in this paper as well.

Our system is developed to support an "Anti-money laundering system" that the banks can use to search for a pattern of transactions and client-client or client-company relations in order to detect potential money laundering activities.

Key words:

Data Mining, Anti-Money Laundering, Pattern Recognition

Implementation Software and Language:

Apache HTTP Server, MySQL database, PHP

Acknowledgement

This dissertation would not have been possible without the guidance and help from several individuals.

First of all, I owe my deepest gratitude to my supervisor Prof Limsoon Wong for giving me the opportunity to do this project as well as his guidance and valuable suggestions and corrections throughout this project.

I would like to thank my advisors Soh Cheng Lock Donny and Tew Kar Leong. They patiently guide us through the project, and continuously advice and support us on this project.

I would also like to thank my colleague Ruchi Bajoria. Without the efforts from her side and our good cooperation, I would not be able to complete this project.

Table of Contents

List of Figures

List of Tables

1 Introduction

Data mining applications are deployed in a wide range of business fields, especially in financial banking, telecommunication, and the World Wide Web that have to deal with extensive amount of data. Simple database querying is far from enough for information retrieval in those business areas. Data mining is used to extract more complex desired information. The desired information is usually presented as a pattern. Thus pattern recognition, although not equivalent to data mining, is usually the framework for data mining.

There are mainly two approaches to mining relational information: logic-based approaches and graph-based approaches [2]. Logic-based approaches involve logical definition of a data pattern which is usually more complicated than that of graph-based approaches. They allow recursions and variables in defining a data pattern, which are not easy to implement with graph-based approaches. With certain limitations, graph-based data mining is however more data-driven.

Banks collect detailed personal information from their clients as well as information such as the relations between clients and the association between clients and certain companies. Data mining systems are used to extract interesting and valuable information from the large database. Relations among clients and companies, accompanied with monetary transaction histories, could be used as effective indication of suspicious money laundering activities.

We develop the system where banks could define a pattern with prior experience or knowledge that could be used as an indicator of money laundering activity, and our system searches for the clients and companies that matches such a pattern from the database.

First and foremost, we have to define a pattern in a visualized and human-readable way in the front end. The pattern is to be defined by an end user, therefore, the representation of a pattern need to be intuitive but also expressive. It has to be intuitive enough so as to make it easier for users to convey their ideas with the pattern. And it has to be expressive enough so that whatever the users want to express can be represented by the pattern. We use graphs to represent patterns. Graphs are expressive in describing relations between objects, and it is easy to understand as well. The pattern information encapsulated by the graph would be sent to the backend and get processed and interpreted. How the patterns are interpreted and processed is the core part of the backend. Finally, the backend would return the whole set of data that matches the user-defined pattern.

Users are allowed only to query for patterns, but not write into the database; therefore multiple users can access the system at the same time.

2 Background

2.1 Problem Definition

Money laundering activities can be detected by looking for certain traits in a relation or activity pattern. For example, purchasing a luxury car in a foreign country where a person has no purchase record in might be an indication of money laundering. Of course, there are other various kinds of patterns banks or governments would use to detect money laundering activities. After all, it is a data mining problem. What makes it different from other data mining problems is that we need to mine a dynamic pattern (graph). Besides the data mining process, we need to create an interface for users to easily define a pattern they want to search for.

2.2 Objectives

The objective of this whole project is to provide an interactive system for mining relational data patterns. In order to achieve this objective, there are several tasks we need to accomplish. First of all, we need to come up with a way to represent a pattern in a graphical and user-friendly way. Secondly, we need to pass the pattern to the backend and get interpreted and queried. Thirdly, the querying results have to be passed back to the frontend. Finally, the frontend will display the results in a graphical way.

Our major task here is the second step, where we need to come up with an algorithm to query for a relational pattern. In my part of the project, which is the backend of the system, the backend will be expecting XML data representing a relational pattern, and we will design an algorithm to efficiently interpret the pattern into a query and get the results to return to the frontend.

2.3 Related Works

A lot of research work has been done on graph data mining. Graph data mining is the task of finding novel, useful, and understandable patterns in a graph representation of data [1]. In a lot of works, graph data mining is used for finding frequently occurring structures, such as in molecular biology, people are interested in finding certain structures comprising of some elements. Diane and Lawrence et al. developed the Subdue system to find frequent patterns. Subdue system is the process of incrementally compressing frequently occurring substructures into units until reaching the pattern occurring frequency. Mohammed J. Zaki proposed a TreeMiner algorithm for finding frequent structures [5]. This algorithm performs

DFS to find frequent subtrees, and this algorithm uses strings to encode the tree structures.

Besides graph data mining from database, there are also quite some research done in mining XML data and RDF data [2].

3 Architecture

The system is comprised of three components, the frontend, the backend search engine and the database. The communication between the frontend and the backend search engine is through XML data passing. The data mining modules of the backend process the data and pass the results to the GUI frontend. The architecture of our system is depicted as follows.

