UHASSELT

KNOWLEDGE IN ACTION

Faculteit Bedrijfseconomische Wetenschappen

master in de toegepaste economische wetenschappen: handelsingenieur in de beleidsinformatica

Masterthesis

An exploration to what extent process mining techniques can be applied to extract a collaboration model from version control system logs

Leen Jooken

Scriptie ingediend tot het behalen van de graad van master in de toegepaste economische wetenschappen: handelsingenieur in de beleidsinformatica

PROMOTOR :

Prof. dr. Mieke JANS

BEGELEIDER :

De heer Mathijs CREEMERS

2018
2019

UHASSELT

KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

UHASSELT

KNOWLEDGE IN ACTION

Faculteit Bedrijfseconomische Wetenschappen

master in de toegepaste economische wetenschappen: handelsingenieur in de beleidsinformatica

Masterthesis

An exploration to what extent process mining techniques can be applied to extract a collaboration model from version control system logs

Leen Jooken

Scriptie ingediend tot het behalen van de graad van master in de toegepaste economische wetenschappen: handelsingenieur in de beleidsinformatica

PROMOTOR :

Prof. dr. Mieke JANS

BEGELEIDER :

De heer Mathijs CREEMERS

An exploration to what extent process mining techniques can be applied to extract a collaboration model from version control system logs

JOOKEN LEEN, Hasselt University

ABSTRACT

A precise overview on how software developers collaborate on code could reveal new insights such as indispensable resources, potential risks and better team awareness. One might have an idea about how these developers work together, but this will often deviate from and be less nuanced than reality. Version control system logs keep track of what team members worked on and when exactly this work took place. Since it is possible to derive collaborations from this information, these logs form a valid data source to extract this overview from. This concept shows many similarities with how process mining techniques can extract process models from execution logs. The fuzzy mining algorithm [8] in particular holds many useful ideas and metrics that can also be applied to our problem case. This paper describes the development of a tool that extracts a collaboration graph from a version control system log. It explores to what extend fuzzy mining techniques can be incorporated to construct and simplify the visualization. A demonstration of the tool on a real-life event log is given. The paper concludes with a discussion of the business relevance.

ACM Reference format:
Jooken Leen. 2019. An exploration to what extent process mining techniques can be applied to extract a collaboration model from version control system logs. 1, 1, Article 1 (May 2019), 15 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Companies engaged in software development have to manage a great deal of code on a daily basis. In order to do so, they work with several teams of programmers that focus on specific parts of a project. However, the more programmers a company employs and the more projects are actively running at the same time, the easier it is to lose the overview. It gets hard to keep track of who has knowledge of certain aspects of the code and which programmers are working together on which parts. This lack of awareness can also cause files that are at risk of becoming unknown to any programmer, to go unnoticed. This is the case when you have a non-static file that only gets changed by a very small amount of people. If this select group of programmers were to quit, the organization can end up with code that nobody really knows the details of.

An effective way to improve this awareness is through visualization of these relationships, which can be achieved through a social network graph that represents programmers as nodes and relationships between them as edges. In this way, we can easily identify clusters of programmers that are working on similar things, isolated nodes that should be paid special attention to and cross-functional team members, to name some examples of insights that might be gained. Since we wish to discover this collaboration from real-life project scenarios, we believe that logs drawn from version control systems serve as an ideal primary data source. These systems are collection points of many different kinds of information that could possibly be exploited in a way that benefits the business. Some examples of data it holds are: the different code files that make up the project, the evolution and other time related aspects of the code, commit messages and programmer's activity.

This idea of extracting a social network graph from a version control system log strongly resembles the rationale of applying process discovery to process event logs. Process discovery is a type of process mining. Process mining focuses on extracting knowledge and insights from process data, that is collected in event logs. An event log is a log book of events which occur in the context of a process. All events related to a common case make up one process execution. There exist four different perspectives within process mining, with the organizational perspective being the one that is most closely related to our problem case. This perspective includes techniques to learn more about organizational roles, people and work patterns. One of these techniques is social network analysis, which uses the event log to build a weighted social graph, with the weight of a node or edge as indicator of its importance. To carry out this organizational mining, the resource attribute of an event is the main focus [19]. Van der Aalst et al. identified four types of metrics that can be used to establish relationships between individuals in an event log, which leads to various types of social networks. Those four types of metrics are: metrics based on causality, metrics based on joint cases, metrics based on joint activities and metrics based on special event types [20].

This will be the starting point for our visualization, but with a different kind of data input and a different goal. Instead of event logs, we will be using logs drawn from version control systems. The two are similar in the way that every commit message from the latter can be seen as an event, and both specify a resource for every event. In our case the resource is the programmer that carried out the commit. Aside from these similarities, there are also some aspects in which the two differ. In an event log the events follow a certain process flow, while in the case of a version control system there is no clear process notion involved. The act of developing software might be better described as a project, rather than a process, because it lacks the control that a central process engine provides. This makes it difficult to define an exact process flow [1]. This lack of a clear process notion is also the reason why we do not build on the implementation by van der Aalst et al. [20]. The visualization that we aim for does not fit within the four types of metrics, that were mentioned in the previous paragraph:

- *Metrics based on causality:*
  There is no real notion of causality, as work is not handed over but rather collaborated on by multiple people at the same or different times.
- *Metrics based on joint cases:*
  These metrics count how frequently two individuals perform activities for the same case. Since in our adaptation a case refers to a project and a version control system log concerns one project in particular, all individuals are performing activities for the same case.
- *Metrics based on joint activities:*
  These metrics focus on the activities a resource performs: the more similar the activities, the stronger the relationship between resources. If we relate activities to files in our adaption, this idea holds up. There are some issues, however, that render the exact metrics used by van der Aalst et al. useless for our implementation. First of all, we want relationships between programmers only when they are working on the exact same files. After all, there is no way of knowing with certainty whether different files are related. Secondly, the metrics used by van der Aalst et al. are only suitable if the resources perform comparable volumes of work. This is not necessarily the case for the programmers in the version control system log, as the number of files they work on can vary significantly.
- *Metrics based on special event types:*
  There is no notion of an event type for commits in a version control system log.

In this work, we aim to develop a tool that takes a version control system log of a project as input and produces a visualization, in the form of a graph, that shows how the programmers of that project collaborated. This visualization should make it possible to easily identify the core collaboration teams and isolated nodes, as these are the most valuable insights for an organization. To enable this, the visualization has to be clear and easy to understand. This shows similarities with another aspect within the process mining domain, namely fuzzy mining [8]. The fuzzy miner was the first process discovery technique that envisioned a comparable visual representation. Its goal is to discover process models from event logs of unstructured processes. This comes down to using event log data to produce a visualization of the links between these events. Its approach is based on a roadmap metaphor, which abstracts undesired details by making use of four different concepts: aggregation, abstraction, emphasis and customization [8]. The goal of our research is using log data to build a visualization of the interaction between people. To achieve this, we will examine to what extend the techniques from fuzzy mining are also applicable to our visualization. We start by researching how we can extract the collaboration structure and build the graph inspired by the fuzzy miner algorithm. Next, we will also examine the metrics used by the fuzzy miner for weight calculation. The fuzzy miner uses a notion of node and edge importance for its simplification approach. It bases this importance on two fundamental metrics: significance and correlation [8]. We will explore to what extend our calculation of the importance of programmers and their relationships can be inspired by the fuzzy miner as well.

Contributions. This paper describes a way to extract a collaboration graph from a version control system log, based on techniques used in process mining. It discusses the graph's design choices, explores in what way techniques from fuzzy mining can also be applied to this new problem domain, and lastly proposes several insights that can be gathered from the tool that could benefit the business.

Overview. The remainder of this paper is organized as follows. Section 2 discusses the state of the art. We then continue by elaborating the design choices for a new tool in section 3. This is followed by a detailed description of the tool's algorithms: section 4 explains the algorithms used to calculate the importance of collaborators and their relationships; while section 5 focuses on the approach used to simplify the visualization. Section 6 holds a demonstration of our tool on a real-life example and we conclude this paper with a discussion in section 7.

