DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Incremental Calculation of Graph Centrality Metrics

Alice Rey

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Incremental Calculation of Graph Centrality Metrics

Inkrementelle Berechnung von Graphzentralitätsmetriken

Author:	Alice Rey
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Jürgens, Roman Haas
Submission Date:	15.09.2018

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2018 Alice Rey

Abstract

In the past, graph centrality metrics were already successfully used in static software analysis. An example for this is the identification of the most central classes of a software system and (together with additional information) the identification of unnecessary code of a software system. The graph centrality metrics are applied to dependency graphs consisting of the classes of the software systems and their dependencies between each other. The computation of centrality metrics on large graphs typically is very time consuming due to the fact that these metrics have to be recomputed after every change made to the software. The main idea of the incremental software analysis is to speed up the recomputations to be able to give immediate feedback to developers. This is accomplished by taking the previous state as a basis and together with the changes made to the system computing the new state. Previous work has shown that centrality data provides useful information to developers. However, recomputation of centrality measures for every change to a software system is too costly. For practical use of code centrality information in software analysis, incremental calculation approaches are more applicable as they are much cheaper.

In this bachelor's thesis we discuss different incremental approaches for calculating the PageRank centrality. In the literature, there can be found incremental approaches for the PageRank computation, but these were made for the analysis of the Web structure. We evaluate their applicability in incremental software analysis. The main idea is to only recompute the metric values for changed classes and classes that depend on them, instead of recomputing the metric for all classes of the system for every revision.

We evaluate the incremental PageRank approaches by measuring the runtime improvement, as well as the precision loss by utilizing the incremental instead of the standard PageRank. Therefore, we make a history analysis of 6 different software systems, both open- and closed-source. We show that for our chosen project the runtime is at least three times faster when applying an incremental approach, instead of using the standard PageRank algorithm. However, the actual PageRank values do not differ much, depending on the chosen approximation level. The higher the approximation, the better the runtime and the worse the precision. We present the best compromise between runtime improvement and precision and show that our approximation works well so that it can be applied in incremental calculation of code centrality.

Contents

1 Introduction

1.1 Background and Motivation

In the field of software analysis there exist two types of analysis tools. The tools using the standard analysis approach recompute all software analyses from scratch every time the software repository changes. The number of changes is usually very high during the software development. However, the amount of change a single developer adds to the repository with one commit is very small compared to the total size of software projects. The other type of software analysis is needed to provide quick feedback to the developer after adding some changes without having to wait for hours until all analyses are finished. Developers usually only want a short feedback to check if there are any defects in the analysis results that need to be fixed. This type is the so called incremental software analysis. The main idea of the incremental approach is to utilize the previous state of the software analysis as a base and to combine this base with the changes made to the system to compute the new state. This enhances the runtime of especially large software projects and leads to a much quicker feedback for the developer.

Analyses that can be performed incrementally are for example type change analysis, instability analysis or the origin analysis of source code files [1]. The base of these and many other analyses are the dependencies between files and types of a software system. To keep those dependencies up to date, one needs to check all changed files for changed outgoing dependencies. Based on these, the changed incoming dependencies of other files can be found as well. In many software projects a significant amount of duplicated, so called cloned code, can be found. In the non-incremental software analysis the entire software system has to be read and all clones have to be detected which implies high computational costs due to the number of comparison that have to be performed. In the incremental case only the clones affected by the changed code are reanalyzed and only based on these changed classes new clones are searched [2].

Very interesting problems of the software analysis are the central class recommendations and the unnecessary code identification. The central class recommendation should help new developers to understand a software system more quickly by looking at the most important code first [3]. The idea behind the unnecessary code identification is that in every software project there is usually a certain amount of unused code which is

not needed anymore. The approach should help developers to find this code which can be very difficult to identify per hand especially for large software projects [4]. As a base, both problems use centrality metrics to specify the importance of the classes. These have to be computed from scratch every time the software system undergoes changes which makes the computation not applicable in the incremental software analysis. Both applications have successfully applied PageRank as centrality metric for their use cases. The purpose of this bachelor's thesis is to find a solution that improves the runtime of the recomputation of PageRank by computing it incrementally. If we succeed, these software analysis problems can be integrated into the incremental software analysis.