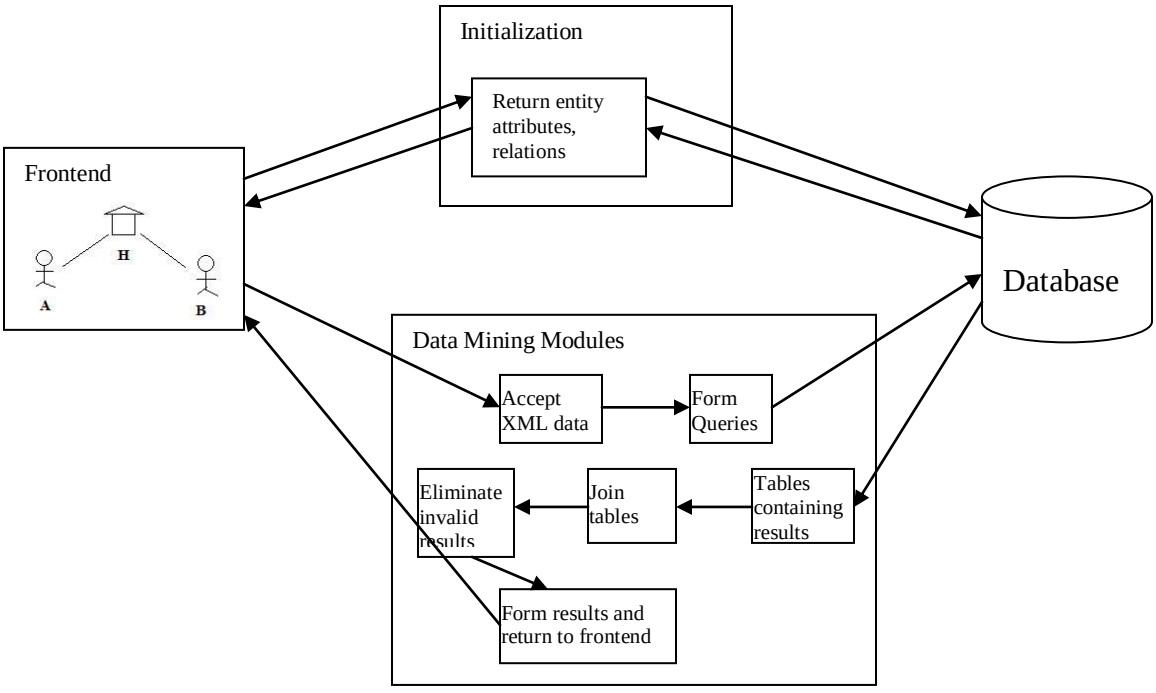


Figure 1. The architecture of the system

4 Data Representation

The database stores large amount of information of the clients and companies, and some of the information is shared by multiple entities. In order to minimize redundancy and to improve efficiency, the data is stored in third normal form in the database. And the relations between clients and companies are stored in binary form. Data in the front-end defining the search pattern has to be well-defined as well. The data representing the search pattern has to be informative and comprehensive. That is for the purpose of easy processing of the pattern at the backend.

4.1 Data Representation in the Database

The database mainly consists of tables of clients, companies, and one-to-one relations among the clients and companies. In order to prevent data redundancy, we decompose the database into smaller and well-structured tables [3]. For example, a client may have duo citizenship; therefore, the information of passports is stored in a separate table.

Figure 2b shows how data is stored in our system. All the one-to-many information is stored in separate tables. Each client and each company has a unique ID (UID), however, the UID of client and UID of companies might overlap since they are from different domains, and therefore, we create another list of sequential IDs to represent both clients and companies. We use this set of IDs to keep the relation information in the relation table.

Person

ID	Last Name	First Name	Address	Passport
1	Smith	David	54 Madison Avenue 34501 Chicago	100003945(US)
2	Eriksen	Justin	16 Pacific Street 98195 Seattle	71057032(US)
3	Roberts	Jean	23 Columbus Ave 44870 Sandusky	165006037(US) 22200345(CA)

(a)

Person

ID	Last Name	First Name
1	Smith	David
2	Eriksen	Justin
3	Roberts	Jean

Address

ID	Address	City	State	Country
A1	54 Madison Avenue 34501	Chicago	IL	US
A2	16 Pacific Street 98195 Seattle	Seattle	WA	US
A3	23 Columbus Ave 44870 Sandusky	Sandusky	OH	US

Passport

ID	Passport
1	100003945
2	71057032
3	165006037
3	22200345

Address

ID	AddressID
1	A1
2	A2
3	A3

(b)

Figure 2. Data modeling in our system

We store the relations between clients and companies in normalized binary form. This is to ensure that the structures of the relations are well-organized and consistent, and that the way of defining a pattern is more dynamic and flexible. For example, in real life, we would have a non-binary relation between a house and some people as follows:

House	Owner	Tenant	Subtenant
H	O	T	S .

Table 1. Non-binary relation

This relation pattern is represented by the graph below. Every two of the four entities are constrained by a certain relationship. For example, person T and person S have the tenant-subtenant relationship. This is a strictly constrained pattern; the pattern is satisfied only when the relations between any two of the entities are satisfied at the same time. Therefore, the following graph is a complete graph.

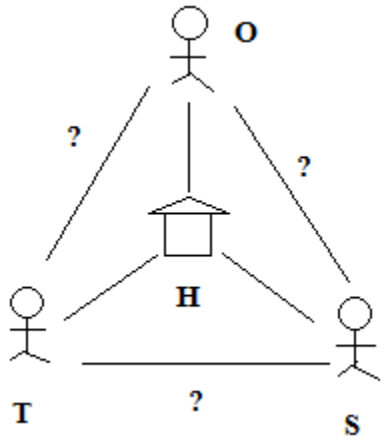


Figure 3. Graph representation of a non-binary relation

Now, let's say the user wants to define a more loosely constrained pattern, which is expressed as follows.