2 RELATED WORK

Most of the literature covering potential uses of version control system logs focuses on using the files stored in such a system to visualize the source code hierarchy, rather than collaboration aspects [5–7, 12, 14, 15, 17, 22, 24]. The benefits of code visualization have been widely explored in an abundance of studies [5–7, 12, 14, 15, 17, 22] and come down to the following:

- Improving a programmer's understanding of the software
- Facilitate feature localization
- Support software maintenance
- Speed up bug tracking
- Assist with new developer training
- Serve as a base for design quality assessment

These source code visualizations come in various shapes and sizes such as: flowcharts (Moritz [6], Crystal Flow [6]), activity charts (Flowgen [6], Kayrebt [6]), graph representations (SHriMP [16] [17], Rigi [17], Imagix 4D [17], CARE [17], VIFOR [17], Hy+ [17], PECAN [17], Whorf [17], NV3D system [7]), control structure diagrams (GRASP [17], CSD [12]), virtual environments (ImsoVision [7], Software World [7], SOLVEN [7]) and many more.

In contrast to this numerous amount of tools that deal with code structure and flow, there are very little - to almost no- tools that make use of data related to social aspects. This is accompanied by a notable lack of research regarding the benefits of using this kind of data to gain new insights.

A tool that does involve this type of data up to a certain level, is the Manhattan tool, developed by Lanza et al. [11]. This tool produces a 3D visualization of the code, based on a city metaphor first used in CodeCity [23]. However, its main goal is supporting team activity by increasing workspace awareness. This means that the tool tries to make a programmer more aware of what his team members are working on and whether it could potentially impact his own work. In order to achieve this, the system visualizes in real-time what each team member is working on, notifies the user about

emerging conflicts, shows classes that have been changed or deleted by others, and developers who modified a class more frequently. This results in a more efficient and preventative way of working than the current approach, which consists of raising awareness and conflict solving through human interactions like meetings and informal conversations [11]. Although the tool makes use of collaboration data, it does not explicitly visualize these relationships between programmers. There is no functionality to query the system about how often and how closely members work together.

3 DESIGN OF THE COLLABORATION VISUALIZER

3.1 Choice of data source

There are several reasons why we chose version control system logs as the data source to extract collaboration insights from:

(1) These logs collect a lot of data that is useful and necessary to study how people collaborate on certain projects. They include the files that make up the project, the different programmers that worked on them, timing aspects, pair programming and the way the project evolved over time.
(2) Most (if not all) companies that are involved in software development make use of a version control system.
(3) There is no need to implement a new system to start collecting data. The current and past logs can immediately be analyzed, which also means that testing the tool can be done instantaneously.

For this first study we will focus only on logs produced by the Tortoise SVN system.

3.2 Solution requirements

We want to construct a visualization of the collaboration relationships between programmers, in the form of a graph. In order to gain valuable insights about the social structure of a development team, we set the following requirements for our solution:

(1) It is able to read and process the data from a version control system log.
(2) It uses the aforementioned data to construct a graph, in which programmers are represented by nodes and the collaboration between them by edges.
(3) It renders a visualization of this graph.
(4) Both the programmers and the edges between them have a weight that reflects their importance.
(5) This weight is visualized by respectively the size of the node and the thickness of the edge.
(6) The graph is clear and easy to understand, which implies that some simplification measures have to be taken.
(7) The graph is presented in an aesthetically pleasant way, with as few overlapping nodes and edges as possible.
(8) Multiple colours are available to distinguish between different kinds of nodes and edges. A distinction is made between regular nodes and nodes that represent clusters. Further, there are three possible categories an edge can belong to:
    - **Pair programming edge:**
      The two programmers only ever engaged in pair programming and did not work separately on the same files.
    - **Disjunct programming edge:**
      The two programmers only ever worked separately on the same files and never engaged in pair programming.
    - **Pair and disjunct programming edge:**
      The two programmers worked separately, but also engaged in pair programming.
(9) The visualization allows interaction in the form of zooming, panning and drag movements.
(10) The tool works on a Windows platform.

3.3 Choice of tools

First and foremost, we had to carefully select which tools to use in order to build our solution, while making sure that all the requirements stated in subsection 3.2 were met. We started off by collecting information about 23 different social network analysis tools and libraries, and compared them based on main functionality, input format, output format, platform and license cost. After some consideration we decided to use a combination of a self-written Python program and the Gephi tool. Gephi is an Open Source interactive graph exploration and manipulation software, suited for all kinds of networks [2]. By making use of this software, we can focus our efforts on developing an algorithm that builds the graph structure from the log, without also having to encode the visual rendering and user interaction from scratch. Of all possible network visualization tools, Gephi was chosen because it offers functionalities that best meet all of our visualization requirements proposed in subsection 3.2. These functionalities include the following:

- The program is able to read a graph from a wide variety of input formats and visualize it. *(Meets requirement 3)*
- Edges can be given a weight that will automatically alter its thickness. The size of the nodes can be dependent on any user-defined attribute. *(Meets requirement 5)*
- Different kinds of graph drawing algorithms can be used to render the visualization, including Force Atlas [10], Fruchterman Reingold [10] and Yifan Hu (Proportional) [9]. The aforementioned algorithms are all force-directed and specialize in drawing graphs as clear as possible, making sure that the edges are of more or less equal length and that there are as few crossing ones as possible. *(Meets requirement 7)*
- The colours of both the nodes and edges are adjustable and can be set based on the ranking or partition of an attribute. *(Meets requirement 8)*
- Interactive visualization is made possible by offering user interaction. The structures can be manipulated by zooming, panning and drag movements among other things. Shapes and colors can be set to reveal hidden properties and the graph can be interactively filtered on many attributes. *(Meets requirement 9)*
- The program can be used on any system supporting Java 1.6 and OpenGL. *(Meets requirement 10)*

Furthermore, the tool also offers additional functionalities that are valuable:

- The 3D render engine can display large networks in real-time and speed up the exploration. This is necessary to ensure a

decent user experience when analyzing the logs of projects that have many contributors.
- Different statistics can be calculated such as modularity, average clustering coefficient and eigenvector centrality. The results of these statistics can also be used to manipulate the visualization, for example: different node color or size. So can, for example, the number of connected components be calculated and used to colour-code the nodes. This makes it easier to spot isolated nodes and clusters.

With the help of Gephi we can shift our focus to the development of a program that will handle the remaining requirements (1, 2, 4 and 6). We have chosen to write our program in Python and use CSV as the input format for the Gephi visualization.

3.4 Detail of the program structure

The program will comprise of several algorithms that carry out the steps in the following list in order to generate the resulting graph. The calculation of the file importance, the edge weights, the node weights and the simplification of the graph will be explained more into detail in sections 4.1, 4.2, 4.3 and 5 respectively. Calculating an importance value for every file is necessary to get a right understanding of the importance of both the programmers and the collaboration between them. The algorithms that handle the calculation of the weights and the simplification of the graph will to some degree be inspired by the fuzzy mining algorithm [8].

