University of Zurich
Department of Informatics

The extended GraphSlider-Framework

A temporal Approach to internal Fraud Detection in Transaction Databases

Diploma Thesis, December, 19, 2008

Michael Meier
of Flaach ZH, Switzerland

Student-ID: 03-715-414
meier.spruso@bluewin.ch

Advisor: **Jonas Luell**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/ddis

Dynamic and Distributed Information Systems

Acknowledgements

First, I would like to thank my supervising assistant Jonas Luell for his patience and his confidence in me. With his suggestions and remarks, he helped me to take this thesis to the point it is now. I would also like to thank Prof. Abraham Bernstein, who gave me the opportunity to write this thesis and make this interesting experience

Abstract

This diploma thesis addresses the automatic recognition of fraudulent activities in the transaction databases of a bank. Therefore, the already existing fraud detection program GraphSlider gets extended with new functions. The first function addresses the recognition of fraud, based on temporal data in the database, because this data is almost always available but very seldom used for fraud detection. The second new function addresses the recognition of internal fraud on the employee level. In order to achieve this, our approach tries to track the fraudulent actions back to the single employee. At the end, the new approaches are tested with synthetic data if they are capable and if they have a good performance.

Zusammenfassung

Diese Diplomarbeit befasst sich mit der automatischen Erkennung von betrügerischen Aktivitäten in den Transaktionsdatenbanken einer Bank. Zu diesem Zweck wird das bestehende Betrugserkennungsprogramm GraphSlider mit neuen Funktionen erweitert. Die eine Funktion beschäftigt sich mit der Erkennung von Betrug basierend auf temporalen Daten in der Datenbank, da diese bisher selten genutzt werden, obwohl sie fast immer verfügbar sind. Die zweite neue Funktion beschäftigt sich mit der Erkennung von internen Betrugsversuchen auf der Ebene der Angestellten. Zu diesem Zweck wird versucht, die betrügerischen Handlungen zu den einzelnen Mitarbeitern zurückzuverfolgen. Die neuen Ansätze werden zum Schluss auf synthetischen Daten auf ihre Tauglichkeit und ihre Performance hin überprüft.

Table of Contents

1 Introduction

In our modern society, almost everything is connected in some or another way, from mobile phones to PDAs, from computers to kitchen equipment. And the more things get connected, the more activities are handled through these connections.

This not only applies for the private use, also business transactions are handled this way. But this connectivity not only has vantages. The more things get connected, the more vulnerable they are for illegal actions, because of the lack of control.

This especially applies for electronic transactions and bank transactions in particular. The more bank transactions are handled through electronic mediums, the more they are vulnerable to so called fraudulent actions. These actions can come from the outside of the bank, but they also sometimes come from the inside. This is the reason why a lot of work is done in the sector of fraud detection and fraud prevention. The later tries to identify fraudulent behaviour and to adjust the business model or they way, transactions are handled, to give fraudsters no opportunity to commit their crimes. But as every prevention mechanism, this does not work with a hundred percent reliability. If the fraud prevention has failed, fraud detection comes into account.

There are many papers and studies on how to detect possibly fraudulent behaviour and transactions. These detection mechanisms are not only applied in the banking sector, but in the telecommunication or insurance sector, too. Most of these techniques focus on the detection of fraud from the outside. But there is also a not so small percentage of fraud coming from the inside, which these techniques do not cover.

1.1 Goal of this Thesis

With this thesis, we try to approach this problem. The GraphSlider framework is a fraud detection system, tuned to find fraudulent chains in a transaction database. We took this approach and modified it in two ways. First, it not only should search for external imposed fraud, it should also look for internal fraud, caused by the employees of the bank. Therefore the GraphSlider was extended to take the registrars of the transactions into account and trace fraudulent behaviour back to them.

The second modification was to take temporal data into account. Normally, temporal data of the transactions is available but not used in fraud detection. We tried to set up a new approach which takes this unused data into account together with the already used data, like the amounts transferred or the involved bank accounts.

We modified the GraphSlider to comply with these new requirements. Because we mainly wanted to show, that there are other possibilities of detecting fraudulent transactions, the GraphSlider is not optimized to be very high-performance or memory-saving. It has to be viewed as a proof of concept, and we did in fact prove that this new approach can work.

1.2 Structure

This thesis follows the structure of a design science thesis proposed by [Bernstein 2005]. First, the motivation behind the actual thesis will be set. A description of the actual work follows. The evaluation will show the actual results, followed by a discussion of them. At the end, some final remarks will be added.

Chapter 2 contains the definition of the problem and the motivation why this problem should be solved and why it is a worthwhile topic.

Chapter 3 contains the description of the original GraphSlider. It shows the concept behind its original function and it also shows the concepts that finally lead to the implementation of the extended GraphSlider.

Chapter 4 contains the evaluation of our new approach. It covers the measurement of computation time, memory usage and the accuracy of the new approach.

Chapter 5 discusses the results of the evaluation. It describes the advantages and the disadvantage of the new method. It also covers the drawbacks that were discovered during the evaluation.

Chapter 6 contains the conclusion of this thesis. After comparing the results with the initial goals, some suggestions for future works are given. Finally a personal résumé is drawn.

2 Motivation

In this section, the motivation behind the whole thesis will be explained. Therefore, the terms of fraud and internal fraud are first explained and their impact on the banking business is identified. Then, we will have a closer look at the previous work done in this sector. What are the different approaches good for, where do they lack something? We will also give a short insight into the other fields of research, which are important for this diploma thesis, like pattern matching and sequential pattern mining. Finally, an approach to a possible solution will be drawn. It will state what the requirements are for a solution and what it should and what it shouldn't do.

2.1 Fraud, internal fraud and fraud detection

To develop a method against fraud, we first have to clarify, what fraud is and what it means for the business.

2.1.1 Definition of fraud

In [Phua et al. 2005] it is stated, that fraud is the "abuse of a profit organisation's system without that abuse leading necessarily to legal consequences". In [Balton et al. 2002], fraud is declared as "the use of false representations to gain an unjust advantage". So it can be said that fraud is the usage of a system in a way out of its intended rules and purpose, but close enough to legal use that the chance is high to go undetected and therefore not prosecuted.

2.1.2 Internal fraud

A lot of fraud is committed by external entities such as criminal organizations or single fraudsters. But there is also a percentage of internal fraud. This means, the fraudulent behaviour is shown by someone who has direct access to the system or who is in the system itself. Internal fraud is defined in [Phua et al. 2005] as "fraudulent financial reporting by the management and abnormal retail transactions done by employees".
One problem of fraud is that, as a company, you can never be sure who behaves fraudulent. There were surveys which tried to generate a statistical profile about the average fraudster. For internal fraudsters, a Federal Bureau of Investigation's study states, that the average female fraudster is 19 to 30 years old, has an employment in the low or mid-level and sees her fraudulent behaviour as "borrowing" money. The male fraudster is 25 to 30 years old, is an executive or administrator and the financial loss he causes the company is often ten times

higher than the one of female fraudsters[1]. Unfortunately, such surveys are not very significant for regions and companies other than the one surveyed. So it is not that easy to generate a general profile for potential fraudsters, although there exist some indicators that can point to suspicious behaviour.

2.1.3 The problem of fraudulent behaviour

Fraud is becoming a serious problem in modern times, because a lot of business transactions are not handled manually anymore. The more a business switches to wired transfers, the more it is exposed to potential fraud. And the development of new technologies has widened the area for potential fraud, too. Although this applies for many fields of business, fraud is a special problem in the credit-card, e-commerce, insurance, and retail and communications business [Phua et al. 2005]. Because not only the transfers of credit-card data are wired nowadays, but almost all financial transactions are, the whole banking business is more than ever a target of fraudulent behaviour.

Fraud is becoming business relevant or even critical, because if only a small percentage of the transactions committed each day is fraudulent, this can have a big impact not only on the financial side, but also on the reputational side. A bank not capable of detecting fraudulent behaviour in its transactions can be blamed to be not careful enough. And a loss of customer trust can have an impact much bigger than only the financial loss caused by the fraudulent transactions committed.

2.1.4 Fraud prevention and fraud detection

After Balton and Hund [Balton et al. 2002], it is absolutely necessary to try to prevent fraud as good as possible. This can be achieved though several steps of monitoring and designing the business system in a way that makes fraudulent behaviour nearly impossible. Unfortunately, this rule is not always followed and fraudsters are often as smart as the developers of fraud preventions mechanisms. Therefore it is absolutely necessary to also implement fraud detection techniques [Balton et al. 2002]. These techniques have to be applied continually, because it is not possible to determine, if and when the fraud prevention mechanisms have failed.

Earlier, fraud detection was handled manually. It included manual screening and checking of the various transactions. It could only be done to the most suspicious transactions and was not only very time consuming it was not fail proof either. Therefore, automated fraud detection

[1] Internal Fraud - http://www.bankersonline.com/articles/bhv03n01/bhv03n01a2.html

mechanisms had to be introduced to guarantee a continuous monitoring and to relief the workers.

One of the biggest problems in automated fraud detection is the fact that one can never be completely sure, if a transaction is really fraudulent or if it is only by coincidence that the fraud detecting mechanism has marked this certain transaction as fraudulent. Transactional fraud is normally added to regular transactions to prevent it from discovery [Phua et al. 2005]. Therefore it is nearly impossible to handle the fraud detection completely without human interaction. Today's fraud detection mechanisms can make a good pre-sort of suspicious transactions, but the results normally have to be reviewed by a specialist nonetheless. This means fraud detection mechanisms can only give a hint, which transactions have to be investigated further. Although these pre-sort mechanisms can achieve a 99% detection rate, it can be very time consuming to find the real fraudulent transactions and to relieve the "innocent" transactions. Therefore it is very important to have rules on how to weight false positives and false negatives. False negatives are often more costly than false positives [Phua et al. 2005], because an undetected fraudulent transaction can cause much more loss than a wrongly accused legitimate transaction. This is because the money of the fraudulent transaction is really lost, but it is only a matter of personal effort to double-check the non-fraudulent transaction. Therefore monetary factors often are introduced to measure the performance of fraud detection mechanisms. Another potential weak point of the automated fraud detection mechanisms is that they are not capable of detecting fraudulent behaviour that they are not designed for. If a fraudster finds a way to circumvent the detection system, this fraud can go undiscovered, even for years [Balton et al. 2002]. As long as the mechanisms do not adapt to the new situation, fraud will most certainly go undetected [Tuyls et al. 2000]. So it is a continuing race of the two factions to gain the advantage over the other. The trend in newer works is therefore to create mechanisms than can adapt or be adjusted to new situations easily. This is not only because the fraudsters tend to adapt to the new detection mechanisms, but because the legitimate behaviour tends to shift over time, too. As a remark it is also very important for the developer of anti-fraud mechanisms to maintain the old mechanisms to prevent fraudsters to switch back or to deny new, inexperienced fraudsters the access to old flaws.

2.1.5 Problem of cooperation in fraud detection

Unfortunately, most of the work done in fraud prevention and fraud detection is not open to the public. This has several reasons. One is, that if the new findings would be accessible by anyone, potential fraudsters can inform themselves about the newest development and adapt

even faster as they do at the moment. Another important reason, why most of the research work is not published is that they often use sensitive data from the particular company. For banks these are real transactional data which are under the protection of the banking secrecy.
Despite the lack of real data there is a good common sense, which attributes are important for the development of a good fraud detection mechanism. These attributes are often dates, amounts of transferred financial values, locations involved in the transaction, the transactional history, the payment history and other information, like the age of the involved accounts [Phua et al. 2005].
To compensate these non disclosure agreements and the lack of exchange in this sector, most studies are using artificially generated data. As with the experience of years of work, this data is nearly as realistic as real data would be and can therefore easily be used for further development and investigations. There are studies [Barse et al. 2003] with the topic of artificially generated data compared to real data and they state, that it is a legitimate approach to generate synthetic data to train and implement new fraud detection mechanisms although, the results may vary when the simulated data is applied to real data.

2.2 Earlier work in fraud detection

As stated in the last section, fraud is a problem in many different businesses today. Quite a lot of work has been done to create efficient fraud detecting mechanisms. In this section, a few of them will be introduced.

2.2.1 Fraud detection survey

In [Phua et al. 2005] Phua and his co-writers were trying to give an overview about the papers written by the year 2005. They surveyed 51 unique and published papers in the different areas of fraud detection. As this thesis is about fraud detection in the financial sector, it is interesting, that the papers with the topic of transactional fraud have the highest count. But this thesis is also about internal fraud and remarkably, there is not that much work done. In fact, there are only a few papers about fraud at the management level and only one paper about fraud at the employee level.
Phua et al. state in their survey that the methods of modern fraud detection can be roughly categorized into four different categories

2.2.1.1 Supervised approachs with labelled data

The first category is the supervised approach on labelled data. This seems to be one of the most used approaches in this field of science. The goal of it is to get some data from which is known which part is fraudulent and which is regular. Then, various algorithms are used, like neural networks, decision trees, case-based reasoning, regression or support vector machines, to give a prediction model. As soon as this model is generated and applied to new data, it should be able to determine, which part of the new data is fraudulent. The problem with this class is that there are some restrictions in the different algorithms. Some can only process non-discrete attributes, some need every attribute filled and some are too slow when applied to novel data. This category is the typical data mining approach.

2.2.1.2 Hybrid approaches with labelled data

In the second category there are the hybrid approaches with labelled data. They are quite similar to the non-hybrid ones with the small difference, that more than one algorithm is used to generate the rule set. Normally, there are two or more of the data mining algorithms used in a sequential way to generate the rule set. This applies for the supervised hybrids. There is also a small part of unsupervised hybrids. This method is used especially in the telecommunications sector. But two studies state that the supervised approaches excel the unsupervised ones.

2.2.1.3 Semi-supervised approaches with non-fraudulent data

The third category consists of the semi-supervised approaches with only non-fraudulent data. The approach by Kim et al. [Kim et al. 2002] discussed later is of this category. They trained their T-detectors only with data known to be non-fraudulent. In other approaches in this category, it was tried to identify fraudulent behaviour by supervising the customers known to be non-fraudulent and watch for deviances. These deviances would fire an alert and indicate a potential fraudulent behaviour. This is achievable especially in the credit card or communications sector, where the customer tie is quite strong and long term data is available. If for example a customer, who tented to make phone calls only in his vicinity and only during daytime, starts to make calls to foreign countries and in the night, his account is very likely to be taken over by a fraudster. With these observances of deviations from standard behaviour, rules are generated to make predictions about fraudulent behaviour.

2.2.1.4 Unsupervised approachs with unlabelled data

The fourth and last category is the one with unsupervised approaches and unlabeled data. Link analysis and graph mining are often used here. They are researched especially for anti-terrorism and similar security areas. But as [Phua et al. 2005] states, the possibilities of graph mining seem to be ignored by most of the fraud detection community, as only very few papers use a visual fraud detection system for example. As it will be shown later, this approach should be pursued.

2.2.2 Biological approach

In their paper, Kim, Ong and Overill [Kim et al. 2002] try to counter internal fraud by a system called CIFD (Computer Immune System for Fraud Detection). The paper is set in the retail business, where a lot of the transactions are handled electronically. Therefore it is also relevant for the banking business, where most of the transactions are handled electronically, too. The special thing in this paper is the fact that they try to approach the problem in a biological way. With negative and positive selection, they try to train an artificial network to detect anomalies in retail transactions. As they state, this is not very easy. They try to take T-cells as the model for their approach. Biological T-cells are selected via positive and negative selection, but as a drawback, it is not yet certain, how this functions in detail. Nevertheless they use these two mechanisms to train so called T-detectors. With positive selection, they select immature T-detectors that generate the contradiction of the consequences of the two strong association rules ("if A than C") that they where combined from. If the confidence of this new rule is above a certain threshold, the rule is selected, otherwise it is eliminated. The negative selection is applied thereafter by generating mature T-detectors which have a certain self tolerance, i.e. don't detect themselves.

2.2.3 Criticism and problems of earlier work

As an interesting side note, it should be stated that the choice of the method in the real world is often not really dependent on the technical restrains implied by the data or the lack thereof. It is more often dependent on the practical use and the commitment of the management towards the use of fraud detection mechanisms. As noted before, fraud can also be caused at the management level. So it seems to exist some sort of tradeoff between guarding the firm against losses caused by fraudulent behaviour and the feeling of being monitored by the firms own anti-fraud system.