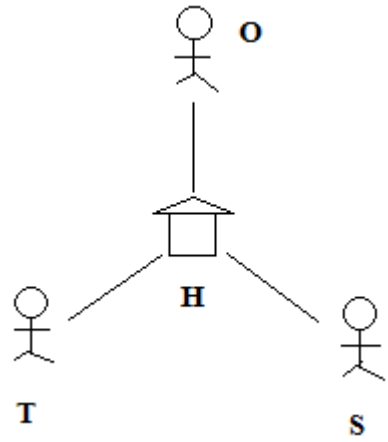


Figure 4. Graph with reduced constraints (T, O and S are not directly related)

In this relation pattern, the relations among the three parties are not constrained, i.e. person S may or may not have the tenant-subtenant relation with person T. Same rule applies to any two of the three parties. This definition of a graph will return a superset of that of the previous graph.

Saying that person O and person T does not have an owner-tenant relation might not be reasonable since O being the owner and T being the tenant normally binds the owner-tenant relation. However, the subtenant S and the tenant T need not have the tenant-subtenant relation. It is true when S rented house H from another tenant of the house T_2. In this case, person S is the subtenant of house H, and person T is the tenant of house H, while there is no tenant-subtenant relation between person S and T. Therefore, this graph is a representation of the relations that follow.

House	Owner	Tenant	Subtenant
H	O	−	−
H	−	T	−
H	−	−	S

Table 2. Non-binary relations representing a loosely defined relation pattern

The relation table above can be decomposed into binary relations as follows.

House	Person	Relation
H	O	Owner
H	T	Tenant
H	S	Subtenant

Table 3. Binary relations decomposed from non-binary relations

If the user does want to make a stronger constrained pattern, he could define the relations as shown in table 4. This set of binary relations is a breakdown of the non-binary relation in table 1.

House	Person	Relation
H	O	Owner
H	T	Tenant
H	S	Subtenant
O	T	Owner-tenant
O	S	Owner-subtenant
T	S	Tenant-subtenant

Table 4. Binary relations representing fully related relation pattern

Therefore, relations among multiple entities can always be interpreted in binary form. Representing the relations in binary form makes the system more dynamic. If we want to import a new relational database into our system, we can always decompose the non-binary relations into binary relations to be compatible with our system and algorithm. However, the complexity of a relation graph would drastically increase if we want to define a fully related relation pattern. For example, if we wish to define a relation pattern like the one of table 1, the user will have to draw a complete graph as Figure 3, and the backend will search for the patterns based on the relations as table 4 instead of table 1. The complexity and difficulty in searching increase as well.

4.2 Data Representation in the Frontend Graph

A user-defined graph can come in three types: a single entity, a one-to-one relation, and a relation graph with more than two entities. Suppose the whole relational database is represented by a huge graph with nodes being the entities and edges

being the relations. We look for patterns that match the user-defined graph from the entire graph.

The first two types of queries are more trivial and straightforward. We simply search for a person or company that matches user's description, and that are bonded by a specific relation if it is for a one-to-one relation search. They do not have dynamic structures, and the variables expected by the backend are limited. However working on complex relation graph queries presents more challenges for frontend-backend communication. Even with the pattern-matching interpretation of a complex relation query, there are some minor design problems we need to tackle. The visualized pattern of relations on frontend has to be passed onto backend in a proper structure that preserves all information of the relations. Each connection between two entities in the frontend graph represents a relation that the backend is going to search for. A chain of relations require the backend to search for each relation and join the results in proper manner. The backend has to be aware of the entities on which two relations join. For example, a graph of a house connected to two people can be decomposed into two person-house relations. What's more, the backend has to be aware that these two relations relates to each other on the house entity rather than the person entity, i.e. we are searching for one house related to two persons instead of one person related to two houses. Therefore, extra information about relations between two relations must be defined as well.

4.2.1 Two Ways of Interpreting a Graph

Besides creating an information passing protocol between the frontend and backend, there is another problem we need to deal with. For a one-to-one relation or a relation graph, there are usually more than one ways to interpret users'

intention with such a graph. With different interpretation, the results of a query might be different. Therefore, we need to decide on a certain way to interpret a given graph.

A graph of a house connected to two people can be interpreted in two ways:

1. A house that has exactly these two people living in it.
2. A house in which these two people live.

The graph below can be interpreted in either one of these two ways:

1. H is a house where only A and B live.
2. H is a house where A and B live. (There might be other people living in the house as well.)

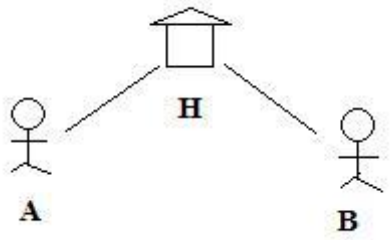


Figure 5. Interpretation of a house with two persons

Obviously, these two different interpretations will return different results. The result returned from the first interpretation is always a subset of that of the second interpretation. We need to pick one of these two ways of interpretation for our design. If we take the first interpretation, we will have a problem when users want to define a graph with the second interpretation, since what we can get is always a subset of what the users want. On the other hand, if we use the second interpretation, when the users want to define a graph with the first interpretation, we can use the graph below to solve the problem.

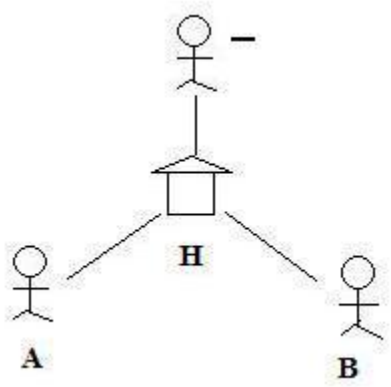


Figure 6. Use the second interpretation to resolve a graph intended with the first interpretation

The person with sign "—" can be used to indicate non-existence of a third person in the relation other than A and B (which conforms to the first interpretation). The graph with three persons will return results where there are more than or exactly three persons (including A and B) living in the house. We subtract these results from the set of results from Figure 5, and then we can get the set of results conforming to the first interpretation for Figure 5.