PageRank is a centrality metric originally used to identify the most important pages of the Web for the search engine Google [5]. The Web graph is huge and has to deal with a lot of changes. If this centrality metric gets recomputed from scratch after every change, this will not be applicable at all, due to the high computational costs. There exist research papers which try to improve the runtime needed to recompute PageRank. Some of them are relying on the previous state and change made to the graph and just recompute the PageRank incrementally [6] [7].

1.2 Problem Statement

Centrality metrics like PageRank are commonly used for comparing the importance of pages in the Web graph. These centrality metrics can also be useful for centrality related topics in the software analysis area, like the central class recommendations. In this case the algorithm is applied to the dependency graph of the software project instead of the Web graph. Both, the dependency graph of a software project and the Web graph have to deal with a huge amount of changes. To keep the centrality metrics always up to date, they have to be recomputed from scratch after every change. This recomputation can be time-consuming depending on the size of the graph. Even if only a few dependencies and nodes have undergone changes since the last computation, the effort that has to be made never changes. This makes the computation of those metrics inefficient, especially when many revisions have to be computed in a row. For large projects the PageRank computation for one revision took up to 15 minutes in our test setup, depending on the number of changes. During the development phase of large projects, 50 commits per day is not an unusual number. This leads to a total runtime of approximately 10 hours only for the recomputation of the PageRank values. Thus, these metrics are not applicable for a continuous software analysis.

1.3 Contribution

In this bachelor's thesis, we present an approach for an incremental recomputation of the centrality metric PageRank. There already exist papers which describe these incremental approaches for the web graph, but they have not been used in the field of software analysis yet. We will apply these incremental PageRank approaches which are originally made for the Web graph to the dependency graph of different software systems. We evaluate their runtime improvement and their loss of precision. Finally, we apply the incremental approach to the already presented software analysis problems that use PageRank to check how applicable they are in the software analysis field.

1.4 Thesis Structure

The thesis is structured as follows: Chapter 2 gives definitions of important terms used in the bachelor's thesis. Chapter 3 will start with an overview of related work in the incremental PageRank area as well as in the area of software analysis where centrality metrics like PageRank are applied. The next step is the explanation of our approach to incrementally recompute the PageRank which is presented in Chapter 4. In Chapter 5, we examine how much faster the incremental approach is, how much precision we lose due to the approximations and how applicable the approach is in the software analysis field. We conclude with the summary of our results and ideas for future work.

2 Terms and Definitons

2.1 Dependency Graph

A software system consisting of many classes which are depending on each other can be represented as a dependency graph. This dependency graph is used as a base for many software analysis and we will utilize it in this bachelor's thesis to specify the centrality of the classes. A dependency graph is a graph consisting of vertices and edges. In our software dependency graph the vertices correspond to the classes of a software system and the edges correspond to the dependencies between those classes. A class A depends on another class or interface B if at least one of the following cases are fulfilled:

- A implements or extends B
- A creates an instance of B
- A type-cast to B occurs in A
- There exists a field, a local variable, a method parameter or a method return type of the type B in A
- A accesses a field or calls a method of B
- A throws an exception of the type B

2.2 PageRank

PageRank is a centrality metric which will be utilized throughout the thesis to specify the centrality of the different classes of a software system. It was developed as a ranking algorithm based on a graph with the original purpose to compute the importance of Web pages. The method was invented by Page for the search engine Google [5]. A page is highly ranked if the sum of the ranks of its backlinks is high. Therefore a page with many low ranked backlinks and a page with only a few but highly ranked backlinks can have similar PageRank values. The idea behind PageRank is a Web surfer who keeps clicking on successive links randomly. Normally, this surfer will jump to some

other page after a while and this possibility of randomly choosing a page without following a link to it is also considered in the formula:

$$PR(p) = d \cdot \frac{1}{n} + (1 - d) \cdot \sum_{(q,p) \in G} \frac{PR(q)}{OutDegree(q)}$$

G is the dependency graph, n is the number of nodes and the *OutDegree* is the number of hyperlinks on the page q. The second term describes the case that the surfer is arriving at a page by clicking a link. Therefore the term sums up all the rank contributions made by all pages q pointing to the page p. The first term deals with the special case that the surfer arrives from anywhere, which has therefore the uniform probability $\frac{1}{n}$. The factor d gets assigned a value between 0 and 1 and determines the probability distribution. The factor d gives the probability of a random jump and is also called dampening factor and $1 - d$ is the probability of arriving on a page by clicking on a link [6].