Besides the fact, that the development of fraud detection systems seems to be hindered by firm politics, [Phua et al. 2005] states that there are other criticisms. As declared in the previous section, most of the known fraud detection and fraud prevention systems published are only based on synthetically generated data, rather than on real data (although studies seem to prove that they are similar). This could lead to the claim, that these methods only work with synthetic data and that they would never be adaptable to real world applications. This seems to be true to a certain extend, because only a few of the proposed methods were actually implemented in a productive system. This could have changed in the meantime, as the survey of Phua et al. is from 2005. But it is nearly irreproducible because if some of the methods were actually implemented, it will likely not be known in public, because of the security issues stated earlier.

One of the strongest drawbacks of the previous works is that nearly any of them incorporates temporal data for fraud detection. As it is known, especially in transactional data, temporal information is almost always available. As stated in one of the telecommunication papers, this information can, together with other information, give a good hint for fraudulent or non-fraudulent behaviour. Another drawback is the fact, that spatial information is not used in any of the surveyed papers. This may not seems as important for the fraud detection problem, but like temporal information, it can give a good hint, especially in bank transactional data. For example a lot of transactions from one account in Boston to an account on the Cayman Islands can be suspicious (even if this example is a bit of a cliché).

The last critique on most of the methods is about their complexity. In the sense of science it is very nice to have some new and complex methods. With them, one can show that there are many possibilities to solve the problem of fraud detection. One can also show that (with synthetic data) very high rates of detection are achievable. But on the other hand, the more complex the methods, the more difficult they are to handle. As stated before, fraud detection is a time critical matter, especially when it comes to adapting to new fraudulent techniques or to a shift in regular or legal behaviour. This is because as long as the detecting mechanism did not adopt, the fraudulent transactions are processes but not recognised. The second problem with complex techniques is that they can indeed be more accurate, but often are slower than the less complex ones. Here the potential loss caused by fraud not detected because of the lack of accuracy has to be weighted against the potential loss caused by fraud not detected because of the mechanism taking to long to detect. A last drawback of the complex mechanisms is the fact, that they tend not to scale as well with high amounts of data to be processed. Especially in bank transactional fraud, where more than a few millions of transactions have to be handled each day, scalability can be crucial.

2.2.4 Pattern Matching

In the previous section, we got to know the four different categories of modern fraud detection. In the fourth category, there are the so called pattern mining algorithms. In the contrary to these data mining algorithms, pattern matching's main goal is to check for the existence of predefined patterns. In this section, we would like to look into the previous work done in this field of research and give a short overview about this important technique as it can be used to identify graphs of fraudulent transactions produced by the GraphSlider.

Pattern matching is a common technique in many areas of computer science. Its most well known application is in the field of expert systems [Held et al. 1987]. But graphical pattern matching is also used in the field of anti money laundering. The question is why are patterns that important? [Kuklas et al. 2005] indicates one possible answer. They state that patterns and motifs are very important when it comes to finding, uncovering and analyzing important properties of a graph.

2.2.4.1 Definition

Before the term pattern can be declared, first the definition of a graph has to be given. Formally, a graph consists of a set of nodes *(N)* and a set of edges *(E)*. Combined, they represent the graph *(N, E) = G*. The term of a sub-graph, which will also be used shortly, is defined as the following. *N'* is defined as a sub-set of *N*, *E'* is defined as a sub-set of *E*. Combined, they represent the sub-graph *S = (N', E')* of *G*. It depends on the algorithm, if the edges are directed or not [Kuklas et al. 2005, Rome 2002].

In graph pattern matching, different variables (this depends on the field of research in which the pattern matching is used in) are mapped to these nodes and edges. As we will see later in this thesis, our program will make use of this to represent the different bank financial transactions. An interesting side node is that [Varró et al. 2006] states, that there are some pattern matching algorithms that only store the nodes and not the edges.

One problem is, that the term "pattern" is highly discussed among researchers and that there is no real consensus about it [Rode 2005]. One of the more accepted definitions is the one of [Kuklas et al. 2005]. "Patterns are labelled, directed graphs". According to this, graph pattern matching is simply defined as "the problem of finding a homomorphic (of isomorphic) image of a given graph, called the pattern in another graph, called the target" [Valentine et al. 1997]. They state that this "is also known as the sub-graph homomorphism (or sub-graph isomorphism) problem". So to do graph pattern matching, two specific graphs are needed. One has to be well known and defined as the pattern to look for. The other graph, most of the time, is unknown. So normally it is even not known, if they contain the pattern graph at all.

2.2.4.2 Problems

One problem with graph pattern matching is that it is not that simple to find a given sub-graph in a much bigger target graph. As [Valentine et al. 1997] states, graph pattern matching is a computational complex problem. As to be exact, the finding of a sub-graph in a set of graphs is known to be NP complete after [Giugno et al. 2002]. It is known, that an NP complete problem is not easy to solve. The major drawback is that these problems are very complex and normally can not be handled in an efficient way. On top of that, there is the problem that most of the pattern matching algorithms are not very efficient, aside the NP complete problem. Most of them don't work incrementally and therefore use a lot of time to find the same sub-patterns over and over again. Varró et al. gave a good introduction on how to circumvent this problem [Varró et al. 2006].

One way to simplify the inefficiency problem is the fact, that simple patterns can be combined to complex ones. This makes the finding of patterns a bit easier, but it still remains a very complex problem. Another way to circumvent the complexity problem is to source it out to the underlying database. The problem with this approach is, that most of the relational databases and query languages only provide rudimentary pattern matching facilities [Held et al. 1987] and therefore are not sufficient for complex patterns.

2.2.4.3 Solutions

Because the problem of complexity can not be circumvented that easily, a lot of research has been done to find another solution. The reason why most of the optimization efforts come at such a late point in the whole discussion, although the problem of NP complete algorithms is known long in advance, is the fact that there has to be a well known and researched matching algorithm, before optimization can take place. As [Valentine et al. 1997] states, most of the effort has been done to find an efficient way to do graph pattern matching or to find new, useful heuristics that don't have the problem of being NP complete. But this is in fact not really necessary. In most of the cases, exact pattern matching is far too accurate and therefore too complex. Despite of that, one can never be sure that the found pattern is really a match or a false positive. Therefore, some of the newer algorithms turn down accuracy on purpose. They are much faster and if the accuracy is not turned down too much, the result does not significantly differ.

As we have seen in this section, pattern matching and especially graph pattern matching is not a simple problem. But the discovered methods in this field of research can also be applied to our problem of fraud detection. Most of the transactions in wired businesses can be described as graphs and the fraudulent behaviour can be described as known sub-graphs or –patterns. With graph pattern matching, we are able to track these fraudulent transactions and mark them for further investigation and processing.

2.2.5 Sequential Pattern Mining

As stated before, most of the known fraud detection techniques don’t use temporal data. This seems to be a bit short-sighted, because temporal data is one of the attributes, almost every transaction includes. Especially in bank financial transactions, this information is almost always available and should therefore be used to support the known fraud detection techniques or even to develop a new one, based on temporal data. To show that temporal information is very useful, this section will give a short overview about other fields of research, where temporal and sequential mining of patterns and data is used.

2.2.5.1 Earlier work

As [Pei et al. 2002] states, there are three different classes of sequential pattern mining algorithms:
Apriori-based with a horizontal formatting method
Apriori-based with a vertical formatting method
Projection-based pattern growth method
The earlier work in this field of science came from Agrawal and Srikant [Agrawal et al. 1995, Srikant et al. 1996]. They tried to find a method to discover sequential patterns in a database with different customer transactions. Their goal was to be able to make predictions about the future purchases of a customer e.g. customer A has purchased good D and than good F. How likely is it that when customer B purchases good D he will purchase good F in the future.
They proposed in [Srikant et al. 1996] that their finding in [Agrawal et al. 1995] was too simple and discovered that users often want to define the time-gap between adjacent elements of a sequence of interest. This means a customer who purchases good D and then waits two years to purchase good F will likely not be important for the predictions and the user of the system should therefore be able to define this sequence as not being relevant. As we will see later, our problem is similar, but we don’t want to define a minimal /maximal time-gap of elements to be relevant. We want to define a fixed time-gap between a sequence of elements, where elements not matching this sequence will be marked as potential fraud.

2.2.5.2 Problems

As [Mannila et al. 1997] states, one of the major problems in sequential pattern mining is to discover frequent episodes. Episodes are elements that occur relatively close to each other in a given partial order. One of the goals of sequential pattern mining is then to find the occurrence of such episodes. As soon as they are known, they can be used to make predictions about the next occurring of these elements. As stated above, if good D is always followed by

good F, you can predict that after an occurrence of good D, F will follow shortly. And this can not only be applied to goods. This can also be applied to other patterns, e.g. financial transactions. If one of these patterns is known to be fraudulent, one can eventually make a prediction about a future fraudulent behaviour, so one can use this as a fraud prevention technique.
Sequential patter mining algorithms on an apriori base are stated to have one major drawback, compared to pattern-growth based algorithms. They are slow i.e. use a lot of execution time. Bettini et al. showed in their survey [Bettini et al. 1998], that with optimization it is possible to achieve similar execution times with apriori based algorithms. This means, the drawback is in fact none.

2.2.6 Conclusion

The past sections should have given some small insights into different methods and fields of research, linked to fraud detection. Some of them have a strong link, some of them were never really used, although they could provide very interesting and useful information to help generate better fraud detection and fraud prevention mechanisms. In the next section we will look into how this can be useful for our thesis.

2.3 Approach to a possible solution

After the theoretical base set in the last sections, we will now go on to the main topic of this thesis. In this section, the problem to be solved will be described. Then, some possible approaches will be outlined and finally, we will discuss what a possible solution should and what it shouldn’t do.

2.3.1 Definition of the problem

The main goal was to find a solution for the fraud detection problem in the banking sector. Not only that there are many possibilities for fraudulent behaviour like smurfing or money laundering. There are also a lot of transactions per day. One can visualize them and try to make predictions or find suspicious patterns manually. But it would take a lot of manpower and would not be very failsafe. Therefore it was necessary, to introduce a new system for fraud detection, which could do an automatic pre-selection of suspicious transactions. There was a small system that could do this, called GraphSlider. One of the Problems with this new system was its lack of optimization. So the first goal of this thesis was to find a way to optimize the GraphSlider in both, memory usage and computation time. Because of the extended graphical capabilities – it drew every transaction as edge in a graph and each involved account as a node – it took far more time for the computation than there was really necessary.

After a few weeks, a new problem was discovered. Although the GraphSlider seemed to do well, there were already systems that could do the same. But one other problem of the banking sector was not covered by most of the solutions to date. As stated in the survey of Phua et al. [Phua et al. 2005] there was only one solution for the detection of internal fraud on the employee level. So we decided to shift the main goal of the thesis to implement a new version of the GraphSlider (hereafter called extended GraphSlider) to account this new problem.

2.3.2 Approaches to the solution

The approach to the first problem was to discover, where the extended computational time comes from and if there are possible memoryleaks. So it was necessary to go through the whole program, step by step, to find potential bottlenecks in the computation. A few were discovered but the goal was finally abandoned and the focus was shifted to the other problem.

The approach to the second problem was a different one. First, it had to be decided, how internal fraud should be accounted for. What were the attributes available, what could be done to indicate internal fraud? Luckily there was an attribute that described the responsible registrator for every transaction. This could be used in the extended GraphSlider to assign the transactions with the proper registrator and to monitor each of his actions individually.

Then, a way to indicate possibly fraudulent behaviour had to be found. As seen in the previous sections, there are many possible ways to do that. From pattern matching algorithms to data mining approaches to temporal or sequential pattern mining. There where many possibilities. As encountered in the literature research, there where only a few papers that used a temporal approach. But it seemed, as if there was a lot of information in the temporal data stored with each transaction. So it was self-evident to try a new approach and use this temporal data along with other information stored with the corresponding transactions to find possibly fraudulent behaviour.

The next step in our approach was to modify the GraphSlider to look for internal fraud. Therefore two new methods to handle different aspects of temporal internal fraud had to be introduced and developed. The exact functionality will be described in appendix A

2.3.3 Requirements of the solution

To give the whole process of development a frame, it had to be decided, what the newly extended GraphSlider had to do and what functions would not be necessary.

One of the main goals was obviously the detection of (possible) fraud. So the program should be able to find some predefined temporal patterns that would indicate possible fraud. Therefore two different algorithms should be applied. One that should be able to indicate smurfing, i.e. a lot of small transactions in a short period of time to hide the fact that they combined are a much bigger transaction that would have been suspicious. The other should check all transactions if there are irregular ones between two accounts that normally only have regular transactions (e.g. on a monthly base). Both of these algorithms should be adjustable in different ways via thresholds, variances in both, temporal and monetary data and weightings of the different incidents.

The program should also be able to indicate suspicious employees, or registrators, that may have committed a fraudulent action with their user-ID. This should be achieved by giving each registrator an account-score, calculated from the different potential fraudulent actions done in his account. The higher the score, the more suspicious transactions were committed with this registrator-account and the more likely it is that there are in fact fraudulent transactions among them.

Because of the fact, that normal fraud detection and internal fraud detection use a different approach, but the program should be able to handle both, it had to be made sure, that there is an easy way to switch between the different detection mechanisms. This had to be done through some switches in the program. It also had to be possible to switch between different detecting algorithms. This could best be achieved through a strategy pattern.

The program should also be able to adapt to different databases and different database structures. So there had to be a way to easily change the name of the database, the name of attributes and different database drivers.

After the programs run, it should be possible to store the results in a database to preserve the data for further processing. Therefore not only a reading database access had to be implemented, a writing access was necessary, too.

Although the main goal of the thesis shifted after a while, optimization was still a topic. So there had to be made some decisions on how to handle the graphical visualization, which not only used a lot of computation time, but also a lot of memory. It was decided to hide the graphical implementation so that the graphs don’t show and the computation gets much faster. For several reasons, testing of the new algorithms among them, it however was decided to not completely delete the graphical interface, so it can still be reactivated via a switch in the configuration file.

As for the topic of financial fraud, the program is specialised in this sort of transactions and the inherent attributes thereof. The program does not have to be easy adjustable to other businesses and therefore, the database-connection only has to satisfy known attribute-counts.

After declaring the problems, the approach to a possible solution and the requirements for this fraud detection program with the main goal to find internal fraud, the next two sections will be the description of how the standard GraphSlider worked and how the extensions in the extended version were coded. It will also explain the design choices behind the different implementations and will give a brief hint of the drawbacks, if necessary.

3 Description

This section contains the description of the two implementations of possible solutions of the fraud detection problem. The GraphSlider is the base program, which can handle one sort of fraudulent behaviour and provides the framework to implement further detection mechanisms. The extended GraphSlider is, as its name states, the improved version, which can also handle internal fraud and has wider variety of functions to be used.

3.1 The GraphSlider

In this section, the first prototype of the new solution will be introduced. The way, how it works will be outlined, key features will be illustrated and the design choices made will be declared. The exact functionality can be found in Appendix A.

3.1.1 Functionality of the GraphSlider

The GraphSlider's main purpose is to find fraudulent chains in the transaction database. Therefore it iterates sequentially through all transactions in a defined time frame. This is given through the START_DATE and END_DATE variables in the configuration.
A sliding window, whose size can also be defined, propagates through all available transactions. Figure 1 shows, how the window propagates forward. Note, that the size of the propagation step can also be defined, but it should at most be equal to the size of the sliding window itself, or some transactions may be omitted.