(1) Parse the log into a usable data structure
(2) Calculate the importance of each file
(3) Build the base graph:
    - Include every programmer that is mentioned in the log
    - Add a pair programming edge between two programmers when they have participated in this activity at least one time
    - Add a regular edge between two programmers when they have both worked on at least one file that is the same, and this collaboration was ***not*** in the form of pair programming
(4) Calculate the edge weights:
    weighted sum of the frequency significance and the proximity correlation of the edge
(5) Intermediate clean-up of the edges with a weight equal to 0, necessary to entail a correct calculation of the node weights
(6) Calculate the node weights:
    weighted sum of the frequency significance, betweenness centrality, eigenvector centrality and degree centrality of the node
(7) Simplify the resulting graph:
    - Phase 1 - edge filtering: filter out weak edges
    - Phase 2 - aggregation: form coherent clusters of less significant programmers that have strong relationships between them
    - Phase 3 - abstraction: remove insignificant programmers that are not strongly enough connected to other programmers so they can be aggregated
(8) Write the simplified graph to CSV format, so Gephi can carry out the visualization

4 CONSTRUCTING THE BASE GRAPH: WEIGHT CALCULATION ALGORITHMS

4.1 Calculating the file importance

As already mentioned in section 3.4, the program should calculate a value that reflects the importance of a file. This value is necessary to create a well-substantiated weight for both the programmers and their relationships. Rather than to focus on the business importance of a file, we have chosen to let this value represent how important the file is for collaboration. We cannot base this importance of how large a certain file is, since this information is not standard available in a version control system log. So let us consider a file important if it continues to *'grow over time'*. A project is dynamic, it evolves over time to accommodate for new features and new customer requirements. As a result of this, files that make up the core of the application will get altered regularly in order to support this evolution and will also be the ones that catch the most bugs as they are constantly being changed. Less important files, like configurations or trivial pieces of code, stay static. These files that get changed on a regular basis are good candidates for collaboration from a business point of view. The reason for this is two-fold:

(1) Since these files are constantly being updated, it is good to have multiple people with knowledge of them. This increases the chance that there is always someone around that can help with trouble-shooting and further development. That way, people leaving the company or falling ill will not pose a threat for the further evolution of the code.
(2) Collaboration in the form of pair programming, ensures faster development, decreases the chance of introducing bugs and speeds up bug fixing, since there are multiple people contemplating the same code [4].

Based on these assumptions, we will develop an algorithm to calculate the file importance. To ensure that files that have been around for a long time are not favoured over relatively new ones, we will work with a ratio that takes the life span of the file into account. This ratio considers the number of months the file in question exists or existed, and calculates in how many of those months the file got changed. For example: consider a file that was created in March and changed in April, May and July of that year. If the log data ends in August and the file did not get deleted between July and August, the file importance will be 4/6. A file that existed for 1.5 years and got altered in 12 of those months, also has a file importance of 4/6. We do not factor in the number of commits related to this file that occur within a month. The reason for this is that a programmer is free to choose whether he commits his work in many small pieces or just all at once. A downside to this approach is that files that were created towards the end of the log will in most cases have a larger importance than files that have been around for a long time. There is no way of telling whether these new files will get altered regularly in the future, but we do not consider this larger importance to be a problem since these files are recent and therefore important at this very moment in the developing stage.

Another aspect that must be touched upon before we can elucidate the algorithm, is the fact that a file can be added and deleted multiple times. A file is identified by its path. After deletion of a certain file, it is possible to add that exact file path again. This implies that a file can have multiple time stamps in which it was added to or deleted from the version control system.

This brings us to the following algorithm that calculates the importance value for a specific file:

File importance algorithm

(1) Collect the time stamps:
  (a) Collect the time stamps of the commits where this file was ***'added'*** and sort them chronologically
  (b) Collect the time stamps of the commits where this file was ***'deleted'*** and sort them chronologically
  (c) Collect the time stamps of the commits where this file was altered (this includes both ***'added'*** and ***'modified'***) and sort them chronologically
(2) If the file was added before the start of the log data, take the time stamp of the start of the log as starting point;
  Else take the time stamp of when the file was first added as starting point
(3) Use this starting point to calculate how many months the file existed up until it got deleted. If the file never got deleted, take the last time stamp of the log as ending point
(4) Calculate in how many of those months the file got altered
(5) If the file got deleted within the time span of the log, search for a point where the file got added again. If such a point exists, take this as a new starting point and repeat steps (3) to (5), while accumulating the number of months the file existed and the number of months in which the file got altered
(6) Calculate the file importance as:

$$\frac{the\ number\ of\ months\ in\ which\ the\ file\ got\ altered}{the\ number\ of\ months\ the\ file\ existed}$$

Figure 1 illustrates six possible scenarios on how to determine which time span to take into account to calculate the importance value. It shows a timeline including the start and end point of the log data. The orange indications show the time periods in which the file exists in the log. These periods will be used to calculate the aforementioned ratio.

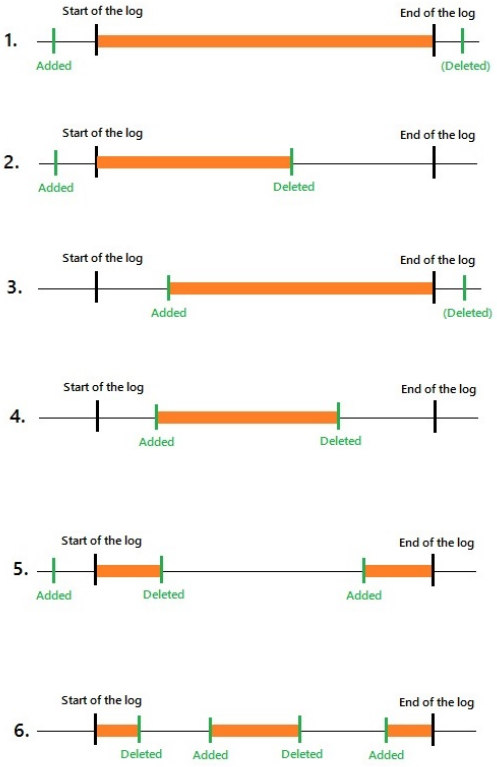

Fig. 1. *Different scenarios that illustrate the possible life spans of a file in the log.*

4.2 Calculating the edge weights

The edge weights are indicators of how strong the collaboration between two programmers is. In order to logically substantiate these weights, we examined and selected some metrics used in fuzzy mining that we believe are also applicable to our problem. Fuzzy mining simplifies a model by using abstraction and aggregation. It decides between these two options based on the significance and correlation value of a node or edge. Several metrics are used to calculate these values [8]. We examined these metrics and decided that two of them could also be used to calculate the edge weights for our problem. These metrics are the ***binary frequency significance*** and the ***proximity correlation***. Both of these contribute a value to different aspects that together determine how important the collaboration is. We will further elucidate these metrics in section 4.2.1 and 4.2.2.

4.2.1 The binary frequency significance.

The first metric we include in the calculation is the ***frequency significance***. Classified under the binary significance metrics, this metric describes the relative importance of the relationship between two nodes. Within fuzzy mining, it is defined as *'the more often two event classes are observed after one another, the more significant their precedence relation'* [8]. We will adapt this idea to our problem

context as *'the more files are worked on together, the stronger the relationship'*. However, there are several issues with this statement, indicating that, in order to apply this metric to our problem case, some constraints have to be added.

First of all, we cannot base the importance of the relationship solely on the number of distinct files they have worked on together, as this does not take into account the file importance. If we implement this metric without any constraints regarding the file importance, two programmers that worked on only a handful of, but very important, files will end up with a weaker relationship than two that worked on a lot of trivial files together. We will solve this problem by making use of the file weights, as explained in section 4.1.

Secondly, we have to deal with the ever-recurring problem of the time between two commits. Two programmers can work together on a file multiple times, but we have to be careful not to favour a programmer with the habit of committing his current work regularly over one that likes to commit very big chunks of code more sparsely.

Lastly, we will only consider files from commits that were not pair programming activities between the two programmers under consideration. The pair programming aspect will be included in the edge weight calculation as a separate factor, further explained in subsection 4.2.2, since this is a special kind of collaboration.

Keeping in mind these three constraints, we suggest the following algorithm to calculate the frequency significance value for a certain edge between two programmers:

Binary frequency significance algorithm

(1) Collect the commit list of the first programmer and that of the second programmer
(2) Filter out all the pair programming commits concerning these two programmers
(3) Using the remaining commits, compose a list of files both programmers worked on
(4) For each of these files, do the following:
  (a) Collect all the time stamps of when the first programmer committed this file; do the same for the second programmer
  (b) Order these two lists of time stamps chronologically
  (c) Count the number of times they have worked on it together, with the constraint that everything that happens within the time span of one week counts as the same *'block'*
  (d) Calculate the frequency significance for this file as:
  $$\textit{the number of blocks calculated in the previous step} \times \textit{the importance of the file under consideration}$$
(5) The final frequency significance of this edge is the sum of the frequency significance of all the aforementioned files

Let us illustrate the principle mentioned in step 4c using the following example:

Suppose we have already collected and ordered the time stamps of when programmer 1 committed a certain file and the same for programmer 2. This gives us the following table:

Ordered time stamps of when programmer 1 committed the file	Ordered time stamps of when programmer 2 committed the file
01/03	02/03
02/03	02/03
03/03	05/05
04/07	06/07
08/07	
19/12	

Table 1. *In this example two blocks are identified where programmer 1 and 2 collaborated on the same file: the yellow and the pink one.*

Table 1 shows the dates on which two programmers worked on a certain file. Only when both programmers edited this file within the same week, it counts as collaboration. Moreover, only when there is more than one week between two consecutive dates it counts as a separate collaboration session. So this example shows two collaboration sessions, namely the yellow and the pink block.