In our case the dependency graph will not represent the Web pages and the links between them, but the classes of a software project and their dependencies.

2.3 Spearman's rank correlation metric

We use the spearman's rank correlation metric to evaluate how precise the PageRank values are when an incremental instead of the standard PageRank approach is applied. The runtime-improving incremental approach can only be applied if the correctness of the resulting values is guaranteed. The spearman's rank correlation coefficient [8] is used to determine the relation between two data sets.

Let $(X_1, X_2, ..., X_n)$ and $(Y_1, Y_2, ..., Y_n)$ be two data sets of size n. R_{X_i} denotes the rank of X_i. If there exist no two values in a data set with the same rank, one can utilize the normal spearman's rank correlation coefficient formula:

$$p = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1))}$$

with $d_i = R_{X_i} - R_{Y_i}$

In our case there exist classes which have the same rank. Therefore the spearman's metric has to be computed with the following formula:

$$p = \frac{cov(R_X, R_Y)}{\sigma(R_X)\sigma(R_Y)}$$

where $cov(R_X, R_Y)$ is the covariance of the ranks of X and Y, $\sigma(R_X)$ is the standard deviation of the ranks of X and $\sigma(R_Y)$ is the standard deviation of the ranks of Y.

The spearman's rank correlation coefficient is used to determine if there exists a relation between two random variables. Two data sets have a positive correlation if both, the large and the small values tend to be associated with each other. The correlation of the two data sets is negative if the large values of one data set tend to be associated with the small values of the other data set and vice versa. In the third case the two data sets are independent and not correlated.

A correlation measurement between two variables only takes values between -1 and 1. A positive correlation exists if the value is positive. The data sets are identical if the metric has a value of 1. For negative correlations, the metric value is negative. One dataset is the opposite of the other, if the metric has a value of -1. If the correlation is close to zero one can say that the two variables are not correlated [9].

3 Related Work

In their paper Desikan et al. [6] present an approach to deal with the key challenge of PageRank: The computation of PageRank for a large and evolving graph. They denote that the rate of changes is very low compared to the size of the graph. According to them, their incremental approach is a significant improvement of computational cost because it only recomputes the small portion of the Web graph that has undergone changes since the last computation. They group the graph into two portions which is depicted in 3.1.

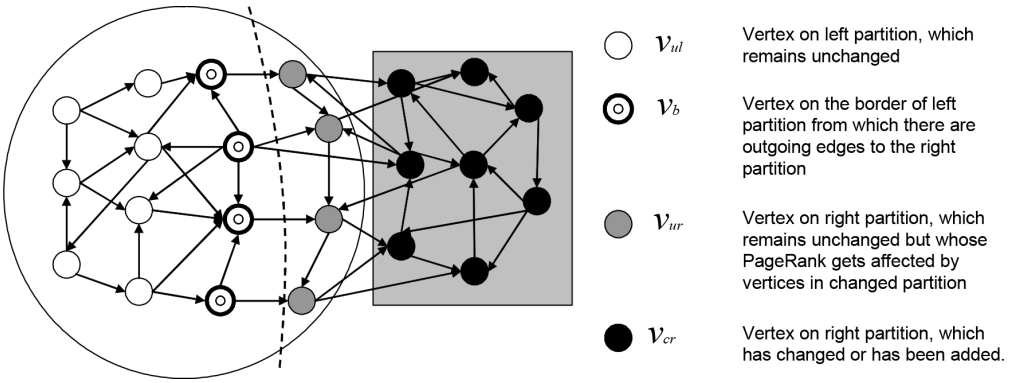

Figure 3.1: The partition of the dependency graph. Reprinted from [6]