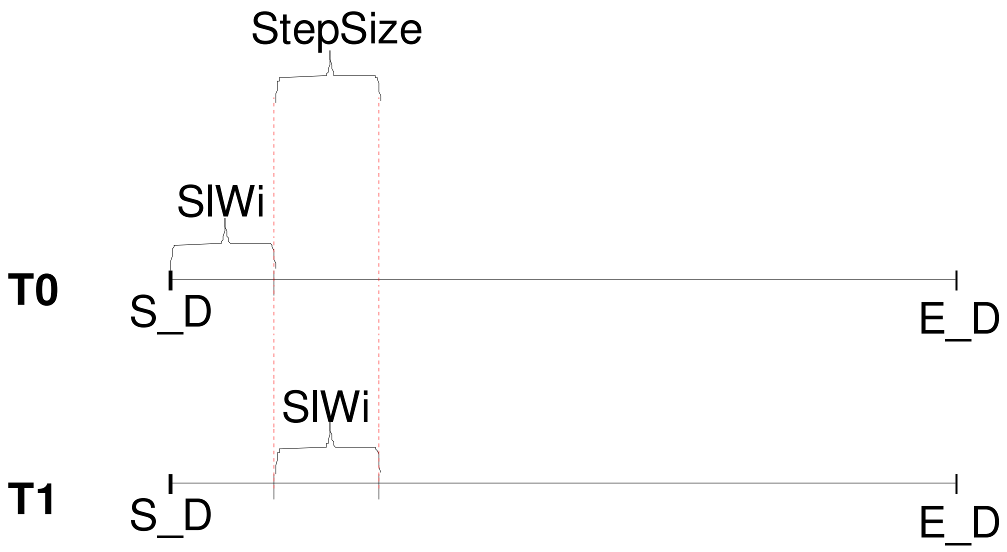


Figure 1: Propagation of the sliding window

The transactions within the sliding window are then compared to each other. To be indicated as fraudulent, a chain must consist of at least two transactions and three involved bank accounts. First, a transaction from account A to account B must take place. All transactions within the sliding window are then searched to have B as originator for another transaction. Say this is the case and a transaction from B to account C takes plate within the window. If such a transaction is found, the amounts transferred are compared. If they are within 10% of each other, the chain is scored. As the minimal scoring is 1 and the scoring threshold of a chain to be marked as fraudulent is also 1, the chain gets marked as fraudulent, as soon, as a scoring occurs.

The scoring takes place on the intermediary account. Depending on the involved accounts, the scoring is bigger or smaller. Figure 2 shows some possible scorings of the account B. The actual scoring is the incoming transactions meeting the conditions multiplied with the outgoing transactions. If the chain of transactions is only between two accounts, they are never scored.

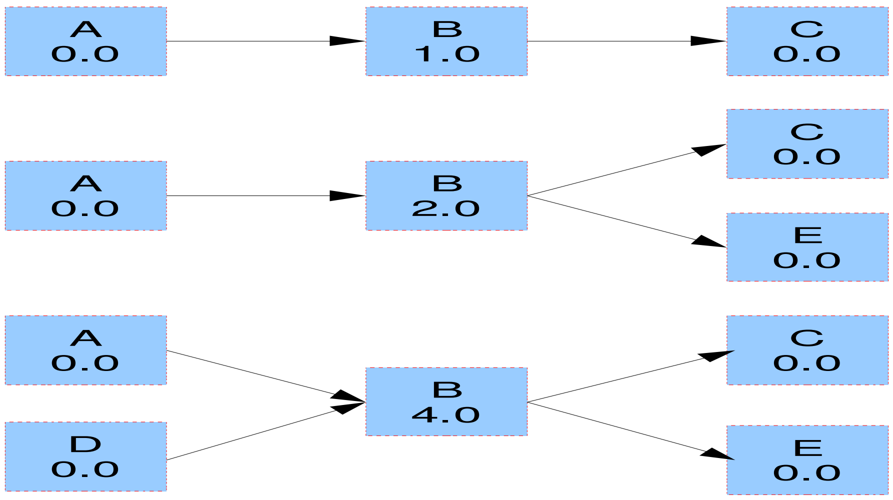


Figure 2: Possible scorings

With this algorithm, it is possible to find bank accounts that only act as intermediaries. The possibility is high, that such accounts are only used for money laundering.

As soon as all transactions are handled and the sliding window reaches the defined END_DATE, the GraphSlider terminates. The accounts, that fired an alert and where marked as fraudulent are then stored in a separate database for further processing.

3.1.2 Conclusion

We showed in this section, how the GraphSlider works. As we have seen, it uses a new approach in fraud detection, as it relies on the temporal relationship between the transactions and the amounts of money transferred to score the individual transactions and the bank accounts involved. With this and the corresponding thresholds it is possible to detect suspicious transactions, to indicate them and to store them in a database for further processing.

3.2 *The extended GraphSlider*

As seen in the previous section, the GraphSlider is a pretty neat solution for the problem of detecting fraud and money laundering on a temporal base. But it does not more, than some of the existing fraud detection programs already do, although it does it in a new way. So we came up with a new idea.

3.2.1 Requirements of the extended GraphSlider

The survey of existing fraud detection mechanisms showed that there are only a few papers about internal fraud and only a single one about internal fraud on the employee level. So the decision was made, to modify the GraphSlider to account for internal fraud as well.

Every transaction in the database has an attribute about its registrar. That means, the employee who was responsible for this transaction is known. With this new attribute, it should be possible to modify the GraphSlider in a way to account for internal fraud. As internal fraud not only consists of the pattern used in the old GraphSlider, it was necessary to formulate the new requirements.

1) The new attribute of the database should be imported into the program. Alerts should not only be fired by accounts, they should also indicate the registrar of the corresponding transaction.
2) The transactions on a regular base between two accounts, should not fire an alert. Transactions on an irregular base should initiate a scoring. If the scoring gets too high i.e. above a certain threshold, an alert should be fired. This should be done to indicate cash flows deviating from weekly, monthly or annual payment.
3) Transactions on a regular base but with a high deviation in the amount transferred should fire an alert anyway. This is to prevent the circumvention of rule 2).
4) Single transactions between two accounts should not initiate a scoring and therefore should not fire an alert.
5) A lot of single, small transactions between two accounts in a short period of time should initiate a scoring. If there are too many, this should fire an alert.
6) All of the above scorings should be weighted, to be able to give them different relevance for the summed up scoring of an account and to have another mechanism to control the firing of an alert besides the already present threshold.
7) At the end, the components of the AlertGraph built during the run of the program should be transferred to a new data structure and then be stored in a database for further processing.

The seven points above set the frame for the new implementation of the GraphSlider. Because the GraphSlider is quite modular, some of the new requirements could be implemented with new implementations of the existing interfaces. The remaining requirements had to be met by modifying the already existing classes or introducing completely new ones.

3.2.2 Concepts of the extended GraphSlider

In this subsection, we would like to take a look at the concepts behind a possible approach to the requirements imposed above. As the seven points differ, it is not possible to cover them with a single approach, although some of them can be combined.

3.2.2.1 Concept of the new main methode

The first requirement is the simplest to comply with. The new attribute of the registrar has to be imported into the program. Then, the program should not process the whole time interval set by START_DATE and END_DATE anymore. It should go through the whole interval several times, once for every registrar. This, unfortunately, can lead to a longer runtime, but is in fact inevitable. At the end of each interval, the now build AlertGraph should be scored as a whole and the scoring should be assigned to the corresponding registrar.

3.2.2.2 Concept of the periodicScore() function

Every Transaction consists at least of a date, the transferred amount, the involved bank accounts and the number of the registrar. As a transaction occurs, it will be stored in a collection, together with all its attributes. The following transactions will be compared to the ones already stored. If one of the new transactions has the same bank accounts as source and target as an already stored one, they are considered to be between the same customers. As this occurs, the date of the first, already stored transaction will be compared with that of the new one and the difference will be stored, together with the new transaction and all its attributes (hereafter called the reference transaction). The old transaction will be discarded to save memory. As soon as a third transaction with the same source and target occurs, its date will be compared to the date of the reference transaction and a new difference will be computed. This new difference will be compared to the already stored one. If they do not match or the delta is not within a certain threshold, which should be defined by the user, the transactions should be considered as irregular. If this occurs, a scoring will be imposed. The amount of the scoring depends on the deviation from the difference stored with the reference transaction. The scoring will then be multiplied with a weighting factor before it gets distributed to the involved accounts.

The base amount of the scoring depends on the deviation of the two differences in dates. That means, the larger the delta between the two differences, the larger the base scoring. The base scoring multiplied with the weighting factor will be stored with both involved accounts, as said before. If these accounts already have a scoring, the new one will be added. Together with the weight factor, the threshold and the tolerance in the deviation, the user is able to make complex adjustments how many deviations it takes to finally fire an alert.

If the delta is within the defined tolerance, two things should occur. First, a new delta should be computed and replace the old one. This is to account for the slight shifting in transaction frequencies. Then, another comparison will be made. As the amounts transferred by regular transactions between the same accounts normally are in a certain range (e.g. 15% of each other), a deviation from this value is suspicious. So even if the transaction is regular in the meaning of being frequent, this could be only an adaption to the detection mechanisms by the fraudster. Therefore, this second comparison is imposed. It checks regular transactions and imposes a scoring, if the amounts transferred are not within this 15% deviation of the amount of the reference transaction. Like the deviation from the date, this deviation is first multiplied by a weighting factor before it becomes the actual scoring of the two involved accounts.

With these two mechanisms, it should be possible to track transactions that are not frequent or irregular and it should be possible to track transactions that are regular, but have a high deviation in the amounts transferred. Figure 3 shows the three possible outcomes of this conceptual mechanism.

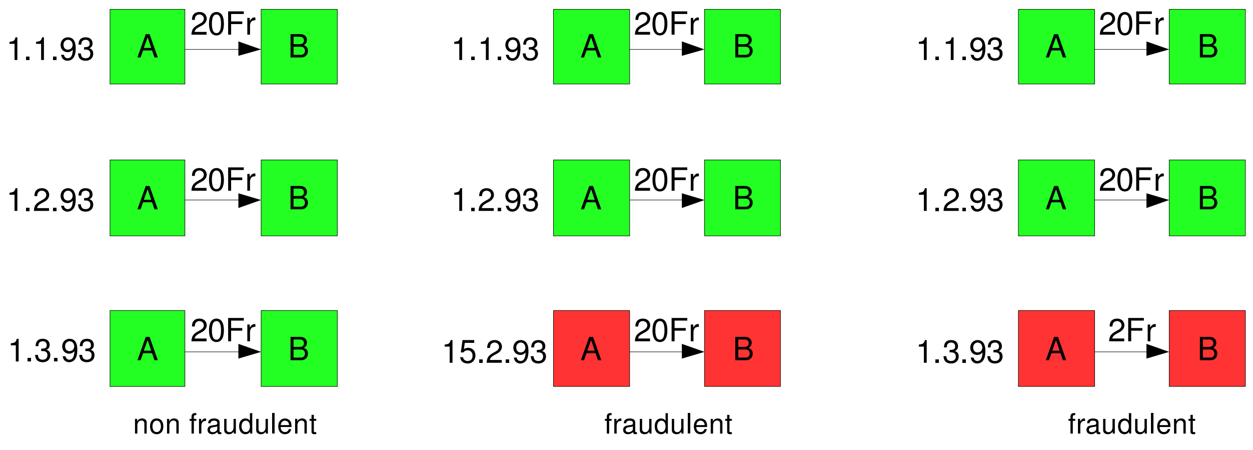


Figure 3: Three possible outcomes of periodicScore(

3.2.2.3 Concept of the sequenceScore() function

To explain the approach to the solution of the smurfing problem, we assume the following simplified transactions:

TRX1	ORIG3	BEN4	DATE1
TRX2	ORIG3	BEN7	DATE2
TRX3	ORIG3	BEN4	DATE3

The algorithm tries to catch the originator- (ORIG) and the benefactor- (BEN) account of every transaction within the sliding window. If there are two or more transactions (e.g. TRX1 and TRX3) that have the same originator (ORIG3) and the same benefactor (BEN4) within the timeframe given by the sliding window, a counter will be raised. The counter will be multiplied with a weighting factor, so that the user is able to take influence on the severity of more than one transaction between the same accounts occurring in a short amount of time. If the counter reaches the threshold, the alert will be fired. After a defined amount of time, the counter will be reset to zero. This should solve problem 4 as it prevents single transactions from firing an alert, if they do not occur all together. This is because a single transaction will not be able to raise the counter high enough to trigger the alert. The scoring mechanism is illustrated in figure 4.

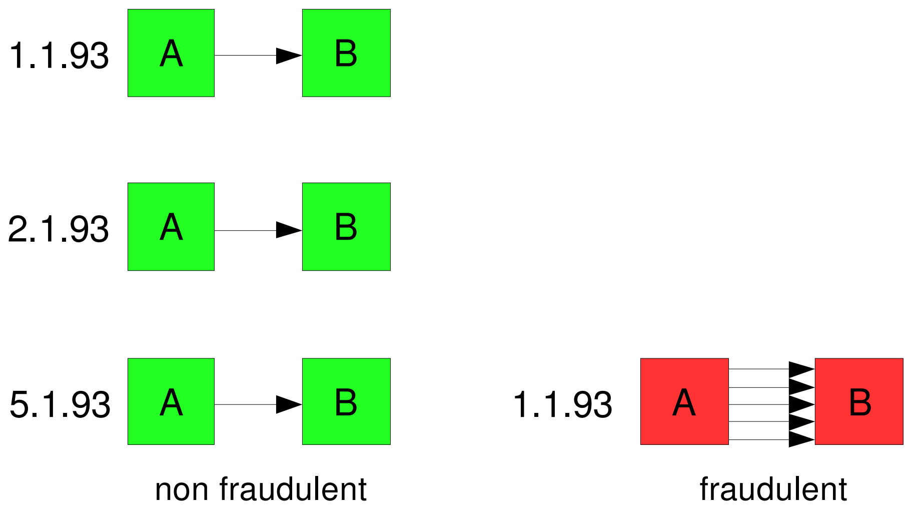


Figure 4: Scoring mechanism of sequenceScore()

3.2.2.4 Concept of the getSuspiciousPatterns() function

The components of the AlertGraph should be collected, identified and stored to a database. Therefore, a deep first search should go through the single sub-graphs and identify all adjacent nodes, which are suspicious bank accounts, interacting with each other.

4 Evaluation

After we have introduced the problem of detecting internal fraud, have made clear our intentions to the approach of the solution and have explained the actual concepts behind a possible solution, this section will cover the evaluation of the extended GraphSlider. The evaluation is based on three main topics. First, we compare the different methods and their evolution in the matter of computation time. We then compare the different algorithms in the matter of memory usage. For this, we will heavily rely on the JConsole program, which is included in the Java SDK (JDK J2SE) distribution since release 5.0. With the JConsole, we are able to show the memory usage over time. We are also able to show, where in the memory most of the objects reside in. We will also be able to have a comparable value of how much computation time is lost to the garbage collection. The last topic is the accuracy of the extended GraphSlider. Therefore we will insert fraudulent transactions into a synthetic database. The evaluation will then show, if these injected transactions are recognized by the program.

4.1 Test configuration

As there are many different configurations possible in the Configuration class for the different scoring algorithms and many of them have an impact on the results of the different evaluations, the standard configuration used for each algorithm will be set and explained in this subsection.

4.1.1 periodicScore()

The following values are set as the standard values for periodicScore() in the Configuration class for the runs on the three databases.

- START_DATE: 1993-01-01
- END_DATE: 1993-06-01

The time period contains 5 complete months. It is to be assumed, that longer time periods will lead to longer computation times.

- SLIDING_WINDOW_SIZE: 15

There are 15 days within the sliding window. Lowering this value can have an impact on the computation time and the results.

- STEP_SIZE: 1

With a step size of 1, we ensure that every transaction is accounted, as the sliding window propagates one day forward every step. With a window size of 15 and a 5 month time period, this leads to 136 steps, as the first 15 days are loaded in one step.

- CHAIN_SCORE_THRESHOLD: 1

As soon as one of the nodes reaches a scoring of 1, an alert is fired.

- PERIODIC_WEIGHT_FACTOR: 0.1

Every day off the stored delta in the periodicScoring() will be weighted with the factor 0.1. A deviation of 10 days off the delta at once will fire an alert.

- PERIODIC_AMOUT_WEIGHT_FACTOR: 0.001

If the transactions are regular, but the amounts transferred have a high deviation from each other, a difference of 1000 monetary units will lead to an immediate alert.