4.2.2 *The proximity correlation.*

The second metric we include is the ***proximity correlation***, which is a binary correlation metric. Within the fuzzy mining algorithm, the proximity correlation evaluates event classes which occur shortly after one another, as highly correlated [8]. In terms of the timing aspect, we have already taken the proximity into account during the calculation of the frequency significance by limiting the permitted time span between two consecutive commits (see subsection 4.2.1). However, in our case there is yet another interpretation of proximity possible, namely the physical proximity in the form of pair programming. This will serve as our proximity correlation, as we have earlier decided to not include this aspect in the frequency significance. There are two reason why we have chosen to include this as a separate factor:

(1) Pair programming is a closer collaboration since the programmers are physically seated next to each other and both contribute to the same code together.
(2) There is more certainty involved that they really interacted and both know the code, than there is when they worked on the same files but not as a pair programming activity.

This division allows for pair programming to have a greater influence on the final edge weight than the frequency significance.

Because this metric has the same constraints as the frequency significance, explained in subsection 4.2.1, we will again work with the file importance and a predefined time span between commits. This results in the following algorithm:

Proximity correlation algorithm

(1) Collect all the pair programming commits concerning these two programmers
(2) Compose a list of files that appear in these commits
(3) For each of these files, do the following:
  (a) Collect the time stamps of all the commits concerning this file
  (b) Order these time stamps chronologically
  (c) Define the time span that must lie between two consecutive commits in order to count as a separate collaboration. The default is one month
  (d) Count the number of times the programmers collaborated by going through the chronologically ordered time stamps and tallying whenever there is at least the predefined time span in between two consecutive time stamps
  (e) Calculate the proximity correlation for this file as:
  the number of times they collaborated, as calculated in step (d)
  x
  the importance of the file under consideration
(4) The final proximity correlation of this edge is the sum of the proximity correlation of all the aforementioned files

4.2.3 *The final edge weight.*

In order to calculate the final weight of an edge, the two aforementioned metrics are normalized and both get assigned a weight that determines how much the metric influences the final edge weight value. As already mentioned before, the proximity correlation value should have a greater influence on the final weight, because the metric concerns a closer collaboration and there is less uncertainty involved. This leads us to the following formula:

Final edge weight formula

Final edge weight
=
0.40 x normalized frequency significance
+
0.60 x normalized proximity correlation

4.3 Calculating the node weights

The weight of a node shows the importance of that programmer within the project of which the logs are under consideration. We explored which metrics used by the fuzzy mining algorithm could be useful for the calculation of this weight, like we did for the edge weights in section 4.2. Only one metric seemed applicable to our problem case, this one being the ***unary frequency significance*** [8]. In order to better substantiate the weight, we looked for additional applicable metrics stemming from traditional graph theory. The three metrics that were eventually selected are the ***betweenness centrality***, the ***eigenvector centrality*** and the ***degree centrality***. These four metrics will be explained more into detail in subsections 4.3.1, 4.3.2, 4.3.3 and 4.3.4. Each of the values they produce will be used to calculate the final node weight in subsection 4.3.5.

4.3.1 *The unary frequency significance.*

The fuzzy mining algorithm evaluates the frequency significance of a node as *'the more often a certain event class was observed in the log, the more significant it is'* [8]. We will adopt this idea as *'the more often a programmer appears in the log, the more significant he is'*, but we will need to set some constraints. Again, we work with a list of distinct files and the file importance. To counter the *'free choice of when to commit'* problem, the distinct list of files makes sure we count every file only once. However, only considering the total number of files a programmer worked on is not nuanced enough. This would for example cause a programmer that worked on five trivial files to have the same significance as one that worked on five very important files. To counter this problem, we will also factor in the file importance.

This leads us to the following algorithm:

Unary frequency significance algorithm

(1) Compose a list of all the files this programmer worked on
(2) Calculate the sum of the importance of each of these files
(3) The frequency significance will be the normalization of this sum. (Since the importance of a file $\in]0, 1]$, we normalize this sum by dividing it by the number of files)

4.3.2 *The betweenness centrality.*

The ***betweenness centrality*** is the second metric we include in the calculation. It is one of the many centrality measures present in graph theory. We chose this specific measure to handle programmers that are a part of several different teams. This metric considers a node highly important if it forms bridges between many other nodes. So this node is an important gatekeeper of information between disparate parts of an organization [3]. To calculate this value we make use of the NetworkX Python library [18], which computes the betweenness centrality of a node as the sum of the fraction of all-pairs shortest paths that pass through that node [18]. We pass the current graph and the edge weights, calculated as explained in Section 4.2, as arguments. Lastly, we normalize the results based on the minimum and maximum centrality value, instead of using the number of nodes. Otherwise the values will be too small to make a real impact on the final weight.

4.3.3 *The eigenvector centrality.*

The next metric that contributes a value to the node weight is the ***eigenvector centrality***. According to this metric, a node is highly important if many other highly important nodes link to it. However, the centrality does not only depend on the number of edges incident on the node, but also on the quality of those edges [21]. So this metric reveals teams of important nodes that work closely together. We implement this metric by making use of the NetworkX library and the edge weights, calculated as explained in section 4.2.

4.3.4 *The degree centrality.*

Lastly, we also include the ***degree centrality***, which looks at the number of edges incident upon the node [21]. We will use this metric to identify (nearly) isolated nodes. These nodes are important in our visualization because they impose a threat. The reasoning behind this is that when a programmer is the only one to work on a file, he also becomes the only one with knowledge of this code and very important for the further evolution of it. This, combined with the frequency significance (i.e. the importance and number of files), can uncover nodes at risk. These risky nodes represent programmers that have contributed many important files to the project, but have written these all by themselves without any collaboration. This makes the programmer very valuable for the company and not easy to replace. So these nodes should have larger weights than other leaf nodes, to prevent them from being pruned during the simplification phase.

At first glace, this seems to contradict the eigenvector centrality, described in the previous subsection. A (nearly) isolated node will have a low eigenvector centrality value, which does not provide it with the importance it needs. However, we cannot use a low eigenvector centrality value as an indicator to give extra weight to a node. This because a node that is connected to many unimportant nodes will also have a low eigenvector centrality and there is no evidence to imply that this node is also important. Furthermore, imposing a larger weight on a node with a low eigenvector centrality will cause the core team cluster nodes to have a low weight, which is not what we want. For this reason we use the eigenvector centrality to identify the core teams and the degree centrality as a separate metric to identify the (nearly) isolated nodes and give them the weight they deserve. The degree centrality only affects these almost isolated nodes, because it only relies on the number of incident edges. A node with only a few neighbours will have an increased importance, while it will not affect the importance of nodes with many neighbours. This is good because having many incident edges does not necessarily imply that a node is important.

To apply the degree centrality metric to our weight calculation, we have to alter the original formula [13], which looks as follows:

$$C_{Di} = \frac{\textit{the number of links incident on node } i}{\textit{the total number of nodes node } i \textit{ can be connected to}} = \frac{\sum_{j=1}^{n} a_{ij}}{n-1} \tag{1}$$

with n being the total number of nodes and a_{ij} the position in the adjacency matrix. The division by $n-1$ is meant to scale the centrality measure between 0 and 1.