The second way of defining a graph has some advantages over the first. It is more powerful in defining a pattern. It can even be used to define a pattern that conforms to the first definition (e.g. Figure 5)

Another problem with the first interpretation is that if the user is interested in the house entity in the graph, we also need to decide whether the house is the only one that is related to A and B. In other words, the user is querying for the house that has only two persons A and B living in it, but do we also say that the house is the only one that A and B live in?

4.2.2 Graph Interpretation in Our System

In order to resolve the ambiguity and complexity in the definition of a graph, we use the second interpretation for a relation graph, i.e. pattern matching.

Mathematically, we define a graph as follows:

$\exists O_1,O_2,\ldots O_k \in \{$***Entities from the Database***$\}$, $\forall$ $R_m \in \{$***Edges in the graph***$\}$
$(R_m(O_a,O_b) \rightarrow O_a$ relates to O_b by relation R_m, x != y $\rightarrow O_x$,!= $O_y)$

And the following graph is defined as:

$\exists$ A,B,C,D $\in \{$***Person and Company entities***$\}$ (R1(A,B), R2(A,C), R3(C,D) , A!=B, A!=C, A!=D, B!=C, B!=D, C!=D)

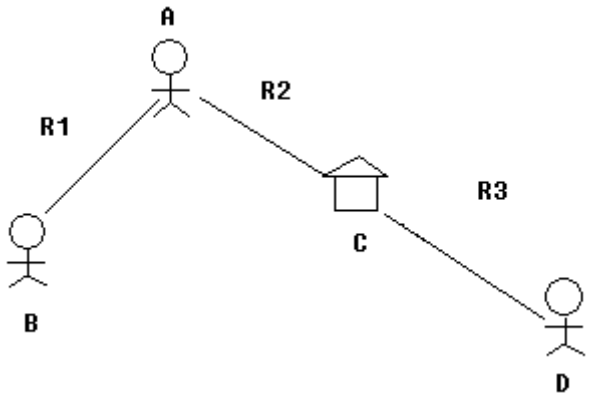


Figure 7. A relation graph with 3 persons and 1 company

It is not concerned whether or not any of the entities is related to some other entities that are not included in this graph. (e.g. house C may or may not relate to a third person)

5 Data Mining Process

The database we got from the bank was stored in an XML file; therefore, we had to parse the XML file into MySQL database. With the XML parser, users will be able to upload their own database and do data mining on their own database using this system. However, this feature is not implemented in our system currently. The basic working process of our system is as follows.

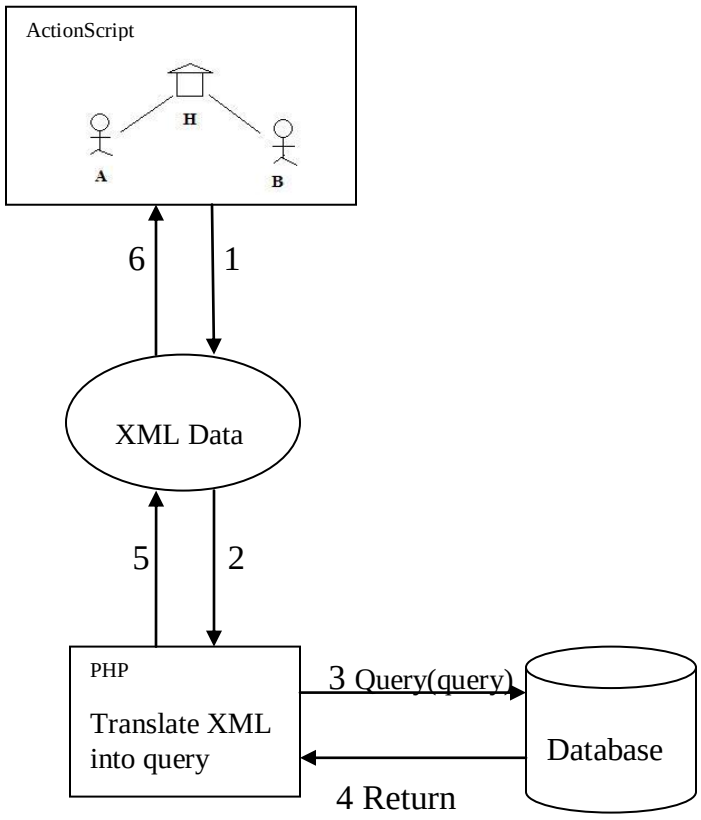


Figure 8. The workflow of the system (numbers indicate process steps)

Firstly, a user creates a relation graph in the frontend by dragging the icons and drawing lines between the icons to indicate relations. Each icon represents a person, or a company, or whatever entity we have in the database. Secondly, the user will define the relations and put constraints on the entities. After the graph is completed, all the user-defined information will be encapsulated in XML data and passed to the backend. Finally, the backend runs the data mining algorithm and returns the results in XML format to the frontend.

5.1 Data Passing Between Frontend and Backend

The frontend communicates with backend using XML data. When the frontend passes a query to the backend, the query is formed in XML format. The XML contains all the information of the entities and relations in the graph. The backend extracts all the information and forms corresponding sql queries. After processing the queries, the results are returned to the frontend in XML format as well. The backend is running on PHP.

When a connection is established between the frontend and backend, the first thing that happens is that the backend will send initialization information to the frontend. The initialization information includes the names of all the entities' attributes in the database (e.g. "first name", "last name", "profession", "title" etc. for person entities, and "name", "location", "industry" etc. for company entities), the relations that exist in the database (e.g. "friends", "owner", "partners" …), and the values of some entities' attributes that have a limit number of options to choose from (e.g. person's professions, countries, etc.).