- DIFF_LIMIT: 4

If the deviation between two computed differences in transaction dates is larger than 4 days, the transactions are considered to be not regular and a scoring will be initiated.

- THRESHOLD_PERCENTAGE: 0.15

Amounts, that differ more than 15% up or down the last transmitted amount, will lead to a scoring, because such a deviation is considered to be suspicious.

4.1.2 sequenceScore()

As with the periodicScore() function, the standard values of the constants in the Configuration class for the runs with periodicScore() are given here.

- START_DATE: 1993-01-01
- END_DATE: 1993-06-01

Again, the surveyed period is 5 complete months.

- SLIDING_WINDOW_SIZE: 4

As the sliding window defines how timely close transactions have to be to get tagged as possibly fraudulent, the size of the sliding window has to be much smaller. With a size of 4, transactions with the same source and target within 4 days are considered for scoring.

- STEP_SIZE: 1

This value is again 1, as we want to cover all transactions and we want the sliding window to propagate only one day at once.

- EDGE_STEP: SLIDING_WINDOW_SIZE

This value sets the time interval, after which an edge is deleted out of the collection *edges*. This is mainly to prevent edges from bloating.

- CHAIN_SCORE_THRESHOLD: 1

The threshold is again set to 1. If this is exceeded for any node, an alert is fired.

- INTERNAL: false

As sequenceScore() is mainly for the detection of normal fraud, the INTERNAL switch is turned off. This implies that the registrars are not taken into account and all transactions are considered to be committed by the imaginary registrar with the ID -1.

- SEQUENCE_WEIGHT_FACTOR: 0.2

A weight factor of 0.2 implies, together with the threshold of 1, that there must be a minimum of 5 transactions, meeting the conditions of sequenceScore() within the sliding window to fire an alert.

- AMOUNT_LIMIT: 100'000

As the amount of the split transactions should be relatively small to be considered as smurfing, this constant gives the upper limit for a transaction to be considered by the sequenceScore() algorithm.

4.1.3 Computer specifications

Especially the computation time, the processing of the various transactions take, is very dependent on the hardware. As we used two different computers, which produced significantly different results, the hardware configurations of theses two machines will be given in this section. As only the CPU and the RAM matter, only these two components are listed. To be able to compare the two configurations, the CPU-ranking taken from the PassMark website[2] is also given. This ranking lists the 107 most common CPUs as of December 1st 2008 and their performance with the PassMark CPU benchmark. Every time, it is of relevance, the computer who produced the corresponding values will be indicated. The setup of the two computers is as follows.

CPU (1)	CPU (2)
Intel Core 2 6300 @ 1.86GHz	Inter Core 2 Duo E8400 @ 3.00GHz
3318MB RAM	2048MB RAM
PassMark ranking: 27	PassMark ranking: 9
PassMark score: 1'058	PassMark score: 1'997

[2] PassMark CPU benchmark - http://www.cpubenchmark.net/common_cpus.html

4.2 Computation time measurement

The first performance measure is the computation time. Therefore we made some tests with different databases with different amounts of transactions and registrars. Every fraud detection mechanism encountered three databases. These databases are constructed as follows:

1) 34 transactions,	5 registrars	referred to as “small”
2) 174'723 transactions,	10 registrars	referred to as “medium”
3) 211'270 transactions,	15'000 registrars	referred to as “large”

The time computed by the Timer class was the reference for comparison. The following chart 1 shows the difference in a run on the database with 34 transactions with the visualization turned on and off. The runs were only taken on CPU (1), as this was only to show the impact of the visualization and not the overall power of the involved computer.

Visualization	Time
on	00:03:47.578
off	00:00:01.500

Table 1: Visualization comparison

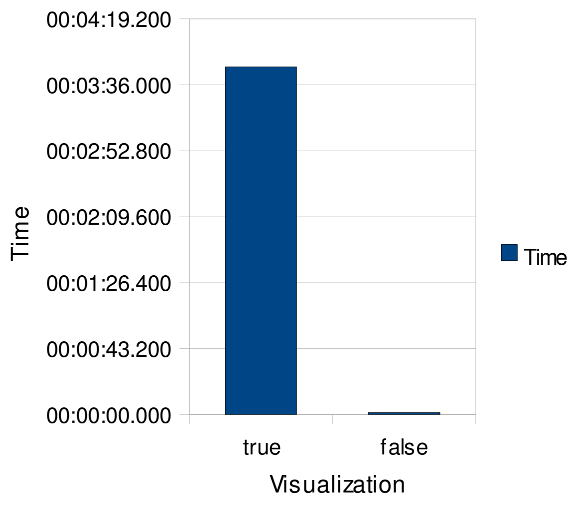


Chart 1: Visualization comparison

It is evident that the visualization has a very big impact on the computation time. With the visualization turned on, the computation takes about 15'000 times longer than without it. We know from earlier tries, that the impact gets smaller, the longer the computation without the visualization takes. For example with the medium database with 174'000 transactions, the computation time goes up from around 15 Minutes to over 3 hours, which still is an increase of 1200%. As it is evident, the visualisation takes a lot of computation time and has no impact on the actual results. Therefore it is turned off for the following measurements.

4.2.1 periodicScore(), internal = false

The first function measured was the periodicScore() with the INTERNAL switch turned off. So the registrars were not considered and there was only one pass. This is a good measure of how big the impact on the computation time is, just based on the amount of transactions in the whole time frame.

Table 2 shows the different databases with their corresponding computation times on the two different CPUs.

Transactions	CPU (1)	CPU (2)
34	00:00:01.470	00:00:00.734
174'723	05:51:54.360	02:58:15.140
211'270	11:16:34.291	07:21:34.141

Table 2: periodicScore(), internal = false

As the computation times of the medium and the large database are much bigger than the ones of the small database, there are two charts to better display the differences between the two CPUs.

Chart 2

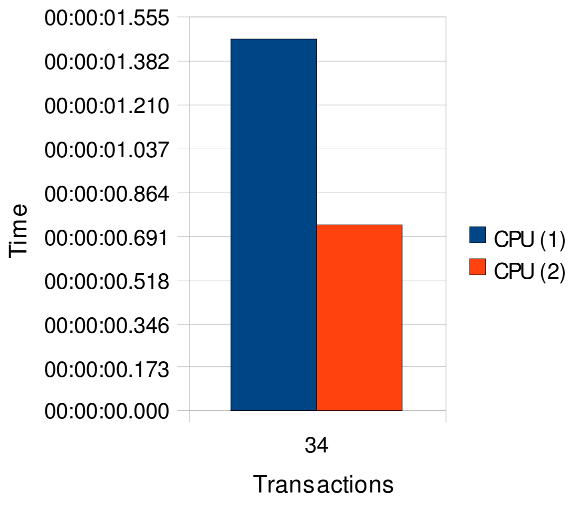


Chart 2: periodicScore(), internal = false, small database

Chart 2 shows, that with small databases, the difference between the two CPUs is obvious. As the PassMark scoring indicated, CPU (2) is nearly twice as fast as CPU (1).

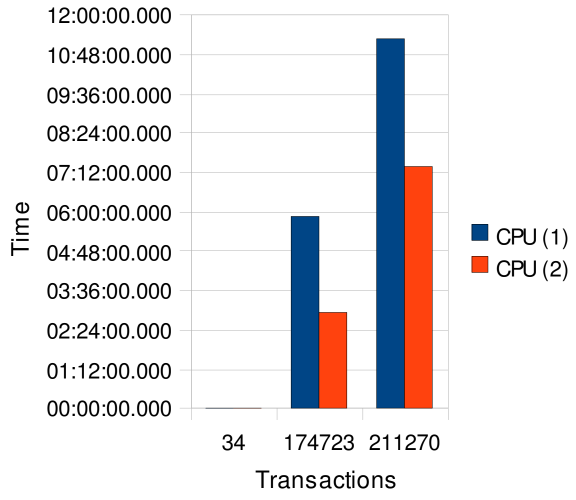


Chart 3: periodicScore(), internal = false, all databases

Chart 3 shows that the computation time increases with the amount of transactions processed. As indicated before, CPU (2) is much faster than CPU (1). However, the gap gets smaller, the more transactions are processed. It also shows that the computation time of the small database is negligible compared to the two larger ones.

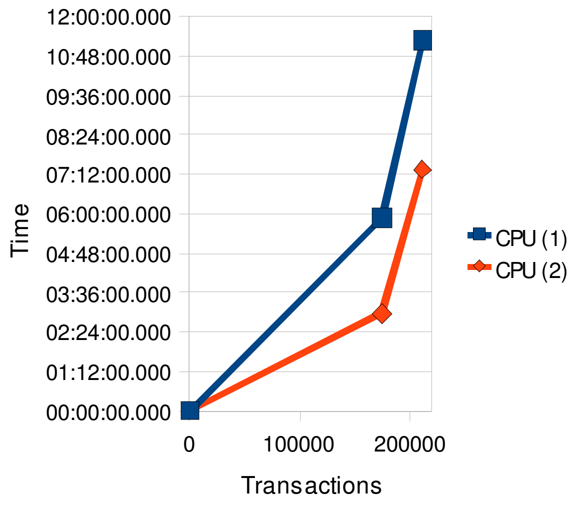


Chart 4: Increase in computation time

Regarding the performance of the actual function, chart 4 indicates, that it does not scale well with the amount of transactions processed. It seems as if the computation time grows exponentially with the amount of transactions processed. 220’000 transactions seem to be on the upper border of amounts processed efficiently by the algorithm. It also has to be considered, that on weaker PCs the computation time will increase significantly.

4.2.2 periodicScore() internal = true

The second measurement was periodicScore() with the INTERNAL switch turned on. Because of the loops that are involved, as soon as the registrars are considered, they have an impact on the overall computation time. As the database with 34 transactions and the database with 174’723 transactions have 5, respectively 10 registrars, they can still be compared. The database with 211’127 transactions is an exception. It has about 15’000 registrars with about 10 to 20 transactions per registrar. This has two effects, as we will explain shortly.

Table 3 shows again the different computation times of the two tested systems.

Transactions	CPU (1)	CPU (2)
34	00:00:01.609	00:00:01.141
174’723	00:24:26.125	00:14:23.828
211’127	~62:14:44.000	~250:00:00.000

Table 3: periodicScore(), internal = true

Chart 5 and 6 show the computation times of the small, respectively the medium database in respect to each other in a graphical manner. As the two charts indicate, the power of the system still has a big impact on the computation time, although it is not as big as before. This can be explained with the fact, that the actual computation does use less time, as there are fewer transactions processed simultaneously.

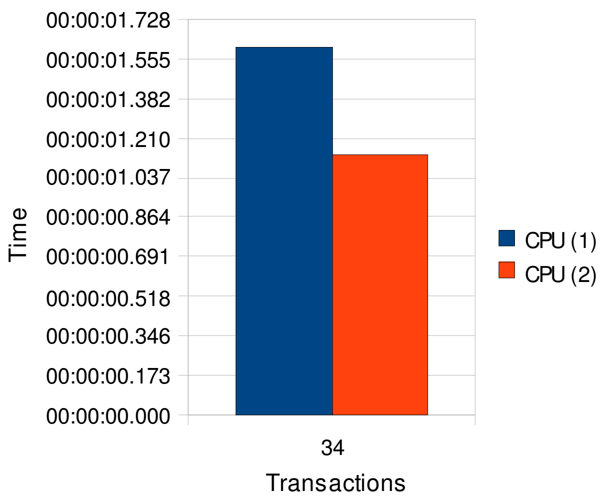


Chart 5: periodicScore(), internal = true, small database

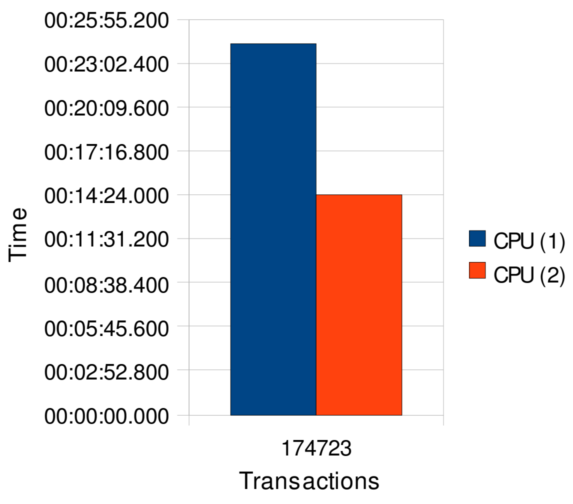


Chart 6: periodicScore(), internal = true, medium database

As stated before, the database with the 211'127 transactions is a special case. Because this was the only database on a server, only reachable though the university network, this had an impact on the computation time. This, together with the fact, that there are about 1500 times as many registrars in the large database as there are in the medium one.

The fact, that the single registrars are handled in a sequential instead of a parallel manner has an impact on the computation time as soon as there are many registrars with only a few transactions per registrar. In this case, most of the computation time is not lost through the actual calculation of the function. It is lost through the various iterations in the main class. As we will see in the memory usage section, the CPU is not really stressed.

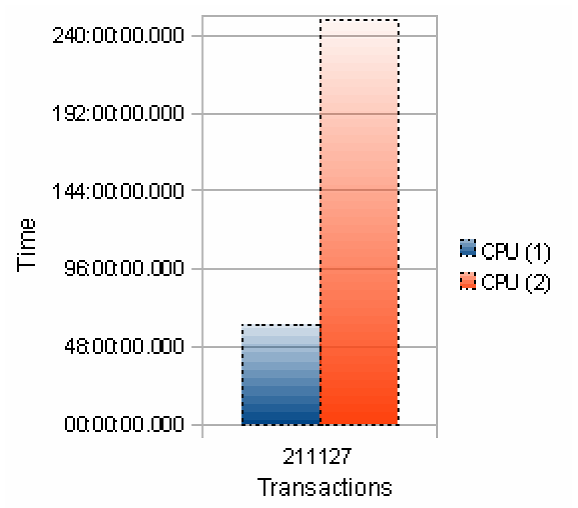


Chart 7: periodicScore(), internal = true, large database

Chart 7 shows the graphical representation of the computation times on the large database. The bars are faded, as the actual values are only estimations. On CPU (1), the program ran exactly 22 hours before it was manually aborted. In this time, 5303 registrars of the 15'000 were processed. As the time per registrar was constant throughout the whole computation, this would have lead to a total computation time of about 62 hours. The interesting thing is that on CPU (2), which was faster in every other computation, the transactions were handled at about ¼ of the speed of CPU (1). As the program terminated after 2:30 hours because of a network failure, the end result is also an approximation. Similar to CPU (1), the time per registrar was constant in these 2:30 hours. Given this constant time, the computation would have taken about 10 days and 10 hours.

Our guess regarding the much bigger computation time on CPU (2) is that it was because of the involved network. As stated before, the large database is housed on a server at the University of Zurich. The CPU (1) was directly integrated in the university-network. CPU (2) had to connect via a secure channel VPN connection though the Internet. In this special case, the fetching of the data took much more time then the actual computation, as there were only very view transactions per registrar and there were nearly any computations (also indicated by a CPU usage of below 3%). As the Round Trip Time (RTT) of the internal university network is much smaller than the RTT though several routing points and the en- and decoding because of the secure channel VPN connection, this lead to a computation time on CPU (2) about four times larger than the one on CPU (1).

These results lead to the conclusion, that periodicScore() in internal mode is not capable of handling too much registrars in a database. As we will explain in the future work section, this should be one of the first issues to be handled when improving the extended GraphSlider.

4.2.3 sequenceScore() internal = false

As the sequenceScore() algorithm is designed to catch smurfing and should not take the single registrars into account, this function was only tested with the INTERNAL switch turned off.

Table 4 shows the actual results of the computations. Note, that for the 3 values designated as NONE, there is no data available. As the computation of the large database took nearly two days, we decided to omit the computation of the medium database, as this result already showed that the algorithm does not scale well with the amount of transactions in the database.