Since we need nodes with less incident links to have a larger weight, we alter the formula as follows:

$$\begin{aligned} C^{*}_{Di} &= \frac{\textit{the number of nodes node } i \textit{ is } \textbf{NOT} \textit{ connected to}}{\textit{the total number of nodes node } i \textit{ can be connected to}} \\ &= \frac{n-1-x}{n-1} \\ &= 1 - \frac{x}{n-1} \\ &= 1 - C_{Di} \end{aligned} \tag{2}$$

with n being the total number of nodes and x the number of links incident on node i.

To calculate our final degree centrality value, we make a distinction between pair programming and non-pair programming edges. The reason for this is that the latter involves a lot of uncertainty, while in the case of pair programming we are **certain** that more than one programmer has knowledge of the code. We implement this logic as follows:

Degree centrality algorithm

(1) Calculate the adapted degree centrality for the distinct collaboration edges *(see equation 2)*
(2) Calculate the standard degree centrality for the pair programming edges *(see equation 1)*
(3) Subtract (2) from (1)
(4) The result $\in [-1, 1] \rightarrow$ normalize to $\in [0, 1]$

4.3.5 *The final node weight.*

After the normalization of every metric we can calculate the final node weight by taking the weighted sum. We used the following weights for the aforementioned metrics respectively: 15%, 25%, 30% and 30%. The eigenvector and the degree centrality have the greatest influence because the main goal of our visualization is identifying the core teams and isolated nodes. We also deemed the betweenness centrality more important than the frequency significance because programmers that are a part of different teams form an important insight and this should be emphasized in the visualization. This leads us to the following formula:

Final node weight formula

Final node weight
=
0.15 x normalized frequency significance
+
0.25 x normalized betweenness centrality
+

0.30 x normalized eigenvector centrality
+
0.30 x normalized degree centrality

5 GRAPH SIMPLIFICATION APPROACH

At this point we have constructed the base graph and assigned weights to both the nodes and the edges. However, this graph is too large and too unclear to derive any useful insights from it. This means that in order to be of any value to us, the graph needs to be simplified. This unstructuredness reminded us of the "spaghetti-like" process models that are often the result of carrying out process mining approaches on real-life processes. The fuzzy mining algorithm offers a way to simplify this kind of process models [8]. Since we have already partly based our weight calculation on how the fuzzy miner values its nodes and edges, we will also examine to what extend its simplification process can be applied to our graph.

The fuzzy miner got its inspiration from a road map, which uses four concepts to provide a high-level view: abstraction, aggregation, customization and emphasis (see figure 2).

Fig. 2. *Road map metaphor used by the fuzzy miner [8].*

The miner includes these concepts in its simplification approach as follows: [8]

- **Aggregation** of less significant but highly correlated behaviour
- **Abstraction** of less significant and lowly correlated behaviour
- **Emphasis** by highlighting more significant behaviour
- **Customization** based on the context, level of detail or purpose of the model

These concepts are then incorporated into three transformation methods that take the base graph as input and produce a simplified version of it as output. The first method is ***conflict resolution*** which aims to remove edges from nodes that are in *conflict*. This is the case when two nodes are connected by edges in both directions. Since our collaboration graph is undirected, this phase is not applicable and can be skipped. ***Edge filtering*** is the second method used in the simplification approach. This method filters the edges on a local basis: the edges with the highest utility value will be preserved, after comparison with an edge cutoff parameter. We can apply this concept to our problem case by removing collaboration relations that are very weak. These relations are not well established and therefore involve a lot of uncertainty about whether collaboration really took place. It is best to remove these edges from the graph. The last method comprises ***node aggregation and abstraction***, which preserves highly correlated groups of less significant nodes as clusters and removes isolated, less significant nodes [8]. We can apply this to our graph as follows: programmers that are not very important but have strong collaboration relations between them can be presented as a group; very insignificant programmers that have no strong relations with others can be abstracted. The latter ensures that nodes that our tool mistook for programmers will be removed from the final graph.

Now that we have a basic understanding of how these methods can be applied to our problem case, we can construct the approach we will follow for the simplification of our graph. Our approach consists of the following three phases: (1) edge filtering, (2) aggregation, (3) abstraction. We will elaborate these steps in the following subsections.

5.1 Phase 1: Edge filtering

In this first phase we filter out edges that are weak and therefore not well established. The fuzzy miner [8] tackles its edge filtering by calculating a utility value for every edge, which is the weighted sum of the edge's significance and correlation value. For every node the edges with the highest utility value are then preserved. This is achieved by normalizing the utility values of all the edges incident on the node, and comparing these values with an edge cutoff parameter. We will adapt this algorithm to better fit our problem case.

The most striking difference between our collaboration graph and the graph used by the fuzzy miner [8], is that the former is undirected while the latter is directed. This results in the fuzzy miner [8] carrying out the edge filtering algorithm twice for every node: once for the incoming edges, once for the outgoing ones. Since our graph is undirected, we perform our filtering algorithm only once on the entire set of edges incident on the node. This creates a new problem: an edge can be unimportant for one node, but important for the target node it is connected to. We solve this problem by only filtering an edge when both its connected nodes agree on it. This also implies that we first determine for all the nodes which edges are candidates for removal, before we actually filter them from the graph. If we would go through the graph node by node and filter the edges immediately, then the chosen edges would depend on the order in which we handled the nodes.

Another important aspect to mention is that it is impossible to create new isolated nodes, because the edge cutoff parameter is compared to the *normalized* utility values. This guarantees that at least one edge (the most important one) will always be preserved. A next problem we encountered is that we did not explicitly calculate a significance and correlation value for every edge. We did however incorporate these concepts into the edge weight. Since the utility value is meant to be an indication of how important the edge is, we can let the edge weight, as calculated in section 4.2, represent it.

Lastly we also have to determine the edge cutoff parameter. This parameter determines how careful the algorithm is with removing edges. A lower cutoff parameter will preserve most of a node's edges, while a higher one only preserves the strongest collaboration

relations. This parameter depends on the log under consideration and the purpose of the desired visualization.

This brings us to the following algorithm:

Edge filtering algorithm

(1) For very node, do the following:
  (a) Collect all the edges incident on this node
  (b) If the node has more than one incident edge, do the following:
    (i) Collect the weights of all these edges
    (ii) Normalize these weights between $[0, 1]$
    (iii) Add every edge for which the normalized weight < edge cutoff parameter to the list of possible candidates for filtering
    (iv) If this edge is already present in the list of candidates, it means that the other node it is connected to also suggested it as a candidate for filtering. Move this edge to the list of edges to filter
(2) Once all the nodes are handled, remove the edges that appear in the list of edges to filter, from the graph

5.2 Phase 2: Aggregation

In this second phase we form coherent clusters of programmers that are not very important but have very strong relationships between them. Such a group of programmers is represented by a single node in the graph, because their individual importance does not outweigh the added value of simplification. However, they are not abstracted from the graph because their strong mutual collaboration could provide interesting insights for the tool's end user.

First we will elaborate how the fuzzy miner [8] handles node aggregation, since this will form the base for our aggregation approach. The fuzzy miner selects every node that has a unary significance value below the node cutoff parameter as a possible candidate for aggregation. Next it examines the most highly correlated neighbour of each of these candidates. If this neighbour is a cluster node, the candidate under consideration gets added to this cluster and all of its incident edges are transitively preserved. If not, a new cluster is formed with the candidate as first member. Once every candidate has been handled, the second phase that consists of cluster merging, sets in. In this phase, each cluster gets checked whether all of its predecessors are also clusters. If so, the cluster under consideration gets merged with the most highly correlated one. The exact same procedure is also carried out for the successors.

Our aggregation approach consists of the same two phases: initial cluster building and cluster merging. After carrying out these two phases we also have to redefine the weights of both the cluster and its connected edges and determine the collaboration type of these edges (disjunct/pair programming, or both). In the next subsections we will explain each of these phases more into detail, including when and why we deviate from the original fuzzy miner approach.