The format of the XML data depends on the method that we use to do the query. Two approaches will be discussed to solve the problem, and for each of the methods, the XML data is formed differently. For the tree structure approach, the XML data is formed in a tree structure as well, with the root as the entity that users search for. For the iterative approach, the XML data is formed in parallel structure. The following figure shows the XML data structure for both the tree structure approach and iterative approach. The tree structure is more compact, it does not have any duplicate information, each entity and relation appears exactly once in the data. In the parallel structure, information is passed in entity pairs that correspond to a relation; therefore, information for each entity is duplicated as many times as the number of relations it is involved in.

Figure 9 shows how the queries are structured in the XML data when passed from the frontend to the backend.

Tree Structure

```
<person>
        <this> A </this>
        <person>
                <relation> R1 </relation>
                <this> B </this>
        </person>
        <company>
                <relation> R2 </relation>
                <this> C </this>
                <person>
                        <relation> R3 </relation>
                        <this> D </this>
                </person>
        </company>
</person>
```

Parallel Structure

```
<query>
        <relation>
                <type> R1 </type>
                <person> A </person>
                <person> B </person>
        </relation>
        <relation>
                <type> R2 </type>
                <person> A </person>
                <company> C </company>
        </relation>
        <relation>
                <type> R3 </type>
                <company> C </company>
                <person> D </person>
        </relation>
</query>
```

Figure 9. XML data structure for tree structure approach and iterative approach

5.2 Types of Queries

There are three types of queries, including single entity queries, one-to-one relation queries, and complex relation queries which involve multiple relations. The first two types of queries are basic queries and are quite intuitive to implement.

When querying for single entities, the frontend passes user-specified attributes of a person or a company to the backend, and then the backend does a simple query for

all the persons or companies that match the searching criteria. When it comes to one-to-one relation queries, users specify searching criteria for both entities of a relation. The backend will do two single entity searches, and then searches for pairs that match the relation. These two types of queries are quite trivial and straightforward. However, complex relation graph queries may be more complicated to implement.

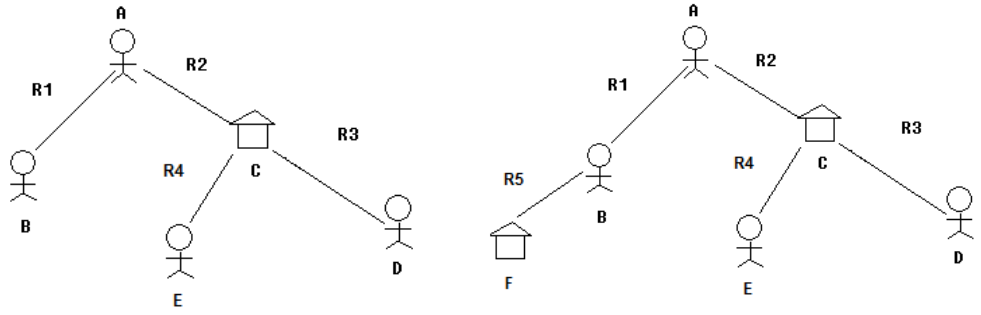
We implemented the searching algorithm in two ways. The first method uses a tree structure to model the relation graph. We search for the results in a bottom-up fashion. We use the entities at a lower level to narrow down the searching space of those at a higher level until we reach the root entity. This algorithm returns only the matching results of root entity. The other method is more intuitive. It iteratively queries for each relation in the graph, and then does some post-processing on the set of results. The processing is largely on joining the tables and remove illegal result entries. In both implementations, only IDs of the interested entities are returned instead of the complete set of information, because large numbers of result entries are expected. The complete information of a certain entities will only be queried and returned upon users' request. The tree structure method is more efficient; however it has certain limitations that make it disadvantageous to the second method. The second method involves large amount of data processing. Fortunately, PHP supports dynamic arrays as a built-in primitive data type, and it largely reduces the complexity in array processing. Instead of joining the query results in SQL, it would be much more convenient to first store the intermediate results in dynamic arrays in PHP, and then do the joining of the arrays.

5.3 Algorithms

The simplest implementation of a pattern search is to form a single query for the entire graph. For example, we can use the following query for the graph in Figure 7.

```
SELECT A.ID, B.ID, C.ID, E.ID
FROM person AS A, person AS B, company AS C, person AS D
WHERE A.$ = '$' AND A.$ = '$' …
        B.$ = '$' AND B.$ = '$' …
        C.$ = '$' AND C.$ = '$' …
        D.$ = '$' AND D.$ = '$' …
      AND (A.ID, B.ID) IN (SELECT PID1, PID2 FROM personperson WHERE
      Type='R1')
      AND (A.ID, C.ID) IN (SELECT PID, CID FROM personcompany WHERE
      Type='R2')
      AND (D.ID, C.ID) IN (SELECT PID, CID FROM personcompany WHERE
      Type='R3')
      AND A<>B AND A<>C AND A<>D AND B<>C AND B<>D AND C<>D
```

The '$'s in the query represent the attributes of the entities and the values of the attributes. This query of simple form returns us the results of (A, B, C, D). This query runs for around 5 seconds and returns 34 records. In order to study the efficiency of this algorithm, we increase the complexity of the graph pattern. Using this algorithm, we tried querying for the graphs in Figure 10.



(a) (b)

Figure 10. More complex patterns

Both graphs take around 5 seconds to finish. We assume there is still space for improvement on the efficiency by reducing the size of joining tables. In order to

reduce the size of tables for joining, we can iteratively or recursively reduce the size of the tables along the way when we join tables or use some divide and conquer methods. We tried two methods to solve this problem.