Transactions	CPU (1)	CPU (2)
34	00:00:00.875	00:00:00.703
174’723	NONE	NONE
211’127	42:25:28.470	NONE

Table 4: sequenceScore(), internal = false

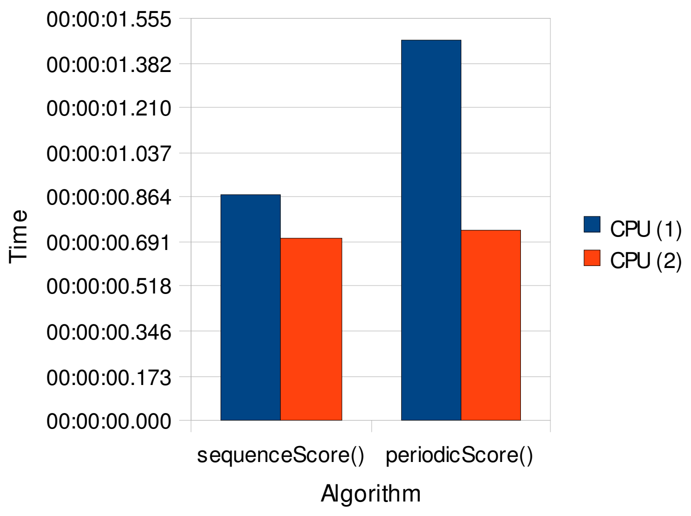


Chart 8: Computation time comparison, small database

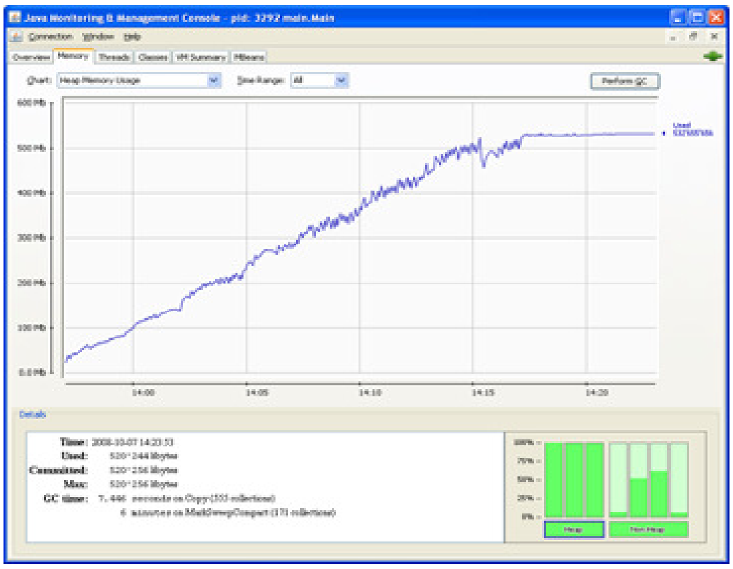
Chart 8 shows, that the difference in computation time between the two CPUs is not as big as with the periodicScore() algorithm when performed on the small database. The faster CPU (2) has nearly the same computation time with both algorithms, which indicates, that at this level, the algorithm used does only have a minimal influence and that the whole program can't get much faster than this. As the computation time of CPU (1) is also getting closer to this point, this means, that the sequeceScore() algorithm is not as CPU dependent as periodicScore(). This may be because there are not that many collections to be searched.

4.3 Memory usage measurement

As the original goal was to improve the GraphSlider in terms of memory usage, this section shows the actual usage after the new algorithms were introduced. As it was no longer a main goal, the usage is not really optimized. Nevertheless, this section gives a good insight into the advantages and flaws of the new algorithms.

All the screenshots are taken from the JConsole. This is a JAVA program capable of connecting to any running virtual machine and displaying the actual values of memory usage, classes loaded, CPU percentage used and threads active. All measurements were taken simultaneously with the recording of the computation time.

After the first implementation, the extended GraphSlider had some problems with the actual heap space. It was set to 60 MB by default. This resulted in heap space overflows and the termination of the program. We tried to set the heap space to 512 MB, as we thought that this should be enough for the program to be efficient. The first tries resulted in the following JConsole chart (chart 9).



```
      Time: 2008-10-07 14:23:33
      Used:    520'244 kbytes
 Committed:    520'256 kbytes
       Max:    520'256 kbytes
   GC time:  7.446 seconds on Copy (333 collections)
               6 minutes on MarkSweepCompact (171 collections)
```

Chart 9: JConsole, Heap overflow

The three fully filled green bars in the lower right corner indicate, that all three heap spaces were filled and therefore, with 512 MB, there was still a memory overflow. We discovered after an intense search, that this was caused by a database connection not closed after fetching data from the database and therefore was only a bug.

As the memory usage did not differ on the two used CPUs, only the charts made with CPU (2) will be showed in this section. Because of the fact, that the memory was constrained by the committed 512 MB and the actual physical memory available does have no impact as long as it is larger than the committed heap space, the smaller RAM of CPU (2) made no difference.

4.3.1 periodicScore() internal = false

Because the computation of the periodicScore() function with the small database with only 34 transactions took less than a second, there is no graph available.

The medium database showed the following behaviour.

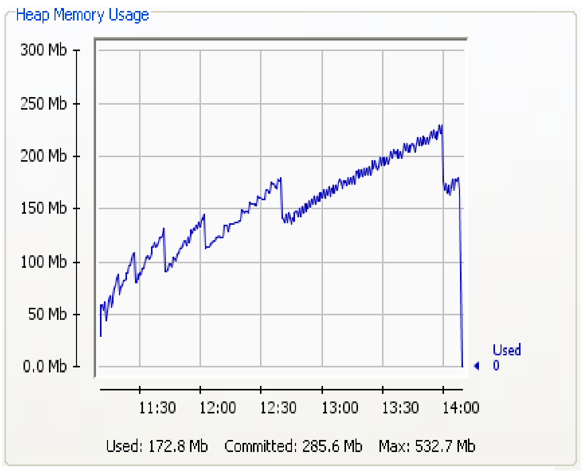


Chart 10: periodicScore(), internal = false, medium database

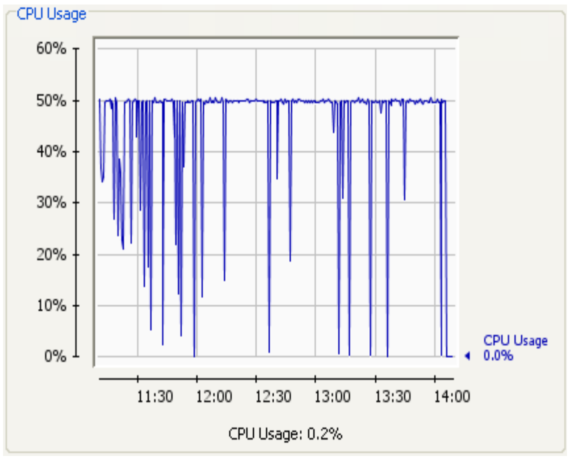


Chart 11: periodicScore(), internal = false, medium database

As chart 10 shows, a bit less than half of the maximum memory gets used by the program. Because of the fact that the transactions are all handled as if there were only one (imaginary) registrar, the stored transactions sum up. The CPU usage is at the allowed maximum (50%) most of the time, because there are a lot of transactions to be compared and scorings to be computed.

The charts of the large database show a quite similar situation. The only real difference is that the memory usage increased a lot, as chart 12 shows, and is near the magical border of 512 MB. This means, a few more transactions, and the program would have run into a memory overflow.

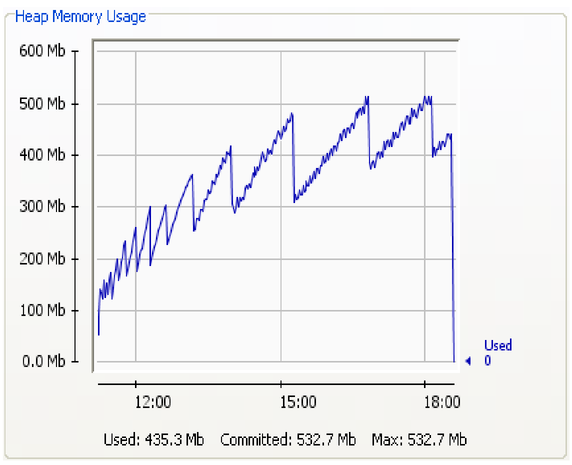


Chart 12: periodicScore(), internal = false, large database

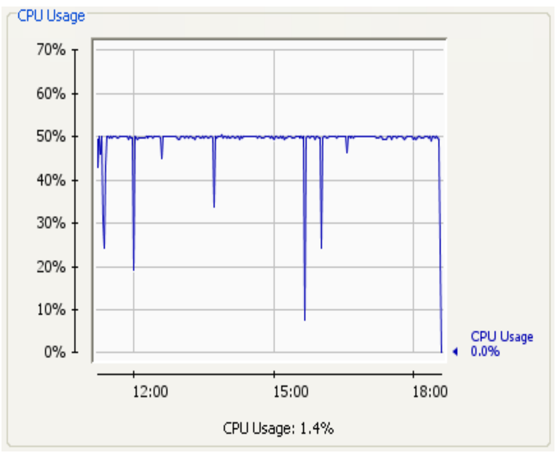


Chart 13: periodicScore(), internal = false, large database

4.3.2 periodicScore() internal = true

Like the periodicScore() with the INTERNAL switch turned off, the computation of the small database was again much too quick to get an actual graph with the JConsole.

The charts of the medium database show some interesting details. Although the collections are cleared after every registrar, some objects were not deleted and stacked up in the so called tenured space. This section of the heap space contains objects that survived several passes of the garbage collection. It is not completely clear, why these objects persisted, especially as there seems to be a correlation between this persistence and the amount of transactions per registrar, as the charts of the large database will show.

Because there are fewer transactions to be compared with each other per registrar, the CPU usage drops, as chart 15 indicates. But the more transactions there have to be searched, the more CPU time is used. There are 10 registrars in this database, so the 10 peaks in the CPU usage chart indicate the end of the computation of each registrar.

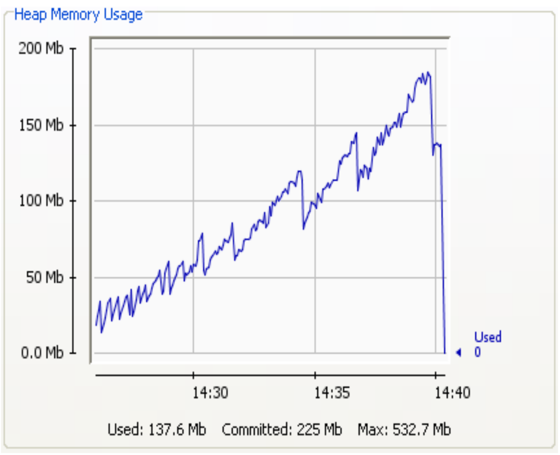


Chart 14: periodicScore(), internal = true, medium database

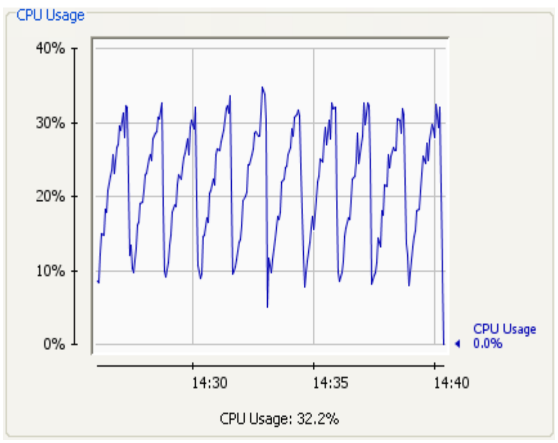


Chart 15: periodicScore(), internal = true, medium database

The charts 16 and 17 show the results of the run on the large database. Although there are more transactions in this database than in the medium one, the memory usage is a lot smaller. There are 15'000 registrars in this database, so the collections get cleared a lot more often. But although this is similar to the case in the medium database, there seems to be no "stacking" problem here. It is not completely clear, why this is the case. Our guess is that the underlying y-files are the problem. The y-files offer some special collection classes. These are specially designed to contain nodes and edges, but have no function to clear them. The reason for this is not evident and does not really make sense in our eyes. Nevertheless, they are used in the alert graph. As there are alerts in the medium database, but none in the large one, it seems evident, that this bloating of the program is an effect of these special collection classes and they should be replaced.

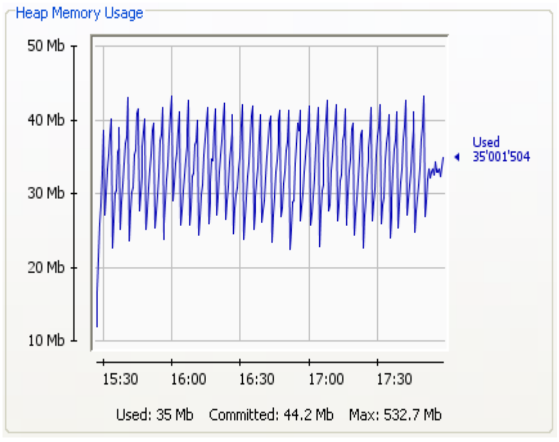


Chart 16: periodicScore(), internal = true, large database

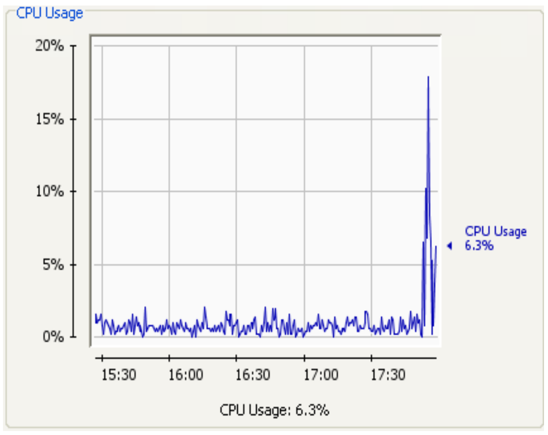


Chart 17: periodicScore(), internal = true, large database

As there are only very few transactions per registrar, there are nearly any comparisons or computations. Therefore, the CPU usage is very low (below 3%). The peak at the end of chart 17 comes from the before mentioned network failure. When the network failed, the program tried to get a connection to the database, but could not reach it and therefore began to throw exceptions. This had an impact on the CPU usage.

4.3.3 sequenceScore() internal = false

With the sequenceScore() algorithm, the small database was handled too fast to get a chart, as with the two periodicScore() variants.

The only existing chart is that of the sequenceScore() performed on the large database on the Merlin server. The very interesting fact in chart 18 is, that it looks similar to the charts 10 and 12, although sequenceScore() is a quite different algorithm. This is another indicator of the fact, that the y-files have some design flaws that are problematic for our approach.

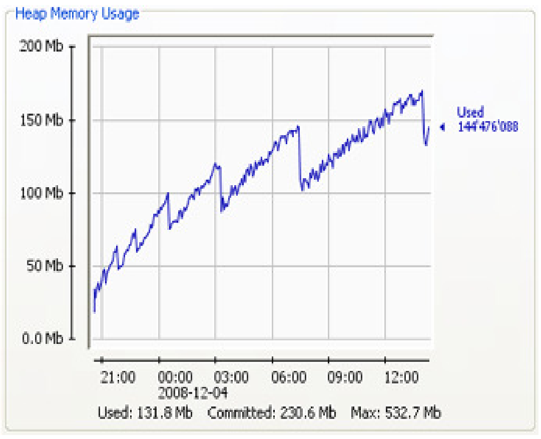


Chart 18: sequenceScore(), internal = false, large database

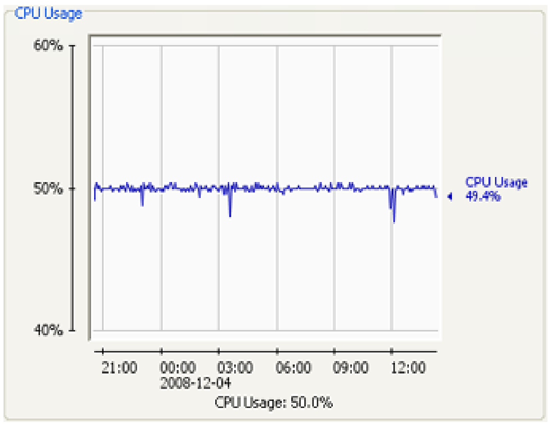


Chart 19: sequenceScore(), internal = false, large database

As expected, chart 19 shows that the CPU usage is again at the maximum mark of 50%. This because of the many transactions handled simultaneously as the program is not in the internal mode. This leads to the conclusion, that as soon as the program is not in internal mode i.e. does handle a lot of transactions at once, the CPU is heavily used and other programs running simultaneously can have an impact on the performance of the extended GraphSlider.
This section showed that the extended GraphSlider is not very well optimized on behalf of memory usage. This, together with the parallelism of the transaction handling, should be the first thing to improve. It is assumed, that omitting the y-files and replacing them with a simpler class with similar functions can really boost the overall memory performance.