One portion contains all nodes which have undergone changes since the last computation (the right partition in Figure 3.1) and the other partition is the one that stayed unchanged and has only outgoing edges to the other partition (the left partition in Figure 3.1). The latter partition will not be affected by the nodes in the first partition. This is true since the PageRank of a page only depends on the pages that point to it and is therefore independent of its outdegree. The first step in their approach is to identify such a partition that has no incoming nodes from the partition with the changed nodes. In the next step the PageRank values of the changed partitions nodes get recomputed. To compute these, the border nodes from the unchanged partition are also taken into consideration, because these nodes influence the values of the changed partition. The

unchanged partition only gets scaled by the factor:

$$\frac{Order\ of\ the\ graph\ at\ the\ previous\ time}{Order\ of\ the\ graph\ at\ the\ present\ time}$$

A vertex is called changed if an edge between itself and any other vertex has been inserted or if the edge weight changed. After identifying all changed vertices, all pages affected by their PageRank are iteratively determined. This includes all vertices that have a parent which has been changed or is affected by a parent vertex as well. In the end the changed partition consists of all changed vertices and all vertices affected by them.

In their paper Chien et al. [7] present a very efficient algorithm to incrementally compute a good approximation to Google's PageRank as the links of the Web graph evolve. According to them, their incremental approach is fast and yields excellent approximations to PageRank. The restriction of their approach is that they only address the addition or deletion of links and not the addition or deletion of a node. In their approach they identify a small portion of the graph in the closest neighbourhood of the given set of changed links. They model the rest of the Web graph as a single node in the smaller graph. All nodes whose PageRanks are most affected by the addition or deletion of an edge will be a part of the subgraph. They choose the most affected nodes by assigning weights to the nodes and if weight of a node exceeds a chosen threshold, it will be added to the subgraph. For a new edge from node A to node B this means, that node A gets assigned the weight 1 and then all descendant node weights are assigned by the following formula:

$$\frac{1-d}{OutDegree(A)}$$

with d as the pagerank dampening factor. The threshold that the weight of a node has to exceed that the node is added to the subgraph has to be chosen in such a way that the result approximates the PageRank values sufficiently good and the resulting graph with the rest of the original graph modeled as a supernode is not too large.

These paper all deal with the problem of PageRank computations on evolving web graphs. Tough, these computations can also be applied in software related topics as one can see in the following works where the PageRank algorithm is used for different software analysis problems.

Steidl et al. [3] present an approach that uses network analysis to recommend central software classes. When developers start developing an existing and unknown software system, they have to understand the system first. This training period can be time

consuming and it is crucial in which order the software artefacts should be read. The usual approach is to understand the key classes of a system at first. The problem is, that for a new developer the key classes of a system are not obvious and other developers are already biased by the part of the software they usually work with. Steidl et al. assume that a class is important if many other classes depend on it. Therefore, they determine the most important classes of the software by looking at the dependency graph of the system. The first step is to compute the dependency graph and in the second step they calculate a centrality index for each node. The used centrality indices are: PageRank, Betweenness and Markov. In the end they recommend the top ranked classes as central software classes.

Sora [10] also deals with the problem of understanding a software system as a new developer. She claims that there is often missing some useful information for the new user to start. In her opinion it would be useful for a start in program comprehension to have a short list of classes which are the most relevant. In her paper she presents an approach for identifying the most important classes based on a graph-ranking algorithm which adapts PageRank. The difference to the dependency graph that we are using is, that she assigns different weights to the different dependency relationships between two classes: A dependency gets assigned the value 1, if it is a local variable dependency. Weight 2 is for a distinct method that is called, weight 3 gets assigned to the parameters, return values and member dependencies. The highest weight has the value 4 and gets assigned to inheritance and realization dependencies. In the end the weight of the edge between two classes is the summation of all dependencies from one class to the other one.

In another paper of Sora [11], she is adding fuzzy rules which have attributes of the classes in their premises. These attributes are the size of the class, the weighted incoming and outgoing dependencies and the PageRank value of the class. She assumes that a big class which has many interactions with other classes is more likely to be an important class of the system.

Kamran et al. [12] present an approach that suggests the classes that are potentially interesting for initially understanding a software system. The assumption they make is that the maintenance phase of already built software projects is more time consuming than the development of a new software project. The part that is the most challenging and the most expensive in terms of time is to build and understand the existing system. In many cases there is no documentation of the system or similar artefacts available or it is not reflecting the current state of the system. They state that these central classes have a supervisory role in the application. They give instructions to a large number of

classes and dictate the work to perform.