5.3.1 Tree Structure Approach

By using the tree structure approach, we need to assume that the users are interested in only one entity in the whole graph. Take the graph in Figure 7 for example, if the user is only interested in person A who is related to person B and company C which is related to person D, we can use the topological structure in Figure 7 for the tree structure. In other words, the entity that the user is interested in is set as the root.

We can use the DFS algorithm with recursive implementation to get the root entity. The algorithm is shown in the following pseudo code.

```
resolve(root)
        IF root has no child
                RETURN Query(root)
        ELSE
                query = Query(root)
                FOR each child c DO
                        childquery = resolve(c)
                        relationquery = relationQuery(Query(root), childquery)
                        query = query + "and ID in relationquery"
                ENDFOR
                RETURN query
        ENDIF
```

Figure 11. The algorithm of the tree structure approach

The method “relationQuery(Query(root), childquery)” returns the root entities that have a relation with the entities from childquery. This algorithm works by narrowing down the searching results from the leaves of the graph and goes up to the root that the users are interested in. For the structure in Figure 7, this algorithm will first resolve relation R1, and get a set of results of A which comply with relation R1, and then resolve relation R3 that gives a set of results for C, and then based on the set of C and R2, we get another set of A. We take the intersection of the two sets of A, and then it gives us the set of results for A that we are looking for. This algorithm searches for the matching pattern of the graph by narrowing down the searching space from the leaves to the root. All the leaves are dropped during the process. Therefore, users will only get the matching results for person A. This algorithm is relatively more efficient because all the information in the graph is effectively and never repeatedly used.

However, this algorithm works well only with tree structures. In a general case where there are cycles in the graph, this algorithm has to be improved or replaced. For a cyclic graph, we cannot use the same schema for data passing from frontend to backend, for one reason it is impossible to arrange the data in hierarchical order. Suppose we have a relation graph with a cycle, as shown in Figure 12. In this graph, there is a relation cycle among A, B and C.

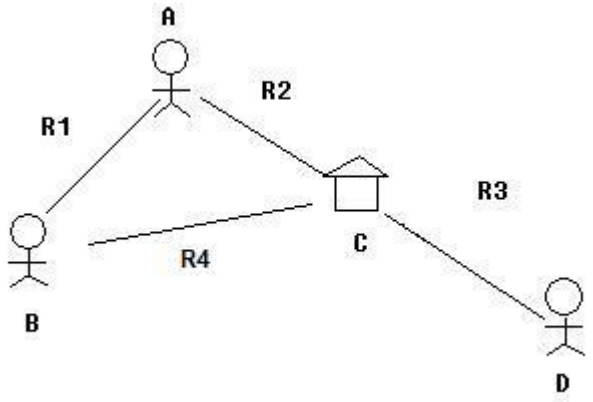


Figure 12. A relation graph with a cycle

If we want to use the tree structure method to resolve this pattern, there is one way we can modify this graph. We can always break a relation between two entities and create two extra entities that are exactly the same as the original two. By breaking the relation between B and C in Figure 12, we can reform this graph into a tree structure as Figure 13.

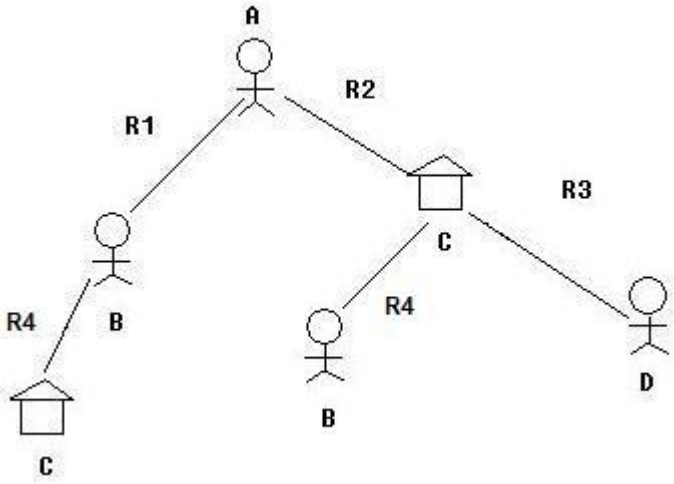


Figure 13. Reform a cyclic graph into a tree structure

Even after we have reformed the graph into a tree structure, the algorithm will not be the same as what we use for the graph in Figure 7, because with this tree structure we have to ensure that the 2 Bs and 2 Cs are the same entities respectively. Each of the entities in the graph is most likely not fully defined by

the users, so according to users' definition for company C there are probably large number of candidates to choose from. Whatever candidate we choose for one C, it has to be the same for the other. However, if we use the recursive DFS algorithm, the C on the bottom left will be searched first. Based on the results of C, we will accordingly get a set of B, and then A, etc. Later on, when we get to the C on the right, since company C is constrained with more relations here, the set of results for C we get here will be much smaller. A problem arises here that we need to revise the set of results for the previous C, and in turn B and A as well. The algorithm would be much too complicated if we want to resolve a cyclic graph using a tree structure. Therefore, we came up with another method to resolve a general graph with or without cycles.

5.3.2 Iterative Approach

The other approach to searching for a pattern is by iteratively searching for the one-to-one relations, and then joining them in a certain way. This is a divide-and-conquer approach. We break down a relational pattern into individual relations, and then join the result tables. This method gives us certain benefits over the single-query form of implementation. The single-query form of implementation is forming a single query out of the whole relational pattern. The tree structure approach is implemented in this single-query form. The iterative approach is much easier to implement than the single-query form of implementation. Besides the easiness in implementation, the iterative approach has another advantage. Since we treat a graph as a composite of one-to-one relations, the part of the graph which occurs at a high frequency can be cached and used for future querying. For example, in Figure 14, if the circled pattern of (A, B, C) is frequently used as part of a user-defined pattern, we can cache the results of (A, B, C) for this small pattern and use it when we see a pattern with this part in it.