4.4 Accuracy measurement

To test the accuracy of the two functions, based on the results of the two previous sections, two new databases were introduced. The new databases are:

- test_ss with 2053 transactions and 10 registrars
- test_ps with 2762 transactions and 10 registrars

These two databases were generated synthetically with random data. The suffixes stand for the functions, the databases are made for. The ss stands for sequenceScoring() and the ps stands for periodicScoring().

4.4.1 Approach

The main goal of this section is to test, if the functions are able to indentify possibly fraudulent transactions in a database with other transactions. As the functions are designed to make a pre-sort of suspicious transactions, the main goal lies in producing no false negatives. False positives are possible, as they only indicate that these transactions are suspicious in the way that they match the given conditions.

To test the functions, fraudulent patterns were manually introduced in the previously generated new databases.

Three chains were injected into test_ps, whereof two are fraudulent, and one is regular and should not be indicated at all. One of the two fraudulent chains is irregular in terms of temporal shifting. The other is irregular in terms of the amounts transferred. In the generated database, registrars 3, 4 and 5 have a scoring of zero. In order to facilitate the test, the fraudulent chains were injected into these three registrars. If a scoring occurs, it can easily be identified.

To test the sequenceScore() algorithm, two chains were injected into test_ss. As the standard search algorithm has a sliding window of four days and the weighting factor is set to 0.2, one chain will have 6 transactions between the same accounts within 4 days and the other will have 6 transactions but within 5 days. If the sequenceScore() algorithm works as intended, the first chain should be scored and the second should be ignored.

4.4.2 Preparations

We injected the following transactions into test_ps.

Transaction-ID	Originator	O-Account	Benefactor	B-Account	Date	Amount	Type	Registrator
2733	1	9999	2	9998	1993-01-01	2000	T	5
2734	1	9999	2	9998	1993-02-01	2000	T	5
2735	1	9999	2	9998	1993-03-01	2000	T	5
2736	1	9999	2	9998	1993-01-01	2000	T	3
2737	1	9999	2	9998	1993-02-01	2000	T	3
2738	1	9999	2	9998	1993-02-15	2000	T	3
2739	1	9999	2	9998	1993-01-01	2000	T	4
2740	1	9999	2	9998	1993-02-01	2000	T	4
2741	1	9999	2	9998	1993-03-01	200000	T	4

Table 5: Injected transaction, test_ps

Because there are no accounts with the numbers 9999 and 9998 we chose this numbers to nullify the possibility of an interference with already existing transactions. To identify the manually injected transactions, they were marked as from type T like “test”.

To test the periodicScore() algorithm, we injected the following transactions into test_ss.

Transaction-ID	Originator	O-Account	Benefactor	B-Account	Date	Amount	Type	Registrator
2017	1	7777	2	7778	1993-03-01	200	T	11
2018	1	7777	2	7778	1993-03-01	200	T	11
2019	1	7777	2	7778	1993-03-01	200	T	11
2020	1	7777	2	7778	1993-03-01	200	T	11
2021	1	7777	2	7778	1993-03-01	200	T	11
2022	1	7777	2	7778	1993-03-01	200	T	11
2023	1	9999	2	9998	1993-04-01	200	T	12
2024	1	9999	2	9998	1993-04-01	200	T	12
2025	1	9999	2	9998	1993-04-01	200	T	12
2026	1	9999	2	9998	1993-04-01	200	T	12
2027	1	9999	2	9998	1993-04-05	200	T	12
2028	1	9999	2	9998	1993-04-05	200	T	12

Table 6: Injected transaction, test_ss

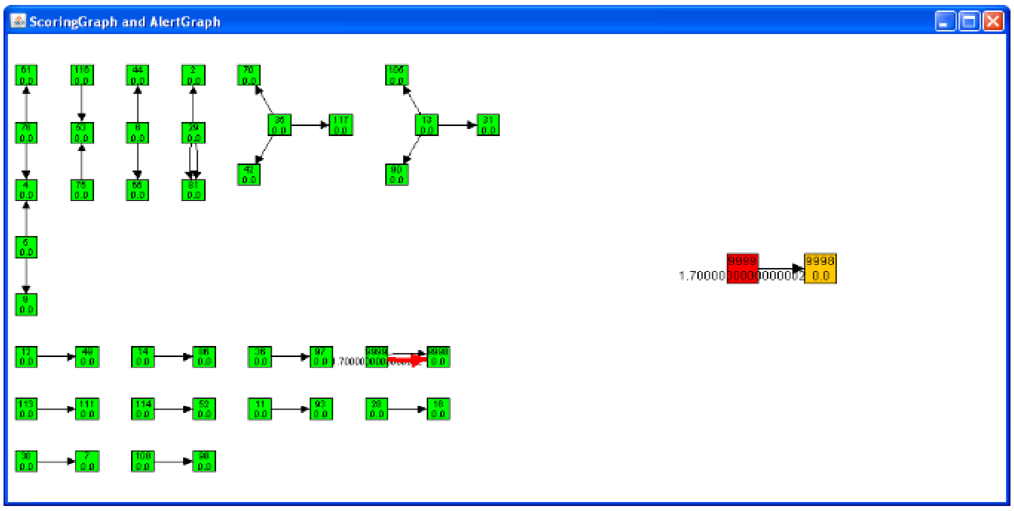
Like in test_ps, the accounts 9999 and 7777 were chosen to avoid conflicts with already existing accounts. The registrars 11 and 12 do not have an actual impact on the computation, as the registrars are not considered in sequenceScore(). But they help to identify the two chains.

4.4.3 Test

We tested the algorithms with their corresponding databases and came to the following conclusions.

4.4.3.1 periodicScore()

Screenshot 1 shows the actual scoring of periodicScore(). The scoring takes place in registrar 3, because the injected transactions are not regular.



Screenshot 1: periodicScore()

The screenshot was taken before the scoring is accounted to both nodes. The red arrow on the left side, in the visualized search graph, indicates the transaction triggering the scoring.

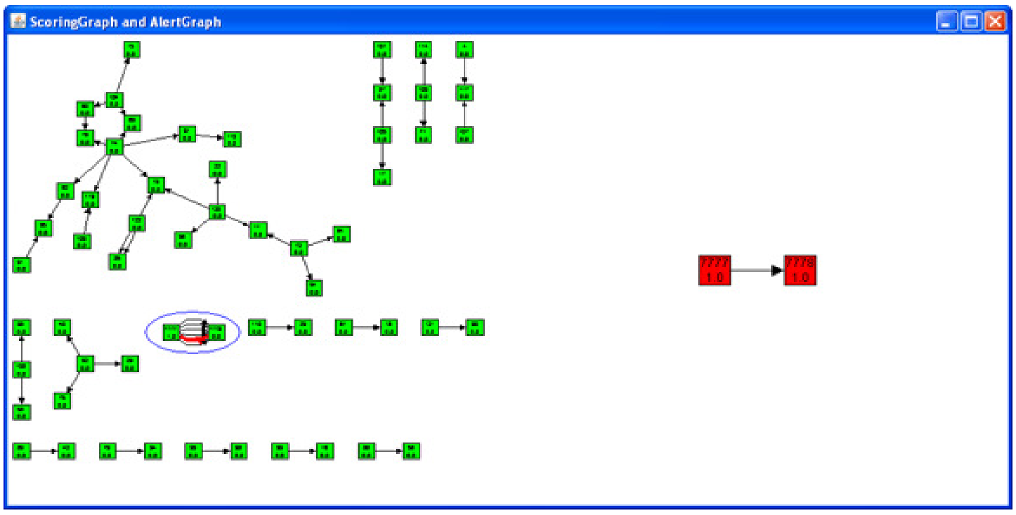
Table 7 shows two excerpts of the *accounts* database, which stores the overall scores of the registrars. As it is shown, registrar 3 and 4, in which fraudulent chains where injected, in fact have a scoring after the injection. Registrar 5, in which a non-fraudulent chain was injected for cross checking, does still have a scoring of zero. This results indicate, that periodicScore() is indeed capable of detecting fraudulent chains and it ignores non-fraudulent ones.

Registrar	Before injection	After injection
3	0	3.4
4	0	396
5	0	0

Table 7: Scorings before and after injection

4.4.3.2 sequenceScore()

Besides the already existing scoring from the original database, only the chain between the accounts 7777 and 7778 was scored. The following screenshot shows the actual scoring.



Screenshot 2: sequenceScore()

On the left side, the search graph is displayed. The red arrow indicates the actual scored transactions. The blue ellipse, which was manually inserted into the screenshot, surrounds the two accounts and the six simultaneous transactions. The right side shows the alert graph with the two scored nodes.

Table 8 shows an excerpt from the component database. There are only the already existing node and the two new scored nodes, which implies, that the sequenceScore() algorithm worked as intended and did not score the second injected chain.

ComponentRegistrator	ComponentNumber	ComponentNode
-1	1	72
-1	2	7778
-1	2	7777

Table 8: Final component database

4.4.4 Remark about the accuracy measurement

The actual test of the two newly introduced algorithm showed, that they are in fact capable of detecting the suspicious chains in their databases. The test also showed that the algorithms ignore non-suspicious chains, even if they are close to be suspicious. With other parameters, it is likely, that other chains would have been detected. This implies that it is very important to make good decisions on which values to be set in the Configuration class.

This concludes the evaluation section. In the next section, we will talk about the advantages of our new approach, but also about the drawbacks which were already known or which where showed by this evaluation.

5 Discussion

In this section, we will discuss the advantages and disadvantages of our new approach to the fraud detection problem. This section will also cover the limitations and drawbacks of the methods, mainly showed by the evaluation.

5.1 Advantages of the new method

As we showed, the new methods are capable of finding two different forms of possibly fraudulent transactions.
sequenceScore() is a good method to find smurfing transactions. It is capable of identifying the involved nodes and marking them as possibly fraudulent for further investigation. With the different customizations, the user is capable of detecting different forms of smurfing. He can detect a lot of small transactions within a certain timeframe or he can search a whole time interval for a certain amount of transactions below a defined threshold. The different configuration values can be adjusted to the actual situation and the effects on the result can vary.
Unlike the manual identification of such transactions, our method is much faster for small amounts of transactions and quite accurate. Because of the fact that the new methods are only responsible for a presorting, false positives are not as bad as false negatives. It was showed in the evaluation, that the sequenceScore() algorithm is capable of finding the suspicious transactions, depending on the values set in the configuration. The evaluation also showed, that transactions, that are close to be suspicious are ignored, so false positives can happen, but are not likely. False negatives are not produced, as the algorithm is capable of dependably finding the injected fraudulent patterns.
It is hard to say, if this is a real advantage, but it is nevertheless a special feature, that the algorithm also works on a temporal base. Instead of just finding a pattern that matches a given subpattern, sequenceScore() decides, if the transactions meet certain conditions in the given timeframe. This makes the program more flexible and the user is able to define the configuration values based on data that can change with experience gained.
periodicScore() on the other hand, is a special method of finding possibly fraudulent transactions and chains in the way that it is tuned for internal fraud detection. As we have seen in the survey [Phua et al. 2005], internal fraud on the employee level is a problem not addressed by most of the papers published till now. Here lies its big advantage. The user is able to use it as a normal fraud detection algorithm for all transactions in the database or in

the specified time interval. But it is that flexible that it can be switched to internal mode, where it only surveys the transactions of the single registrars. With this, the fraudulent transactions can be tracked and assigned to the registrars. These results can then be used, to make further investigations to try to uncover fraudulent behaviour committed by this employee.

Other than normal pattern matching algorithms, periodicScore() tracks the regularity of the surveyed transactions. This is a new approach, as it uses the temporal data, available in almost every transaction database, to uncover possibly fraudulent chains. It also has a fallback mechanism, if fraudsters were able to discover the method behind the monitoring. If the temporal pattern matcher is discovered, the fraudsters will still have to comply with the ±15% rule, or the transaction and the corresponding accounts still get marked as possibly fraudulent.

5.2 Disadvantages of the new method

Although our approach shows a new way of detecting fraud in banking transactions, there are some disadvantages implied by the way, the detection mechanisms work.

sequenceScore() is only capable of detecting smurfing chains between the same two accounts. If the transactions are split among several accounts and thereafter recombined on a third level account, the algorithm is not capable of detecting them. sequenceScore() would only detect them, if there are enough transactions between the initial and one of the intermediary accounts to trigger an alert.

One of the advantages, the customizability, is also some sort of disadvantage. Even if the algorithm is capable of detecting most of the fraudulent chains with the proper settings in the configuration, wrongly set values can lead to the algorithms not detecting any suspicious chains at all.

One of the major disadvantages of periodicScore() is the lack of the capability to adapt to a shift in regular payment. For example, if someone pays its rents always at the first of each month, his transactions are not scored. As soon as he shifts to a payment in the middle of the month, all his transactions after that shift get marked as possibly fraudulent, because of the deviation from the stored value.

Another disadvantage is the fact, that February only has 28 or 29 days. Depending on the value set in the configuration, this can lead to a false scoring, as the difference between the dates gets to big and it will be considered as not being regular. This happens because the algorithm does calculate with absolute rather than relative values.

5.3 Drawbacks of the new methods

Unfortunately, the new methods have some drawbacks, some of them discovered in the evaluation, some of them already known at the beginning of the thesis.

The y-files are a good framework for graphs in many situations. They provide specific operations and graphical representations to handle graphs. The y-files are even used in commercial programs, like the DB Visualizer. Unfortunately, the y-files are not suited for the use in our new methods. We were not able to completely discover why, but they have some issues with objects, that don't get deleted because of still existing references. We presume that the problem arises in the collections provided by the y-files. Unlike the built in collection classes of Java, these collections do not provide a way to clear all contained elements. We presume that this is one of the reasons for the high memory usage discussed later in this section.

Another drawback of the y-files is that they are a proprietary framework. As soon as our new methods would be used in the praxis, they would have to be replaced by other methods to avoid licensing problems. As the y-files are a proprietary framework, they are like a black box. If it would be necessary to extend the framework, this would be hardly possible.

The second, not very severe, drawback is the GUI. It is nice to have a graphical interface, especially for error detection and for better comprehension. But the GUI of the extended GraphSlider has the drawback to be very slow. This is also a limitation of the y-files, as they require pauses between the drawing of some of the elements to prevent the program from running into null pointer errors. These pauses have to be hard-coded and are set to 0.3 seconds. Whenever there are many elements to be drawn, the program slows down significantly. Nevertheless, the program still runs into errors based on the GUI as soon as the user tries to manipulate something in the GUI during the runtime. Normally, these errors have no impact, but they can sometimes lead to a termination of the program. As soon as the GUI is disabled, the errors disappear.

The method how the single nodes are scored seems to have a side effect. If there is an edge, that produces a scoring, both attached nodes get the scoring as intended. But if there is another edge from the originator node within the time frame, that normally would not be scored a all, this edge and its benefactor node get a scoring, too. This scoring is 0.0 and can easily be deleted or ignored, but it can be confusing to have nodes with a zero scoring in the graph.

As mentioned before, the memory usage of the extended GraphSlider is one of its major drawbacks. As seen in the evaluation section, the memory usage rises with the amount of transactions processed. It is possible to circumvent a memory overflow by addressing more memory to the program via the Xmx and Xms parameters, but with the data produced in the evaluation, we can assume, that the memory usage will grow further.

The other big drawback is the exponential increase in computation time. As long as periodicScore() is in internal mode and the number of registrars is small, the computation time is moderate. But because of the sequential manner the transactions and registrars are processed, the computation time increases to an intolerable extent. The same goes for the computation time of sequenceScore(). If there are only few transactions (a few thousand) the performance is good. As soon as there are more transactions, the computation time goes up.