Haas [4] deals in his master thesis with the problem of unnecessary code which is a known waste of time and money in the maintenance phase of large software projects. His approach applies a heuristic based on stability and decentrality to identify the unnecessary code. The assumption he makes is that code which is changed very rarely and only plays a minor role in the software system tends to be not needed. He tries to find the most stable and decentral files, groups them together and recommends the biggest chunks as candidates for deletion. The decentrality parameter is computed using the graph centrality metrics PageRank, Hits and a Markov-chain approach. These centrality metrics are just utilized in the opposite way. In his case not the highly ranked results but the lowest ranked files are interesting. The presented approach calculates a set of the most decentral classes and a set of the most stable classes and intersects them. The suggestions are gained from this intersecting set by building chunks of unnecessary code, where the package structure is taken into consideration as well.

4 Approach

The goal of this bachelor's thesis is to find a way to compute the PageRank algorithm in a more efficient way while the software project and therefore its dependency graph evolves. The following sections describe the incremental PageRank algorithms which are taken from the literature and are adapted to fit in our scenario.

4.1 Iterative PageRank

Our incremental approach still utilizes the original PageRank algorithm, but it narrows down the number of values that have to be recalculated. After finding out which values have to be recomputed and after scaling all values, the iterative PageRank algorithm is executed, as depicted in the pseudo code below. Therefore the iterative PageRank algorithm is explained shortly before presenting the incremental approach.

```
1:  function iterativePageRank(Graph, recalculatedVertices)
2:      iterations ← 0
3:      repeat
4:          iterations ← iterations + 1
5:          maxDelta ← 0
6:          for each vertex N in recalcualtedVertices do
7:              tempPageRank ← N.getPageRank()
8:              N.setPageRank(PageRank(Graph, N))
9:              if absoluteValue(tempPageRank, N.getPageRank()) > maxDelta then
10:                 maxDelta ← absoluteValue(tempPageRank, N.getPageRank())
11:             end if
12:         end for
13:     until maxDelta < tolerance || iterations > 100
14: end function
```

The algorithm is presented as a function with two input values: The dependency graph and the set of vertices that have to be recalculated. The graph is needed to get the total number of vertices in it as well as the parent vertices of the vertex that is

getting recomputed. To determine the PageRank value of a vertex all PageRank values of vertices that can reach this vertex have to be computed before. That is the case, because the PageRank value of a node influences all its successive nodes. Therefore the PageRank formula is applied iteratively to the vertices until the difference between the old and the new PageRank values is sufficiently small. In the formula that is expressed by the *tolerance*. Another boundary is set to the number of iterations. Both the *tolerance* and the maximum number of iterations, prevent the program to end in an infinite loop (Line 13 in the *iterativePageRank* function). The *maxDelta* variable makes sure that all values are sufficiently exact, before the iteration ends. It gets assigned the maximum difference of an old and new PageRank value of a vertex (Lines 9-11 in the *iterativePageRank* function).

To complete the iterative PageRank pseudo code, the PageRank formula already presented in Chapter 2.2 is rewritten in the pseudo code below as a function with the graph and the vertex for which the PageRank gets computed as input parameters.

```
1: function PageRank(Graph, vertex)
2:     pageRank ← d ÷ Graph.getVertexCount()
3:     for each vertex N in Graph.getParentVertices(vertex) do
4:         pageRank ← pageRank + N.getPageRank() ÷ N.getChildVertices().size()
5:     end for
6:     return pageRank
7: end function
```

4.2 Incremental PageRank

4.2.1 Which PageRank values have to be recalculated and why?

To incrementally recompute the PageRank values, one has to specify first which PageRank values are affected by the changes made to the graph. There are two possible ways changes can be made to the graph: The addition or deletion of an edge and the addition or deletion of a vertex. In our context this means the addition or deletion of a dependency between two classes or the addition or deletion of a class. In the following subsections both cases are getting explained in more detail.

Dependency addition or deletion

If a dependency between two classes is added, this is modeled in the dependency graph as an addition or deletion of an edge.