5.2.1 *Initial cluster building.*

We start off by collecting the nodes that are possible candidates for aggregation. This means that we check for each node whether it is unimportant enough to aggregate. The fuzzy miner calculates this importance by only making use of a unary significance metric. In our case, however, the importance of a node is represented by its weight, which incorporates the unary frequency significance along with three centrality measures (as explained in section 4.3). This value is then compared to a node cutoff parameter, of which the value again depends on the log under consideration and the purpose of the desired visualization. By default, this cutoff parameter is set as the average weight of all the nodes in the graph. In most cases there will be a lot of nodes with a low weight, which lowers the average and limits the number of candidates for aggregation.

Once the candidates are selected, each of them has to be evaluated to determine whether they can be aggregated. This is only the case when they have an edge that is strong enough to carry out the aggregation. We deviate here from the original fuzzy miner approach. Once a candidate has been selected by the fuzzy miner it will certainly become a cluster by merging with its most highly correlated neighbour, regardless of the edge's weight [8]. In our approach it is possible that the candidate will eventually not be clustered at all. Because we want a cluster to represent a group of strongly connected programmers, an additional check is performed to ensure that the weight of that edge is large enough. To determine whether an edge is strong enough to pull two nodes together, we have to declare an aggregation-correlation parameter to which the edge weight can be compared. Only when the edge weight exceeds this parameter, the edge is strong enough to contract the nodes and form a cluster. Using the average edge weight of all the edges as the parameter and comparing this to the absolute edge weight is not possible. The reason for this is that the weight of a node depends on the weight of its connected edges. Since we only select nodes with a low weight as possible candidates, their edge weights will also be low. The result is that none of these edges have a weight that exceeds the aggregation-correlation parameter, so none of the nodes will actually be aggregated. So we have to find another way to determine whether an edge is strong enough to carry out aggregation.

We found our solution in a metric that is also used by the fuzzy miner, namely the ***distance significance***. The miner uses this metric to determine the significance of an edge, but it can be adapted to solve our problem. The distance significance looks at how much the significance of an edge differs from the significances of its connected nodes. The bigger the difference, the less its distance significance value. This metric locally amplifies important relationships between nodes and weakens the already insignificant ones [8]. So instead of using the absolute weight of an edge, we will look at the strength of the edge in the local context. We know that the candidate nodes have low weights and we are searching for strong edges, so the bigger the difference between the weights of these two, the more the edge qualifies to carry out aggregation. We choose the aggregation-correlation parameter to be the difference between the average node

weight of all the nodes and the average edge weight of all the edges. So for each candidate node we search the strongest edge whose weight exceeds this aggregation-correlation parameter. If no edge is deemed strong enough, the node remains untouched.

Now we can resume where we left off while following the fuzzy miner approach. For every node that has an edge capable of aggregation, we check whether the target node that the edge is connected to is a cluster or not. If it is, we add this node to that cluster; otherwise we build a new cluster with this node as the first member.

We conclude this first aggregation phase with an overview of the algorithm that was just explained. Next, the merging of these clusters will be discussed in phase 2.

Initial cluster building algorithm

For each node:
If the node weight < node cutoff parameter:
(1) Collect the edges incident on this node
(2) For each of these edges, do the following:
  (a) Calculate the distance significance as : *| edge weight - node weight |*
  (b) If the distance significance > aggregation-correlation parameter → this edge is capable of aggregation, so add it to the possible candidates
(3) From the list of candidate edges, select the strongest one to actually carry out the aggregation
(4) If the target node it connects to is a cluster → add the node and its incident edges to this cluster
(5) Else make a new cluster with this node as the first member

5.2.2 Cluster merging.
As stated before, the fuzzy miner only merges clusters when its pre- or postset contains only cluster nodes. Our collaboration graph is not directed, so we cannot make this distinction. However, it is in our case also not necessary that all the neighbouring nodes are cluster nodes. If two nodes have a very strong connection, they will most likely work on the same or related things and have more or less the same knowledge. So they can be considered as one. Nodes that collaborate with one of these two can theoretically also be seen as working with the other. This implies that in order to merge, not all the neighbouring nodes have to be clusters. This idea is illustrated in figure 3, where the green nodes represent the clusters.

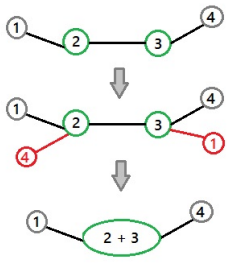

Fig. 3. *Not all neighbouring nodes have to be clusters in order to merge. Node 2 and 3 are cluster nodes that are also connected to a regular node. If node 2 and 3 can be seen as more or less the same, the nodes 1 and 4 can be considered as also working with the other cluster node. So losing the information to which original cluster node a regular node was connected, is an acceptable simplification.*

So for every cluster node, we check if one of its neighbours is also a cluster. If there are multiple candidates, we choose the one that is connected by the edge that has the greatest aggregation power that is above the aggregation-correlation parameter, as described in subsection 5.2.1.

Summarized this gives us the following algorithm:

Cluster merging algorithm

For every cluster, do the following:
(1) Collect its neighbouring cluster nodes
(2) For every edge that connects this node to a neighbouring cluster node, do the following:
  (a) Calculate the distance significance as follows: *| edge weight - node weight |*
  (b) If the edge's distance significance value > aggregation-correlation parameter → add it to the list of possible candidates for merging
(3) Select the candidate edge with the largest distance significance to actually carry out the merging
(4) Merge both clusters and transitively preserve the edges

5.2.3 Redefining the cluster's weights.
Now that we have created clusters of programmers, we also have to calculate a weight for both the cluster and its connected edges. We start by determining the **weight of these edges**. There are two possible scenarios: the edge can connect the cluster to either a regular node or to another cluster.

Let us first consider the scenario where the edge connects the cluster to a regular node. If this regular node is only connected to one node within the cluster, then its original weight is retained. If however, the regular node is connected to multiple nodes within the cluster, these connecting edges will be removed and replaced by one edge that represents all these connections to the cluster. It is this replacing edge that needs a new logical intermediate weight. This new weight will be the average of the weights of the edges that connect a cluster node to the regular node, but corrected for

the strongest edge. The reasoning behind this is that the strongest edge is important for visualization insights. Imagine a node that has a very strong relation to a node within this cluster, but also many insignificant edges to other nodes within the cluster. These many low edge weights will strongly decrease the average weight of all these edges that connect the cluster to this regular node. This causes the final weight for the replacing edge to be an underestimation. We correct this value as follows:

Cluster edge weight formula

Collect the weights of all the edges connecting this regular node to a node within the cluster:
The weight of the replacing edge =

$$\frac{the\ avg\ edge\ weight + the\ strongest\ edge\ weight}{2}$$

We will illustrate this with an example in figure 4:
The average edge weight would be an appropriate value to use as the weight of the replacing edge in the example on top. However, in the bottom example this value would be an underestimation. If we would use it, we would lose the information that the node is very strongly connected to one node within the cluster, and therefore also to the cluster as a whole. In contrast, applying our formula results in a value that represents this information more accurately.

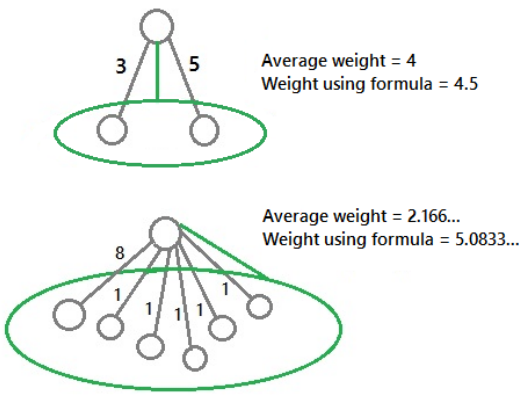

Fig. 4. *Two example scenarios that illustrate how the weight of the new replacing edge that connects the node to the cluster (depicted in green) is calculated.*

The second scenario is the one where the edge connects the cluster to another cluster. An example of this can be found in figure 5. Again, we collect the weights of all the edges between the two clusters and calculate the weight of the replacing edge according to the formula used in the previous scenario.