In this section, we discussed the advantages and disadvantages of our new approach over. We showed that with our approach, we are able to reliably detect possibly fraudulent transactions in a database. But this section also showed that there are some major drawbacks, which should be addressed. The next section will deal with this topic.

6 Final remarks

This section contains the final remarks of the thesis and the practical work behind it. First a conclusion over the whole work will be given. It should give a brief overview of what was discovered during the thesis and the programming of the extended GraphSlider. Then, suggestions for future work will be given. As we discovered some advantages but also some drawbacks in our approach, this section will give a hint, what to do next. The section will be concluded with some final personal remarks about the work, included in a short résumé.

6.1 Conclusion

This thesis was started with the goal of optimization in mind. We wanted to optimize the already existing framework of the GraphSlider to use less memory and to make its computations faster. After a short time, we discovered, that in the actual literature there is a fascinating part of fraud detection, not yet covered by many papers. So we decided to switch to a new topic. The new goal was to prove, that it is possible to detect fraud with temporal data, available with almost every bank transaction and that this data can also be used to discover fraudulent behaviour on the employee level.

The first thing to do was to discover the possibilities of the already existing framework and to find a way to implement the new functions. Because of the fact, that the GraphSlider already was very extendible, the new functions could be relatively easy integrated. We decided to take two different approaches. One should cover smurfing and not take the registrars into account. This method should prove, that it is possible to discover fraudulent behaviour based on temporal data and the accounts involved in the transactions.

As the evaluation showed, our new method was in fact capable of detecting this kind of fraudulent transactions. Unfortunately, the evaluation also showed that the new function needs a lot of optimization.

The second approach was to take the regularity of transactions into account. Non fraudulent transactions are often made at the same time of the month and are from approximately the same amount. Our approach should be able to find transactions deviating from this fact and mark them as possibly fraudulent. This, combined with the capability to indicate the registrar in whose account the transactions were committed, resulted in the implementation of periodicScore().

Again, the evaluation showed that this approach is capable of doing what it was intended for. But again, the evaluation also showed that the implemented functions were by no means optimal.

To cover the two approaches, the GraphSlider framework had to be heavily extended. Although it provided the before mentioned extensibility, mainly based on the fact, that most of the existing parts were implemented in a strategy pattern, some parts had to be modified. These modifications, together with the not optimized new functions lead to the contrary of the original goal. The computation time went up for certain configurations of the underlying databases and so did the memory usage.

Therefore, we would like to point out, that the extended GraphSlider should be viewed more as a proof of concept rather than an actual applicable and optimized solution. It should be possible to extend and optimize these new approaches, but this would require more time than this diploma thesis grants.

Finally it is to say, that our new approaches show a new, interesting way to cover the problem of fraud detection, especially on the internal and employee level. But it also shows in some sort of a painful way, what can happen, if programs are not optimized.

6.2 Future work

The base set with the extended GraphSlider provides many possibilities for future work. One of the most pressing issues is the optimization of the GraphSlider in terms of computation time and memory usage. These two issues prevent the actual program from being really efficient.

The first step towards optimization should be the replacement of the y-files. As many useful features they may provide, they cause some serious issues. Not only that they are suspected to cause some of the high memory usage, they are also not capable of visualize the processed transactions without errors, if the processing is too fast.

The next step should be the optimization of the main functionality of the extended GraphSlider. As of now, all transactions and registrars are handled sequentially. For the transactions, this is not a problem, as they have to be handled that way because of the sliding window. But if the registrars are handled sequentially, the computation time rises with the amount of registrars in the database. As long as there are many transactions per registrar, there should be not big difference between if they are handled sequentially or in parallel. As soon as there are only a few transactions per registrar, the processor is idle most of the time, but the program takes as long as if there were many transactions. This is because even if there are no

transactions in a registrar account at all, it still would move the sliding window through all possible dates. Therefore, the first optimization step should be to find a way to handle all transactions and registrars in parallel.
The actual optimization of the two approaches should be the next step in a future work. Especially the way, the collections of sequenceScore() are handled is not optimal. There should be a better way to find transactions with matching originator and benefactor nodes than iterate through the whole collection of already stored transactions.
Maybe this could be done by a smarter implementation of the database query. So that only the transactions are loaded into the program, that meet certain conditions. This would lead to less transactions in the actual program, what would have an impact on both, computation time and memory usage.
periodicScore() should be optimized in a similar manner, even if the problem of too many iterations is not as imminent as with sequenceScore(). A second thought for future work on the periodicScore() algorithm could be the optimization of the comparisons. Now, they are some sort of crude. It would be an idea to not compare the fixed differences between the dates. Instead, some sort of standard deviation could be used. And a moving average should also be introduced, as this would cover the possibility of a shift in payment behaviour.
As already declared, the actual implementation of the GraphSlider is only capable of marking possibly fraudulent transactions. The built alert graphs are then stored in a database for further processing. One possible future work could be to process the built alert graphs directly in the GraphSlider program. This would require to process the graphs with a pattern matching algorithm so find the real fraudulent chains.
As seen, the actual GraphSlider provides many possibilities for future work. It would also be possible to implement another fraud detection mechanism, e.g. on a spatial base. But the main goal should be, to optimize the existing program and take it from it's actual state as proof of concept to an actual productive and efficient system.

6.3 Résumé

This is my personal résumé about this thesis.
I took this thesis, because it gave me the opportunity to do something praxis oriented. As a student, most of the time, the matter you learn is very theoretical. Sometimes you have to do some practical work, but at the end, it is not really used, as it is something that already exists. With this thesis, I had the opportunity to develop something new, not yet present.

The actual work was very interesting, as I discovered new ways to use my Java skills and improve them. I also had the opportunity to work with a already existing framework, based on patterns I only knew theoretically.

During the work, I found out what it means to be looking for a needle in a haystack, or to look for bugs in other words. I discovered, that bug tracking can be very time consuming and that the actual bug can be not more than a singe line omitted in the code.

During the evaluation, I discovered that sometimes, programs behave other than they are intended to do. I always tested my implementations with a small amount of data in a manually written database. These tests all performed pretty well and the injected fraudulent patterns were always discovered. As soon as the program got hands on a bigger database, things changed a little. The fraudulent patterns were still discovered, but at the price of a high memory usage and computation times sometimes that big, that I had to suspend the actual computation and make some extrapolations instead.

At the end, I wish I had done some of the implementations and decisions in an other way, because as soon as you can see the whole picture (which you can hardly see at the beginning) there are always things that could have been done better, like the parallel handling of the registrars instead of the chosen sequential handling.

Finally I have to say, that it was fun after all and I have learned much about fraud detection and the implementation and ideas behind fraud detection systems.

7 References

[Agrawal et al. 1995] R. Agrawal, R. Srikant. „Mining Sequential Patterns,“ IBM Almaden Research Center, San Jose, CA, USA, 1995

[Balton et al. 2002] R. Balton, D. Hand. “Statistical Fraud Detection: A Review,” Statistical Science, Vol.17, No.3, 2002, pp. 235-255

[Barse et al. 2003] E. Barse, H. Kvarnstrom, E. Jonsson. „ Synthesizing Test Data for Fraud Detection Systems,“ Proceeding of the 19th Annual Computer Security Applications Conference, 2003, pp. 348-395

[Bernstein 2005] A. Bernstein. “So what is a (Diploma) Thesis? A Few thoughts for first-timers.,” Department of Informatics, University of Zurich, Switzerland, 2005

[Bettini et al. 1998] C. Bettini, X. Wang, S. Jajodia. “Mining Temporal Relationships with Multiple Granularities in Time Sequences,” DSI Department, University of Milano, Italy, 1998

[Giugno et al. 2002] R. Giugno, D. Shasha. “GraphGrep: A Fast and Universal Method for Querying Graphs,” Department of Mathematics and Computer Science, University of Catania, Italy, 2002

[Held et al. 1987] J. Held, J. Carlis. “MATCH – A New High-Level Relational Operator for Pattern Matching,” Communications of the ACM, Vol.30, No.1, 1987, pp. 62-75

[Kim et al. 2002] J. Kim, A. Ong, R. Overill. “Design of an Artificial Immune System as a Novel Anomaly Detector for Combating Financial Fraud in the Retail Sector,” Department of Computer Science, King’s College London, UK, 2002

[Kuklas et al. 2005]	C. Kuklas, D. Koschützki, F. Schreiber. „Graph Pattern Analysis with PatternGravisto," Institute of Plant Genetics and Crop Plant Research, Gatersleben, Germany, 2005
[Mannila et al. 1997]	H. Mannila, H. Toivonen, A. Verkamo. „Discovery of Frequent Episodes in Event Sequences," Department of Computer Science, University of Helsinki, Finnland, 1997
[Pei at al. 2002]	J. Pei, J. Han, W. Wang. "Mining Sequential Patterns with Constraints in Large Databases," State University of New York at Buffalo, USA, 2002
[Phua et al. 2005]	C. Phua, V. Lee, K. Smith-Miles, R. Gayler. "A comprehensive survey of data mining based fraud detection research," Artificial Intelligence Review, 2005
[Rode 2005]	B. Rode. "Towards a Model of Pattern Recovery in Relational Data," Cycorb, Austin, Texas, USA, 2005
[Rome 2002]	J. Rome. " Introduction to Spatio-Temporal Pattern Recognition," 2002
[Srikant et al. 1996]	R. Srikant, R. Agrawal. „Mining Sequential Patterns: Generalizations and Performance Improvements," IBM Almaden Research Center, San Jose, CA, USA, 1996
[Tuyls et al. 2000]	K. Tuyls, S. Maes, B. Vanschoenwinkel. „Machine Learning Techniques for Fraud detection," Master Thesis, Vrije Universiteit Brussel, Belgium, 2000
[Valiente et al. 1997]	G. Valiente, C. Martínez. „An algorithm for graph pattern-matching," Technical University of Catalonia, Barcelona, Catalonia, Spain, 1997
[Varró et al. 2006]	G. Varró, D. Varró, A, Schürr. "Incremental Graph Pattern Matching: Data Structures and Initial Experiments," Department of Science and Information Theory, Budapest University of Technology and Economics, Hungary, 2006

Appendix A – Implementations

A.A GraphSlider

A.A.A Introduction to the classes and packages

The GraphSlider is a program implemented in the SUN Java programming language. This implies an object oriented approach. To cover this, the different classes and their functions as well as the corresponding packages will be explained here.

As in every Java program, the anchor lies in the Main class in the package main. The Main class is quite small. All it does is instantiate the class ChainFinderAll of the package algo and than goes into a while loop until a certain condition is met. As soon as the while loop is left, the program finishes. Every method invocation in the Main class is passed to the ChainFinderAll object, so there are no other methods implemented in the Main class. It is only responsible for the program flow.

The class ChainFinderAll is a very special class. As noted before, it can be found in the algo package. It is the only real class in this package. The other five classes are simple interfaces, used for the implementation of a strategy pattern. We will come to that shortly. The ChainFinderAll is not defined as an interface, but its behaviour is quite similar. All the method calls from the Main class are not really handled here. They are only passed through to various other classes. So ChainFinderAll is some sort of a distributor.

One of the most important packages is conf with its class Configuration. Configuration contains no methods. Its purpose is to store different constants. In it, the database connection and the corresponding drivers can be set. It also contains some constants for thresholds, SQL queries, general parameters and the ability to turn the graphical representations of the different graphs on or off.

In the package db, the class DBConnector is resided. It contains all the necessary methods to establish a connection to the transaction database with all the financial transactions to be monitored.

In the package search, one of the most important classes can be found. YSearchGraph contains all the important methods for the program except the pattern matching algorithms for scoring. This class handles most of the program flow as most of the invocation calls caused by the Main class and distributed by ChainFinderAll finally invoke a method of YSearchGraph.

The package alert contains two classes. Alert is, as the name states, the class who stores all the alerts invoked by the fraud detection mechanism. In fact, every alert is an object generated out of this class. Alert has also the ability, to store itself in a database for further processing. The second class in the package alert is YAlertGraph. It handles all the functions on the alert side of the graph, like adding edges and nodes for graphs that caused an alert.

One of the packages not that important is vis. It contains the two classes YAlertGraphVisualizer and YSearchGraphVisualizer. These two classes are responsible for the visualization of the two different graphs. They only contain the graphical information and do not have any computational algorithms, important for the detection of fraud.

The last package in the whole program is chain. It contains the classes with the different algorithms to detect fraud. They have some special purposes. ChainScoreAndSpread is responsible for the scoring and spreading algorithms. It contains the strategies on how the fraudulent behaviour is scored and how the scoring is divided to the different bank accounts involved. ChainAlertAndAge is responsible for the firing of alerts, if necessary. It also is responsible for the ageing of the score on the different accounts. These two classes heavily rely on the other classes in chain. The other classes are the actual strategies and it is possible to switch between them to achieve other distributions or other scorings and ageings.

With this, we have completed the introduction to the different classes and packages and can go on with the actual description of their functionality.

A.A.B Functionality of the GraphSlider

First of all, we have to declare, that the GraphSlider heavily relies on the y-Files for Java. With them it is relatively easy to manage different sorts of graphs as they provide a lot of methods for the edges and nodes. This can be the ability to find all adjacent nodes in an undirected graph or the successor or predecessor nodes in a directed graph. They also provide different functions to cover the graphical representation of the graphs with different layouts and other properties.

The main functionality can be explained as follows. At the beginning, the start of the sliding window is similar to the START_DATE defined in the Configuration class. The SLIDING_WINDOW_SIZE is also given in the Configuration class and can be adjusted if needed. At the start of the program, every transaction in the sliding window is loaded into the program. With these transactions, the graph in the SearchGraph area is built, with transactions being edges and the involved accounts being nodes.

As soon as this first chunk of data is loaded, the *while* loop comes into account. It checks, if the sliding window already has reached the date defined as END_DATE. If the two dates do not match, the program enters the loop and invokes calculateAndSpread(). This function call is passed through to the class defined as scoringAndSpreadingBehaviour.

This is one of the major advantages in the design of the GraphSlider. In order to achieve an easy adjustable program, the strategies for scoring or spreading of the scores, the way an alert is invoked and how the alert-scores are aged, are all implemented in a strategy pattern. This means, the functions are not programmed against the actual classes, they are programmed against interfaces. As interfaces define certain function signatures and all classes that implement an interface must also implement these functions, the invoking class never has to be changed, even if the class behind the interface is a completely different one than before. This is called a strategy pattern and with it, the GraphSlider gets very flexible.

So at the moment, the class behind scoringAndSpreadingBehaviour is ChainScoreSpread. This class goes through all nodes in the actual graph and checks, if they are actually connected to other nodes. If they are not, they are removed from the graph, as there is no longer a transaction in the actual sliding window coming from or leading to that node and therefore it is not relevant for the following calculation.

The main calculation takes place in the class ChainScoring01. This class also can be replaced by another scoring algorithm. We will use that fact in the extended GraphSlider. In ChainScoring01, the incoming and outgoing edges of each node are collected. If a node only has incoming edges or outgoing edges, nothing happens. If a node has both types of edges, the algorithm looks for the dates of the transactions corresponding to the edges. If they are both within the dates defined by the actual position of sliding window, they are marked for further processing. In this, the amounts of the two transactions are compared with the fuzzyEquals() method of YSearchGraph, which returns a boolean. This method compares the two amounts of the transactions with each other. If they are within a certain range (standard is ±10% of the incoming amount), defined in the Configuration class, the method returns a true and the scoring function enters its last processing state. This contains the comparison of the source and the target of the edges. If they are similar i.e. the money is transferred from account A to account B and then back to A, there will be no scoring. Otherwise, a scoring is applied, equal to the amount of incoming edges meeting the conditions times the amount of outgoing edges.

After the calculation for every node is done, the scores are spread among the adjacent nodes, as defined in the spreadingBehaviour. At the time being, the spreading is set to NoSpreading, so no spreading takes place and the scorings do not change.

With the scoring and spreading done, the main method enters its next state, the alarmAndAge state. Here, another important part of the fraud detection process takes place. As with calculateAndSpread, alarmAndAge is passed to an interface with many possible alerting behaviours behind it. At the moment, this is the class ChainAlertAndAge in the chain package.