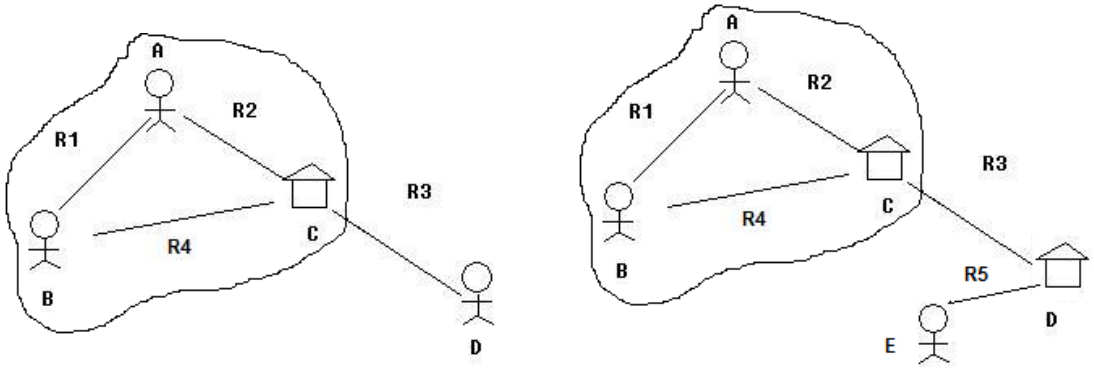


Figure 14. Cache a frequent pattern

However, in the single-query form of implementation, since we have to form a single query for the pattern, we cannot use results from frequent sub-patterns.

Joining is the more complex part in this algorithm. In this algorithm, there is no hierarchical structure for the graph. And the result we get covers all the entities instead of a particular one. Using the tree structure approach, we will get only the results of A for Figure 7, however the iterative approach will return us the results of (A, B, C, D).

Take the pattern in Figure 12 for example, the most intuitive way to query for this pattern is by iteratively querying for a pair of entities for each of the relations. As a result, we get four sets of 2-tuples. The 2-tuples are constrained by the definition of each entity and the type of the relations. The results are stored in dynamic arrays. PHP supports dynamic arrays, and dynamic arrays have the advantage of good locality of reference and data cache utilization, compactness and random access. These advantages make dynamic array an efficient tool to implement table joining. PHP also provides a wide range of array functions that make array manipulation much easier. We store the following three tables in a two dimensional dynamic arrays.

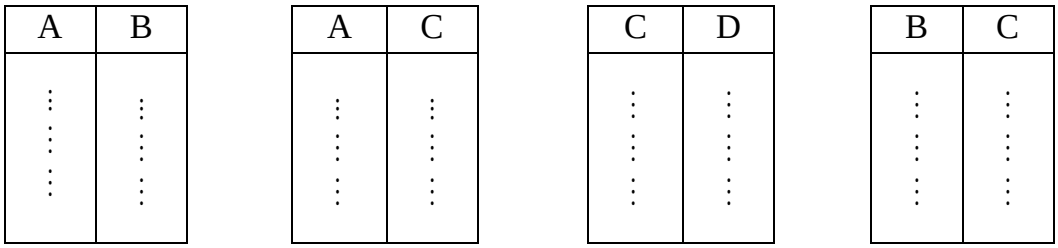


Figure 15. Individual tables for each relation query

Then we implement table joining in dynamic array. We first join the first two tables on entity A, which gives us a table of (A, B, C), and then we join table (A, B, C) with the third table on entity C, as a result we get (A, B, C, D), and then we join this result with the last table on B and C to get the final result (A, B, C, D). The table joining procedure is summarized in Figure 16.

```
Function join(table)
        result = table[0]
        FOR table[1] to table[n]
                FOR each key1 of Result
                        FOR each key2 of table[i]
                                IF key1 == key2
                                        sort( table[i] )
                                        sort( Result )
                                        result = mergejoin( table[i], result)
                                ENDIF
                        ENDFOR
                ENDFOR
        ENDFOR
        RETURN result


Function mergejoin(t1, t2)
        result = null
        idx1 = idx2 = 0
        WHILE idx1 < length(t1) and idx2 < length(t2)
                IF t1[idx1] == t2[idx2]
                        increment idx1 idx2 till t1[idx1'] > t1[idx1] and t2[idx2']>t2[idx2]
                        result = result + Cartesian product( t1[idx1..idx1'], t2[idx2..idx2'] )
                ELSE IF t1[idx1] > t2[idx2]
                        idx2 ++
                ELSE
```

```
                    idx1++
                ENDIF
        END WHILE
        RETURN result
```

Figure 16. Table joining algorithm for the iterative approach

However, the challenge is to ensure the identicalness of the entities in the graph. When we separately do the queries on each of the relations, we cannot ensure that all the entities in the graph have a unique identity. For example, a person P might satisfy the constraints for both B from A-B relation and D from C-D relation. After joining the results, we will get an entry with B and D both being person P. This is obviously undesirable. We have to eliminate the results containing duplicate entities. There are two ways to do this; we can either include the inequality constraints when doing the query or eliminate the results with duplicate entities after joining the results.