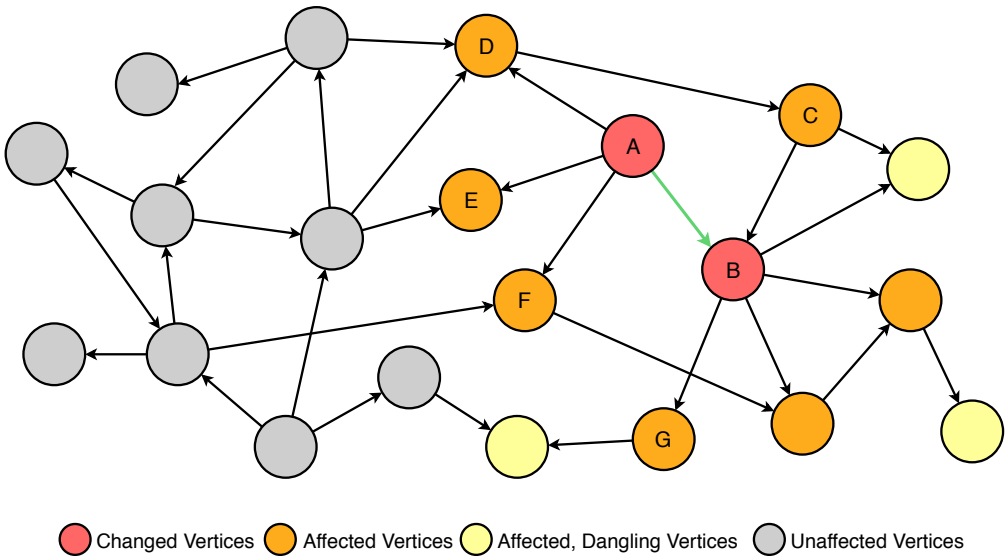

Figure 4.1: The addition of an edge to the dependency graph

As depicted in Figure 4.1 the addition of an edge between A and B leads to a recomputation of the PageRank value of B. This is the case, because every PageRank value depends on the PageRank values of their parent nodes:

$$PR_i(B) = d \cdot \frac{1}{n} + (1-d) \cdot \left(\frac{PR_i(C)}{OutDegree_i(C)} \right)$$

$$PR_{i+1}(B) = d \cdot \frac{1}{n} + (1-d) \cdot \left(\frac{PR_{i+1}(C)}{OutDegree_{i+1}(C)} + \frac{PR_{i+1}(A)}{OutDegree_{i+1}(A)} \right)$$

with i the state of the graph before the edge addition and $i+1$ the state after the edge addition.

A is also marked as changed, but only because its child nodes will be affected by the change. These nodes, in our case D, E and F, are affected, because the impact of A's PageRank value on these nodes is getting smaller, since A has now one childnode more than before:

$$PR_i(D) = d \cdot \frac{1}{n} + (1-d) \cdot \left(\frac{PR_i(A)}{OutDegree_i(A)} \right)$$

$$PR_{i+1}(D) = d \cdot \frac{1}{n} + (1-d) \cdot \left(\frac{PR_{i+1}(A)}{OutDegree_{i+1}(A)} \right)$$

with $OutDegree_i(A) = 3 \neq 4 = OutDegree_{i+1}(A)$

But these vertices are not the only ones affected by the change. In fact, all subsequent vertices of the changed vertices will be affected. They are either childnodes of A (this scenario was already explained above) or one of their parentnodes PageRank value has changed. This implies childnodes of changed vertices but also the childnodes of these childnodes and this scenario continues until there are no subsequent vertices found anymore. In our scenario in Figure 4.1 this happens for example to G:

$$PR_i(G) = d \cdot \frac{1}{n} + (1-d) \cdot \left(\frac{PR_i(B)}{Outdegree_i(B)} \right)$$

$$PR_{i+1}(G) = d \cdot \frac{1}{n} + (1-d) \cdot \left(\frac{PR_{i+1}(B)}{Outdegree_{i+1}(B)} \right)$$

with $PR_i(B) \neq PR_{i+1}(B)$ as shown above

The only vertices not affected by the changes are those which are not a subsequent vertex of a changed vertex, because their PageRank value is only affected by their parents PageRank value and if no parent PageRank value has changed, there is no need to recompute the childnodes PageRank values.