The first thing done is the summing up of the newly computed score with the past scorings for every node (if there are any past scorings). Then, the triggerAlert() function is invoked. This is the most important function and it is responsible if a node is marked as suspected for fraud or not. The key lays in the CHAINSCORE_THRESHOLD defined in the Configuration class. It is important to adjust it to the proper value. At the moment, it is set to 1, which means, as soon as a scoring occurs (because 1 is the minimum scoring that can occur) an alert is triggered. This is because as soon as the summed up scorings reach the defined value or exceed it, an alert is fired. The alert causes the program to create a new instance of the Alert class, which stores the node that fired the alert, its scoring and its type (in this case it is of the type “chain”).

The invocation of an alert leads to the creation of a new graph, this time in the YAlertGraph. This new graph consists of the node, that caused the alert, plus its adjacent nodes. So the graph has a minimum of three nodes.

After the alert is indicated and stored, the ageing function comes into account. As with the other functions in the chain package, it also implements an interface and is part of a strategy pattern. So it can be easily replaced by another ageing mechanism. In its original form, the ageing function is quite simple. It takes the scoring of each node in the YSearchGraph and divides it by two. This way, it prevents a node with a scoring of 1 to trigger an alert again in the next iteration of the *while* loop in the main class. Nodes with a scoring higher than 1 will trigger another alert in the next iteration and the scoring will be added to the actual alertScore. If the scoring of a node drops below a certain amount (0.1 at the moment) the score is set to 0 to prevent an ongoing ageing of this node, since it can not trigger an alert with such a small scoring. This is mainly to save computation time and resources.

Now, the program is nearly done. It moves on in its main method to the last function invoked. This is moveSlidingWindow(). As simple as this function may seems, there is more behind it. Like nearly all of the flow-control methods, this one is implemented in the YSearchGraph. The first thing it does is obvious. It moves the sliding window forward. In the Configuration class, it can be defined, how many days forward the window will be moved. It is possible to move the window more than it actually covers (through the WINDOW_SIZE) and to leave some of the transactions unprocessed. Therefore, the STEP_SIZE should always be smaller than the WINDOW_SIZE. At the time being, STEP_SIZE is set to 1, so the sliding window moves one day forward, after every iteration in the loop.

The next thing moveSlidingWindow() does, is to get rid of transactions that are no longer covered by the sliding window. This deletes the corresponding edges from the graph, what can lead to nodes having no neighbour anymore. These nodes will then be deleted in the next iteration of the *while* loop.

After this is all done, the method loadDataBetween(), which was already invoked at the very beginning of the program, is invoked again. It gets the data that is in the now moved sliding window.

After this last function, the *while* loop will start anew until the sliding window reaches the date defined in END_DATE. As soon as the *while* loop is left, the whole program terminates.

A.B Extended GraphSlider

A.B.A Implementation of the extended GraphSlider

In this subsection, the composition of the implementation of the extended GraphSlider will be explained. In order to do that, some special new or heavily modified classes are picked out to show their actual function and the design decisions behind them.

A.B.A.A Class Timer

As the extended GraphSlider still should be optimized in terms of memory usage and computation time, a possibility to measure these two values had to be one of the first things to be implemented. For the memory usage, Java itself already delivers a capable tool, the JConsole.

For the measurement of computation time, the class Timer in the package timer was introduced. At the beginning of the program, in the Main class, a new object of the type Timer will be instantiated. Timer requires a long in its constructor, encoding the actual time in milliseconds. This can be achieved in passing the value of Calendar.getInstance.getTimeInMillis() to the constructor. The Timer object stores this value in its start variable and does nothing more with it. Should it be necessary, Timer has the method setNewStart(long start) to pass a new value to the object, but normally, this method should not be used.

At the end of all computations in the Main class i.e. after the still present *while* loop and other computations, introduced later, the Timer object is addressed again. First the method setEnd(long end) is called, again with the actual time encoded as milliseconds. This value is stored in the variable end.

As soon as start and end are stored, the most important method of Timer is called. calculateTime() is the main function of this class. First, it only computes the difference in milliseconds between the start value and the end value. This new value, called diff, could already be printed out, but it would be a large number with not much significance, as people normally don't calculate with milliseconds. Therefore, the calculateTime() method partitions the value of diff into its specific values of hours, minutes, seconds and milliseconds. These four values are combined into one string and than printed out to the console. With this new value, it is possible to compare different runs of the program on different computers, with different values of the Configuration class or with different algorithms for fraud detection.

A.B.A.B Class Account

The name of this class may be mistaken as a class responsible for the bank accounts of the customers. In fact, this class refers to the accounts of the employees in the transaction system. For internal fraud, it is important to monitor the transactions of a single employee. The GraphSlider was not able to differ between the transactions initiated by different employees. Therefore this new class was introduced.

The class Account has two class variables. The variable accNumber stores the ID of the employee, which is attached to every transaction in the database. The accSum variable stores the final scoring of this account by summing up all the scorings of the nodes present in the AlertGraph.

Besides the getter and setter methods for the two class variables, Account has one important method to store its values into a database. The method saveToDB() is called at the end of each turn with each registrar and saves the values of the two class variables into a database

A.B.A.C Class Main (modified)

To be able to account for internal fraud, the main() method of the Main class had to be modified. The *while* loop of the old GraphSlider, which loops through all the transactions from the START_DATE to the END_DATE, was not appropriate, as it did not account for the different registrars. So this *while* loop (hereafter called inner *while* loop) was enveloped into another *while* loop (hereafter called outer *while* loop). Before this new outer loop starts, the method getRegistrators() in YSearchGraph is called. This method returns a vector with an Account object for every registrar-number found in the database. So if there are for example the registrars 1 to 10, a vector with 10 Account objects is returned. The outer *while* loop checks, if this returned vector is not empty. Should there be no registrars at all (which implies, that there are no transactions) the program will terminate at once. If there is at least one Account object in the vector, the object at index 0 is taken.

The program then goes through the inner *while* loop, as it did in the original version of the program. The main difference is, that initWindow(acc) and moveSlidingWindow(acc) are modified so that only transactions are loaded into the program, whose registrar ID in the database is exactly the same as the one stored in the accNumber variable of the actual Account object.

As the inner *while* loop in the Main class reaches the date defined in END_DATE, this time the program does not terminate. First, there are some final calculations explained in the section of SearchForPatterns. Then, the function sumUpAlerts(acc) is called. This function leads to all the scorings being present in the nodes of the AlertGraph at this moment being summed up and stored in the accSum variable of the actual Account object. Then, the saveToDB() function is invoked and the contents of the Account object are stored into a database. After this, the first element in the vector with all the Account objects is deleted, which leads to a new element with index 0. If there are still elements in the vector after this deletion, the outer *while* loop is accessed again and the whole program starts anew. Before entering the inner *while* loop it only has to reset the sliding window back to the value of START_DATE. If there are no more elements in the vector after the deletion, the program terminates.

A.B.A.D Class ChainScoring02

To account for the new requirements imposed through the internal fraud detection, the old scoring mechanism had to be replaced. As of the modular architecture of the GraphSlider, this could easily be done by introducing a new ChainScoring class. It only had to be replaced in the strategy pattern of the class ChainScoreSpread. As every ChainScoring class must implement the ScoringFunction interface, two methods must be inherited. There methods are score(Node n) and the newly introduced method resetCollections(). The first method is responsible for the actual scoring of the different nodes. It is the most important function and contains all the actual scoring information. The resetCollection() function had to be introduced as there is more than one pass through all dates, as the sliding window is reset with every registrar. If the resetCollections() function would not be there, results from the last passes would interfere with the actual computation, what would lead to completely false results.

The problem here is, there is only one method for the scoring, but we have two different situations when something has to raise the score, as described in the concept and approach subsection. As we want to be able to switch between the two different scoring mechanisms without changing the whole class in the strategy patterns, some sort of switch had to be implemented.
With this switch in the score(Node n) function of the ChainScoring02 class it is possible to chose, which of the two new scoring mechanisms should be used via two booleans in the Configuration class. These two mechanisms are periodicScore(Node n) and sequenceScore(Node n) and will be explained en detail in the next sections.

A.B.A.E sequenceScore(Node n)

This method is the solution of the so called smurfing problem. Smurfing means, that a large transaction is divided into a lot of smaller transactions. As a matter of simplicity, we assumed that these transactions are between the same two accounts and in a short, to be defined, period of time. This period is given by the sliding window and can be, if necessary, adjusted to cover all transactions between START_DATE and END_DATE.
The function score(Node n) is invoked by ChainScoreSpread for every node actually loaded in the YSearchGraph. ChainScoreSpread iterates through all the nodes and passes them to score(Node n) and with this to the contained function sequenceScore(Node n). The sequenceScore(Node n) function takes the committed node and tries to get a List of all the outgoing transactions of this node, that are within the sliding window (in the standard setting the past four days).
It then iterates through this list of edges. The edges are then saved in a HashSet to be able to access them and compare them with other edges, as explained later in this section. In three HashMap objects, every edge gets it source-node (sources), its target-node (targets) and its transaction-IDs (trxIds) saved together, with the edge object as its respective key. As soon as this is done, the new edge is compared to all edges that are already stored in the collection *edges*.
If source-node and target-node are the same, there is a similar edge still in the collection. This means that there is already a transaction between these two bank accounts within the defined sliding window. As it could be possible, that a finding is a self-reference, the transaction-IDs are compared, too. If they are the same, it is a self-reference and nothing happens, as the algorithm only has found the same transaction it had stored in the collection a few moments ago. With this mechanism, self-reference is prevented. But this also leads to the prevention of single transactions triggering a scoring. However, if the transaction-IDs are not the same, the

algorithm has found a different transaction within the sliding window and between the same bank accounts. If that should be the case, a last test is executed. As the big transactions are normally divided into quite small ones as implied by the term “smurfing”, the algorithm should only respond to amounts transferred, that do not exceed a certain user defined threshold, the AMOUNT_LIMIT.

If the found transaction meets all the conditions and is below the threshold, a scoring is applied. This scoring is 1.0 times a user defined factor, called SEQUENCE_WEIGHT_FACTOR. With it, the user of the fraud detection system is able to define, how many similar transactions have to be within the sliding window to accumulate a scoring high enough to exceed the threshold and trigger the alert. This is another mechanism to prevent single transactions from triggering an alert. As soon as all the transactions from one node within the sliding window are handled and compared, the sequenceScore(Node n) function is terminated and the weighting function is handled next.

A.B.A.F periodicScoring(Node n)

The function should compare different transactions in the meaning of their periodic appearance, in other words, if they are on a regular base.

The first thing this function does, is to get all the outgoing transactions or edges of the node n within the sliding window and store them in a List. This is quite similar to what sequenceScoring(n) does. Then, it iterates through all the edges fetched. The very first edge only gets stored in the HashSet *edges2*. It is not named the same as the HashSet of sequenceScoring(n) as it is possible to run both scoring algorithms at the same time (although this is not advised) and the two collections should not interfere with each other.

As soon as the second edge gets processed, the method calculateNodeScores(Edge e) is invoked. This method was outsourced to make the code better readable and easier to maintain. Its main purpose is to fork the program flow. It first invokes the method compareEdges(Edge e), which compares the new edge with all edges still in the *edge2* HashSet. If there is an edge with the same source and target-node as the new edge, the function compareEdges(Edge e) returns true, else it returns false. If a false is returned, calculateNodeScores(Edge e) does nothing more than to store the new edge in the *edges2* HashMap and then returns to periodicScore(Node n)

But if compareEdges(Edge e) returns true, the two main scoring functions are invoked. First, calculateDifference(Edge e) calculates, as its name says, the temporal difference between the first edge and the actual edge with the same source and target nodes. It then stores this difference together with the newer node in the HashMap diffs. It also stores the newer node together with its amount transferred in the HashMap amounts. This is important as the amounts in periodic transactions should not differ too much and the HashMap amounts will be use to ensure this.

As soon as this is done, the main scoring function comes into play. It is called scoreDifference(Edge e) as the scoring is mainly based on the difference in the transaction dates or the amounts transferred. The method gets the difference between the actual transaction and the last one and, if possible, the stored difference between the last transaction and the one before that. These two differences are then compared by subtracting the one from the other and taking the absolute value of the result. Then, the modulus of this result compared with the stored difference is taken. This modulo calculation is done for the following reason. Assumed, someone pays its cheques on a monthly base and one month, there is nothing to pay. The modulus ensures that, if the difference between the transactions is normally 30 days and one time it is 60 days, this is still considered as a regular transaction. If the modulus is greater than a user defined threshold called DIFF_LIMIT, a scoring is applied. Normally, the greater the deviation between the differences is, the more irregular the transactions are. This is accounted for by making the scoring dependent on this deviation. The absolute deviation multiplied with a user defined weighting factor called PERIODIC_WEIGHT_FACTOR results in the final scoring for the accounts involved in these transactions.

If the deviation is smaller than the DIFF_LIMIT, what means that the transactions are considered to be regular, the amounts of the actual and the last transaction are compared. If they are within a certain limit, which also is user defined and pre-set to ±15% of the last transferred amount, nothing happens and the transactions are considered to be regular and unsuspicious. If the limits are exceeded, this transaction is suspicious and a scoring has to be applied. This is done by calculating the absolute difference between the two transferred amounts, weighting it with a user defined factor PERIODIC_AMOUNT_WEIGHT_FACTOR and storing the result as score for the involved bank accounts.

If one of the two possible scorings occurs, the last saved transaction together with the stored difference and the amount, is replaced by the newer one. As soon as the whole process described above is done to all edges of the node passed by score(Node n), the algorithm is terminated and the program returns to the ChainScoreSpread class where it fetches the next node in the YSearchGraph and continues the calculation.

A.B.A.G Class SearchForPatterns

As soon as the inner *while* loop in the Main class is left, the whole AlertGraph with all the suspicious nodes which have a scoring above the threshold has been built in YAlertGraph. As these nodes only mark the suspicious bank accounts, but it is not yet sure, which are the real fraudulent ones, a new algorithm has to be applied, capable of finding fraudulent patterns in this AlertGraph. This could be done by an external program or by another extension of the GraphSlider. In order to enable this, all the components of the AlertGraph have to be stored outside of the program in a persistent manner. SearchForPatterns should provide this. The name is a bit misleading, as is first was intended to do the actual pattern search. As there was not enough time anymore, it was decided to only enable other programs to do the actual search, but the name was kept.

When the inner *while* loop is left, the function getSuspiciousPatterns(Account acc) is invoked. This is passed through to the YAlertGraph. It has a method with the same signature. In this method, a new instance of the class SearchForPatterns is created. This class has quite a big signature, as most of the collection classes that store elements of the YAlertGraph have to be handed over to this new class for its calculations.

The first thing, SearchForPattern does, is to invoke its own method createMappings(). This method takes all nodes in the YAlertGraph i.e. all suspicious bank accounts. For every node, it creates a List of all its direct successors and its direct predecessors. This is important to be able to build the connected components. Nodes that have an empty list of predecessors are considered as root nodes for the following deep first search.

As soon as all the available nodes are handled, the function createComponents() is invoked. This is in fact a deep first search algorithm. It takes the nodes marked as root nodes and starts its deep first search. Every node connected directly or indirectly to one of the root nodes is considered to be of the same component. There is a special class called GraphComponents(int compNR), which stores all the connected nodes. For every root node, a new instance of GraphComponents(int compNR) is created. It contains the number of the actual component and the IDs of all the connected nodes of this component. It provides three important functions. With getComponentNodes() a caller is able to receive a Set of all nodes contained in this component. With getNodeCount() it is possible to get the amount of nodes stored in this component and with getDensity() it is possible to get the density of this GraphComponent , that means, how many different transactions there are between the nodes of this component.

As soon as all the GraphComponents are built, the function saveToDB() of SearchForPatterns is invoked, which leads to all the data saved in the GraphComponent objects to be saved to a database for further processing.

As soon as this is done, the main() method in the Main class removes the actual account from its vector and restarts the outer *while* loop, if there are more accounts available. If the vector is empty, the performance measure method of the class Timer is invoked and thereafter, the extended GraphSlider terminates.