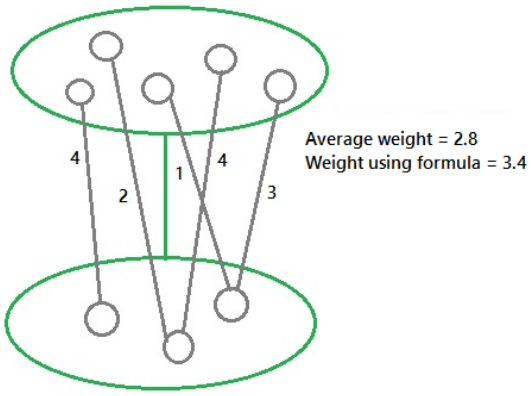

Fig. 5. *An example scenario that illustrates how the weight of the new replacing edge that connects two clusters (depicted in green) is calculated.*

Next we still have to calculate a **weight for the cluster node**. We do this in exactly the same way as with the edges: first we calculate the average weight of all the nodes that make up the cluster, after this we correct the weight by adding the weight of the most important node member and dividing this sum by two. In summary this gives us the next formula:

Cluster node weight formula

Collect the weights of all the node that make up the cluster:
The new cluster weight =

$$\frac{the\ avg\ node\ weight + the\ strongest\ node\ weight}{2}$$

5.2.4 *Redefining the cluster's edges collaboration type.*

The edges in our graph are labeled according to their collaboration type. An edge can get assigned one of three possible types, as already discussed in section 3.2:

- **Type 1:** pair programming edge
- **Type 2:** disjunct programming edge
- **Type 3:** pair and disjunct programming edge

After forming the clusters, we need to redefine the collaboration type of the edge connecting to this cluster. To do this we check the collaboration types of the edges connecting the regular node or the cluster to the cluster under consideration. If all of the edges are of type 1, the cluster edge gets assigned type 1; the same logic applies to type 2. If one of the edges is of type 3, the cluster edge gets assigned this type. If none of the above applies, which is the case when some edges are of type 1 and others are of type 2, the cluster edge gets assigned type 3.

5.3 Phase 3: Abstraction

In this last phase we remove certain programmers from the graph. Our abstraction approach however, is more nuanced than the one the fuzzy miner [8] uses, which removes all the isolated and singular

clusters. The miner already selected all the insignificant nodes in its aggregation phase and consequently removes all the detached parts of the process and every cluster that only consists of one node. The reason behind this is that isolated clusters can never be reached in the process model and singular clusters do not simplify the model. We can apply neither of these approaches to our problem case. Isolated clusters are an important aspect of our collaboration graph, because they draw attention to teams that specialize in a certain area and not engage in what other programmers are doing. The only nodes that should be removed in this phase, are the ones that are insignificant and not strongly connected to the rest of the graph. These programmers hardly contributed to the project, so they do not add any useful insights to the visualization. We do keep singular clusters, which are also programmers that did not contribute much to the project. However, they are relatively strongly connected to the rest of the graph, which is why they got clustered in the first place. This strong connection could possibly provide interesting insights, like for example a fairly new developer that is being trained by a core developer. So we will only remove programmers that are insignificant and not strongly enough connected to other programmers so they can be aggregated. We do this as follows.

First we search through the non-clustered nodes for possible candidates to abstract. This comes down to checking whether the node's weight falls below the node cutoff parameter, as we did with aggregation in subsection 5.2.1. If we find a possible candidate, we check whether all of its incident edges are too weak to carry out aggregation. We do this by comparing the distance significance to the aggregation-correlation parameter, as described in subsection 5.2.1. If all of the edges are indeed too weak, we can remove this node and all of its incident edges from the graph. With this we conclude our graph simplification.

Abstraction algorithm

For every node that does not belong to a cluster, do the following:

(1) Check whether the node's weight < node cutoff parameter
(2) If so, check if every incident edge has a distance significance value < aggregation-correlation parameter
(3) If so, remove the node and all of its incident edges from the graph

6 TOOL DEMONSTRATION AND BUSINESS RELEVANCE

In this section we will demonstrate the tool. The version control system log that we use in this demo originates from Cegeka, which is a multi-national technical service provider company. This log contains 1486 chronologically ordered commits, dating from June 2009 to November 2017, and these are all related to the same project. Each commit describes the contributors, the exact time stamp the commit took place, a message and a list of files that were modified, added or deleted. The log contains a total number of 6329 altered files.

After the collaboration visualizer analyzes this log, we are presented the following graph, in which all contributors have been anonymized:

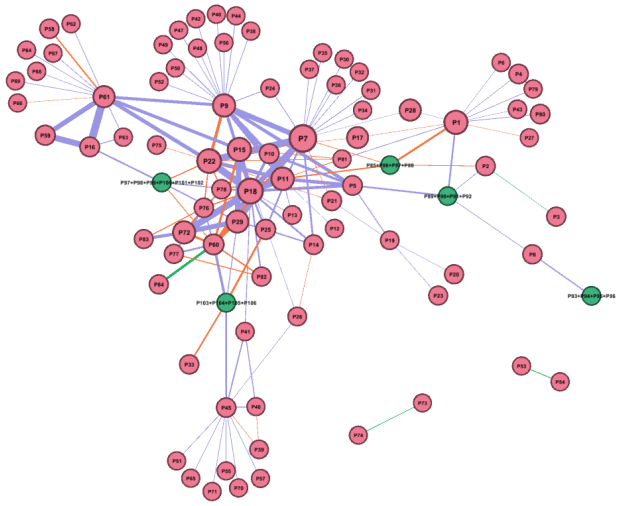

Fig. 6. *Collaboration graph that is the result of the tool analyzing a real-life log. The green nodes are clusters of programmers, while a pink one represents a single programmer. The edges are colour-coded to represent the following: the green edges indicate pair programming, the orange ones disjunct programming and the purple ones imply that the programmers engaged in pair as well as disjunct programming.*

The resulting graph counts 89 nodes and 146 edges. All the green nodes are clusters, which make up 5.62% of all nodes. The green edges indicate pair programming, the orange ones disjunct programming and the purple both pair and disjunct programming, with a distribution of respectively 2.74%, 26.71% and 70.55%.

The first thing that catches the eye is the two isolated pairs in the bottom right corner of figure 6. It is a good sign that there are no single isolated nodes, but these two small groups are also worth investigating a little further. Both programmers 53 and 54 and programmers 73 and 74 are only engaging in pair programming with each other and do not interact with the rest of the graph. This could mean that they specialize in a certain aspect of the project we are analyzing. These pairs show with high certainty (because the relation is pair programming) that there are two people with knowledge of these aspects. However, their relation is not very strong and none of the programmers seems to be very important, as indicated by the size of their node. After inspection of the commits in the log, we could verify that the reason for this is that both pairs made a very limited contribution to the project. It is worth keeping an eye out for such isolated groups, especially if they are small and the members have a large importance. The reason for this is that

they are valuable sources of knowledge and not easily replaced. To mitigate this risk, one might want to expand these groups.

A second aspect that stands out are the nodes P1, P7, P9, P45 and P61 that have a following of many nodes with a degree of one. There are multiple possible explanations for this. First of all we can see that the weights of the followers are rather small, so their contribution to the project is limited. Such a node could be someone that was called in to fix a difficult bug, but you would expect such an expert to play a bigger role in the project on multiple aspects, so this is rather unlikely. On the other hand it is possible that such a node is a starting programmer, who has not had the time yet to contribute a significant part to the project. A last, and most likely, possibility is that these nodes are just wrongly identified as programmers by the tool. To validate this theory we need expert knowledge on the actual composition of the development team of Cegeka. With this information we could further fine tune the tool's parameters to filter these out.

Let us zoom in and analyze some striking relationships. There are three of these relationship that we would like to discuss (indicated on figure 7). The first one is the strong pair programming relation between P60 and P84. This could possibly represent a teacher - apprentice relationship, where programmer P84 is still learning under supervision.