The deletion of an edge has similar effects as the addition of an edge. The edge B also has to be recomputed, because it loses one parent node, the child nodes are also needed to be recomputed because the outdegree of A is now smaller. And because of these changed nodes all their subsequent nodes also have to be recomputed, like in the addition case.

Class addition or deletion

The addition or deletion of a class from a software system leads to the addition or deletion of a vertex in the dependency graph of the system.

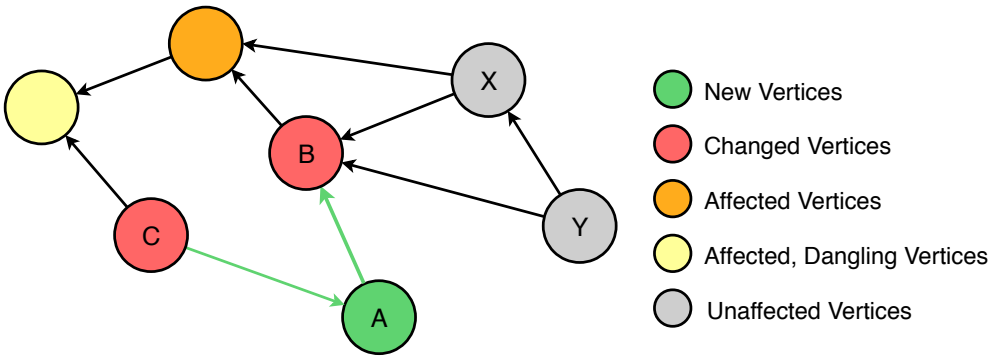

Figure 4.2: The addition of a vertex to the dependency graph

An exemplary situation is displayed in Figure 4.2. There are both a new vertex A and two new edges inserted, one between A and B and the other one between C and A. The behavior of an edge addition was already discussed above. The only new thing here is, that a newly added vertex is added to the list of changed vertices. This is obvious, because it is not only needed to get recalculated, but it needs to get an initial PageRank value. In the case of a vertex addition all vertices have to be recomputed intuitively, but there are vertices which are not directly affected by the new vertex. These are the same ones as in the edge addition case: All vertices that are no subsequent vertex of a changed vertex. These not directly affected nodes only have to be scaled by the factor

$$scalingFactor = \frac{Order\ of\ the\ graph\ before\ the\ edge\ addition}{Order\ of\ the\ graph\ after\ the\ edge\ addition} = \frac{n}{m}$$

.

$$PR_{i+1}(X) = \frac{n}{m} \cdot PR_i(X) = \frac{n}{m} \cdot d \cdot \frac{1}{n} + (1-d) \cdot \sum_{(q,p) \in G} \frac{\frac{n}{m} \cdot PR_i(q)}{OutDegree_i(q)}$$

with $PR_{i+1}(Q) = \frac{n}{m}(Q)$ similar to PR_{i+1} as formulated above. This continues until the border nodes which have no incoming edges. This state is in our example scenario already reached after one step for the vertex Y:

$$PR_{i+1}(Y) = \frac{n}{m} \cdot PR_i(Y) = \frac{n}{m} \cdot d \cdot \frac{1}{n} = d \cdot \frac{1}{m}$$

with n the number of total vertices before the vertex addition and with m the number of total vertices after the vertex addition. The deletion of a vertex has a similar effect on the graph. In that case m is equal to $n-1$ and in the addition case m is equal to $n+1$. Several vertex additions and deletions are possible as well due to the choice of a neutral parameter m which is not fixed to $n-1$ or $n+1$.