Imposing inequality constraints on the entities when doing the query makes the system not scale to the size of a problem. For example, after we get the results for A-B and A-C relations and we need to query for C-D relation, we have to add constraint "D not in SELECT B FROM A-B and D not in SELECT A FROM A-B". We have to ensure that every two of these entities are identical. When the number of entities in the graph grows, the number of inequality checks will increase accordingly. These checking operations are quite expensive, because each "not in" is accompanied by a "select".

Another approach to this problem is to eliminate the results with duplicate entities after joining all the 2-tuple results. In other words, we join all the results even if more than one of these entities share the same identity. For example, after joining all the four tables, we get a table consisting of entities A, B, C and D. Based on the

set of results, where duplicate identity of entities is possible, we can eliminate the entries with duplicate identities. In the results of the above example, it is possible that we get a result entry where ‘A’ and ‘D’ in fact refer to the same person, and then we can remove that entry. This part is very trivial, it simply checks all the records in the result table, and deletes those with duplicate identities.

This algorithm works for a graph of any pattern, because it does not depend on any kind of structures to resolve a pattern. For a single relation query, it takes less than 0.2 seconds. However, when the complexity of a graph increases, the whole process will normally take up to 4 seconds or even more. Figure 17 shows the performance of this algorithm with graphs of different complexities.

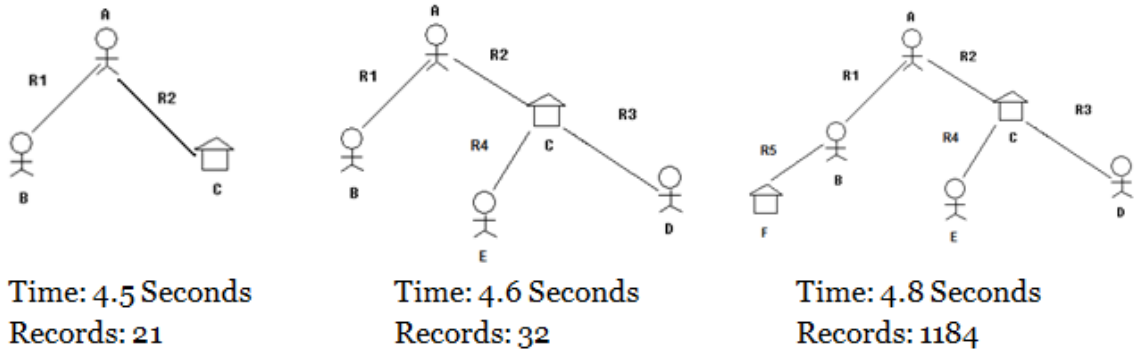


Figure 17. Performance of iterative approach

With the results shown above, we can see that this algorithm has very good scalability. And the efficiency is comparable to the simple implementation. However, using iterative approach gives us the opportunity of doing partial querying. This can be used as heuristics to improve the overall performance of the algorithm.

6 Conclusions

In this paper we have presented two algorithms for graph pattern recognition, the tree structure approach and the iterative approach. The tree structure approach does not successfully solve the problem with the limitation in the structure; however this technique and idea of using DFS traversing algorithm to query for patterns can be generalized in other implementations. The iterative approach can handle all kinds of relation patterns, because it does not reply on any structures other than one-to-one relations. For this reason, the iterative approach can be too expensive in graph data mining although it works better than the simplest and most naïve approach. It queries for each of the relations in an exhaustive manner. Therefore, in order to develop a more efficient algorithm for graph data mining, we can incorporate some heuristics to reduce the query operations. For example, if a sub-pattern of a graph appears to be showing up frequently, the system should remember the pattern and the search results, so that the results can be used right away when this sub-pattern is encountered. This is what we can improve in future work.

The iterative approach works perfectly despite the efficiency could be improved in certain ways. However, in real situations for a pattern searching system, there are more aspects we need to improve on. For example, users might not be fully aware of a specific pattern they want to search for or they do not have enough values for an accurate search. With an error input, users will probably end up with totally different searching results from what they expect. Therefore, the system should be able to return searching results that do not strictly adhere to users' definition of the pattern, but presents some potential to users' interest. The generation of the system-determined results is based on the likelihood between the result and users' intended query. We need to assign a weight value to each searching attribute of a query; such kind of system-determined results can be generated by discarding

some most irrelevant attributes in the queries, or even dropping some important attributes but keeping more less important attributes to compensate the loss. This could also be part of our future work.

Finally, as we have mentioned earlier, we could improve this system in the way that users are allowed to upload their own data file, and use our system to do data mining on it.

7 References

[1] Diane J. Cook, Lawrence B. Holder, Jeff Coble and Joseph Potts. (2005). Graph-based Mining of Complex Data. Advanced Methods for Knowledge Discovery from Complex Data, Springer, 2005, Part I, pp.75-94.

[2] Tao Jiang and Ah-Hwee Tan. (2005). Ontolopy-Assisted Mining of RDF Documents. Advanced Methods for Knowledge Discovery from Complex Data, Springer, 2005, Part II, pp.231-252.

[3] Carlo Batini, Monica Scannapiece. (2006). Data Quality (Concepts, Methodologies and Techniques). First Edition, Springer, 2006.

[4] Vicenc Torra. (2003). Trends in Information fusion in Data Mining. Information Fusion in Data Mining, Springer, 2003, pp. 1-6.

[5] Mohammed J. Zaki. (2005). TreeMiner: An Efficient Algorithm for Mining Embedded Ordered Frequent Trees. Advanced Methods for Knowledge Discovery from Complex Data, Springer, 2005, Part I, pp.123-152.

[6] Ning Zhong. (2003). Mining Interesting Patterns in Multiple Data Sources. Information Fusion in Data Mining, Springer, 2003, part II, pp. 61-78.