The second relation we would like to mention is the strong disjunct programming relation between P1 and the cluster of programmers P85, P86, P87 and P88. This relation implies that they are working on the same aspects of the project's code, but P1 plays overall a more important role. It also seems that P1 is involved in multiple separate parts of the code, as he also collaborates strongly with the cluster that is made up of programmers P89, P90, P91 and P92. This causes programmer P1 to be a very valuable resource.

Lastly, the rather strong disjunct programming relation between P58 and P61 could mean that programmer P58 is solely focusing on one specific part of programmer P61's tasks. As there is no other node that connects to P58, these two programmers are the only ones working on that specific aspect of the code. Scheduling a pair programming activity once in a while or entrusting another programmer with the same task, is recommended to mitigate the indispensability of this programmer.

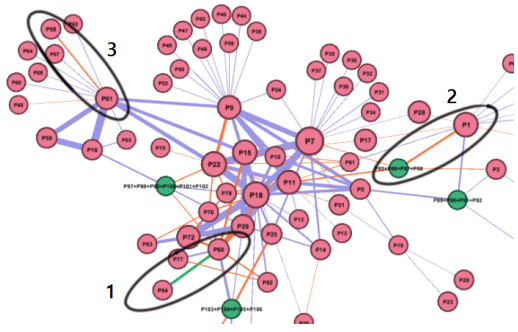

Fig. 7. *3 Striking relationships between nodes*

We conclude this demonstration by identifying the core development teams. In order to do this, we look at the programmers and relationships that have the heaviest weights, which makes them the most important ones. This is depicted in figures 8 and 9, that respectively show the top 20% heaviest nodes and top 10% heaviest edges. We can distinguish two main teams: a smaller one that is made up of programmers P61, P16 and P59; and a larger one with programmers P7, P9, P11, P15, P18, P22, P29 and P72 being the most important ones in this group. Notice that programmer P61 is also to a certain degree involved in the second team and therefore forms a contact point between the two. This makes this programmer very valuable.

If we look at the most important programmers, we can see a risk arising. Programmers P1, P17, P21 and P28 are among the most valuable assets of the project, but are connected by some of the weakest relationships in the graph. If we look back at figure 7, we can see that P1 has rather strong relations with two clusters, so the risk P1 poses is limited. However, P17, P21 and P28 are only connected by the edges that are depicted on figure 8 and therefore pose a significant threat. These programmers make significant contributions to the project but do not strongly collaborate with others. This could cause problems when one of them falls ill or leaves the company. It is advised to look further into these cases and take precautions.

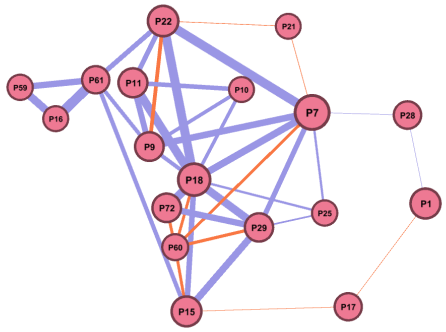

Fig. 8. *The top 20% heaviest nodes show the most important programmers.*

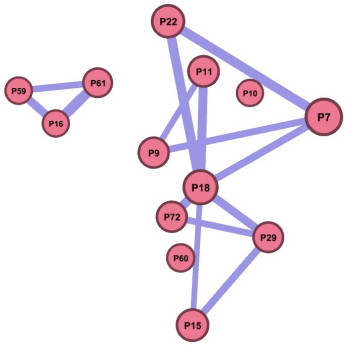

Fig. 9. *The top 10% heaviest edges show the strongest collaboration relationships.*

7 DISCUSSION AND FUTURE WORK

In this paper, we described a tool that can extract how programmers collaborated on code from a version control system log and visualize this in the form of a graph. This graph can be used to reveal potential risks (like indispensable resources) and improve team awareness. Due to many similarities, we gathered inspiration for the development of the tool from the fuzzy miner algorithm [8], which is a process mining discovery algorithm. We combined the metrics from the fuzzy miner that we deemed applicable to our case, with metrics from standard graph theory.

Since this is the very first version of our tool, there are still aspects that can be improved and fine tuned. The proper parameter settings differ from log to log and also depend on the insights the end user wants to gain from the visualization. Further research should be done to provide an improved default parameter setting. Further, the tool currently only works with logs generated from a Tortoise SVN system. There exist however multiple version control systems that can be used, which might structure their logs differently. Including logs from these systems as an input option is a good requirement for the next version of the tool. Lastly, this paper is limited to a demonstration of the applicability of the tool, but validation still needs to be added.

Acknowledgements:
This research is supported by Cegeka, who was kind enough to provide us with their data. I would also like to thank my promoter Mieke Jans and supervisor Mathijs Creemers for their feedback and helpful ideas.

REFERENCES

[1] Bala, S., Cabanillas, C., Mendling, J., Rogge-Solti, A., and Polleres, A. Mining project-oriented business processes. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2015).
[2] Bastian, M., and Heymann, S. Gephi : An Open Source Software for Exploring and Manipulating Networks. *ICWSM* (2009).
[3] Cambridge Intelligence KeyLines. Social network visualization: Using keylines to visualize networks of people, 2019. https://cambridge-intelligence.com/keylines/social-network-analytics/.
[4] Cockburn, A., and Williams, L. The Costs and Benefits of Pair Programming. *Extreme programming examined* (2001).
[5] Eick, S. G., Steffen, J. L., and Sumner, E. E. SeesoftâĂŤA Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering* (1992).
[6] Georget, L., Tronel, F., and Tong, V. V. T. Kayrebt: An activity diagram extraction and visualization toolset designed for the Linux codebase. In *2015 IEEE 3rd Working Conference on Software Visualization, VISSOFT 2015 - Proceedings* (2015).
[7] GraÄĲanin, D., MatkoviÄĞ, K., and Eltoweissy, M. Software visualization. *Innovations in Systems and Software Engineering 1* (09 2005), 221–230.
[8] Günther, C. W., and van der Aalst, W. M. P. Fuzzy Mining âĂŞ Adaptive Process Simplification Based on Multi-perspective Metrics. 2007.
[9] Hu, Y. The Mathematica ® Journal Efficient, High-Quality Force-Directed Graph Drawing. *Methematica Journal 10* (2006), 37–71.
[10] Jacomy, M., Venturini, T., Heymann, S., and Bastian, M. ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. *PLoS ONE* (2014).
[11] Lanza, M., D'Ambros, M., Bacchelli, A., Hattori, L., and Rigotti, F. Manhattan: Supporting real-time visual team activity awareness. In *IEEE International Conference on Program Comprehension* (2013).
[12] Marcus, A., Comorski, D., and Sergeyev, A. Supporting the evolution of a software visualization tool through usability studies. In *Proceedings - IEEE Workshop on Program Comprehension* (2005).
[13] Newman, M. *Networks: An Introduction.* 2010.
[14] Quante, J. Do dynamic object process graphs support program understanding? - A controlled experiment. In *IEEE International Conference on Program Comprehension* (2008).
[15] Sensalire, M., Ogao, P., and Telea, A. Evaluation of software visualization tools: Lessons learned. pp. 19 – 26.
[16] Storey, M. A., Fracchia, F. D., and Müller, H. A. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software* (1999).
[17] Storey, M. A., Wong, K., and Müller, H. A. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming* (2000).
[18] Team, N. NetworkX, 2014.
[19] Van der Aalst, W. *Process mining: Data science in action.* 2016.
[20] Van Der Aalst, W. M., Reijers, H. A., and Song, M. Discovering social networks from event logs. *Computer Supported Cooperative Work* (2005).
[21] Wasserman, S., and Faust, K. *Social network analysis: methods and applications II.* 1994.
[22] Wettel, R., Lanza, M., and Robbes, R. Software systems as cities: a controlled experiment. In *ICSE '11 Proceedings of the 33rd International Conference on Software Engineering* (2011).
[23] Wettel, R., Lanza, M., and Robbes, R. Software systems as cities: a controlled experiment. *2011 33rd International Conference on Software Engineering (ICSE)* (2011).
[24] Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* (2005).