4.2.2 What are the steps to compute PageRank incrementally?

At the initial commit the PageRank is computed iteratively with the formula stated in Section 2.2 until there are no significant changes in the PageRank values between two iteration steps anymore. Due to the computational accuracy of a computer program one cannot choose 0 as maximum allowed change because this could lead to a infinite loop. In every following commit the PageRank values of the graph are only recomputed partially:

In the pseudo code below the incremental PageRank algorithm is displayed as a function called *incrPageRank*. As input parameter the function takes the dependency graph for which the PageRank has to be computed. To determine which vertices have

changed, the function also needs the sets of deleted and added edges as well as the set of added vertices. The *scalingFactor* is computed with the formula mentioned in Section 4.2.1. At first one has to specify which PageRank values will change due to changes in the dependency graph. If there are no changed vertices found, nothing will happen to the PageRank values and therefore the old PageRank values will be reused for this commit. For every edge addition as well as edge deletion (I,J) the vertices I and J are added to the changed list (Lines 2-5 in the *incrPageRank* function). For every vertex addition the added vertex is also added to the changed list (Line 6 in the *incrPageRank* function). After that all vertices affected by the changed vertices are searched as described by Desikan et al. [6]. But different from this, the vertices are split up into two sets in our case: vertices which have no successor (*danglingVertices*) and vertices which have successsors (*recalucalteVertices*) (Lines 12-25 in the *incrPageRank* function). The set of dangling vertices is treated differently, because these PageRank values will not influence the PageRank value of other vertices. Therefore, the PageRank value of dangling vertices does not have to be computed iteratively, but only once after all other vertices get assigned their PageRank values. All nodes that already existed in the previous commit are then getting assigned their old PageRank value times the *scalingFactor* already mentioned above. All added vertices get assigned an initial value (Lines 26-32 in the *incrPageRank* function):

$$initialValue = \frac{1}{total\ number\ of\ vertices}$$

After the iterative PageRank algorithm is applied to the set of vertices that have to be recalculated (Line 33 in the *incrPageRank* function), the last step is to once recompute the PageRank value for all dangling vertices (Lines 34-36 in the *incrPageRank* function), because their PageRank values do not have any successive nodes of which they could influence the PageRank value.

```
1:  function IncrPageRank(Graph, newVertices, newEdges, delEdges, scalingFactor)
2:      for each edge (I,J) in newEdges or delEdges do
3:          changedVertices.add(I)
4:          changedVertices.add(J)
5:      end for
6:      changedVertices.addAll(newVertices)
7:      for each vertex N in Graph.getVertices() do
8:          if not(changedVertices.contains(N)) then
9:              unchangedVertices.add(N)
10:         end if
11:     end for
```

```
12:     while not(changedVeritces.isEmpty()) do
13:         N ← changedVertices.pop()
14:         if N.getChildren().isEmpty() then
15:             danglingVertices.add(N)
16:         else
17:             for each vertex C in N.getChildren() do
18:                 if unchangedVertices.contains(C) then
19:                     unchangedVertices.remove(C)
20:                     changedVertices.add(C)
21:                 end if
22:             end for
23:             recalculateVertices.add(N)
24:         end if
25:     end while
26:     for each vertex N in recalculateVertices, danglingVertices, unchangedVertices do
27:         if N.pageRankValue.exists() then
28:             N.pageRankValue ← N.pageRankValue · scalingFactor
29:         else
30:             N.pageRankValue ← 1 ÷ Graph.getVertexCount()
31:         end if
32:     end for
33:     IterativePageRank(Graph, recalculateVertices)
34:     for each vertex N in danglingVertices do
35:         PageRank(Graph,N)
36:     end for
37: end function
```

4.3 Approximated Incremental PageRank

In addition to the already presented incremental PageRank algorithm, there is another possibility to improve the efficiency of our approach. The idea for this improvement is taken from Chien et al. [7] and combined with the already presented approach.

4.3.1 Which values have to be recomputed and why?

The values that have to be recomputed are very similar to those of the purely incremental approach. The only set that changes is the set of affected vertices. In the approximated incremental algorithm not all subsequent vertices of the changed vertices are added to the set of affected vertices. All changed vertices get assigned the weight

1. All subsequent vertices also get weights assigned, but they are getting smaller and smaller depending on both the number of other child nodes of their parent node and the distance to the changed vertex. All vertices which have a weight that is bigger than a certain threshold will be recalculated and the others will keep their old values only being scaled if the total number of vertices changed. Therefore, these vertices will be treated as unaffected even though they can be reached from a changed vertex. The formula for calculating the vertex weights is taken from Chien et al. [7]:

$$Weight(Vertex) = \frac{Weight(ParentVertex) - d}{OutDegree(ParentVertex)}$$

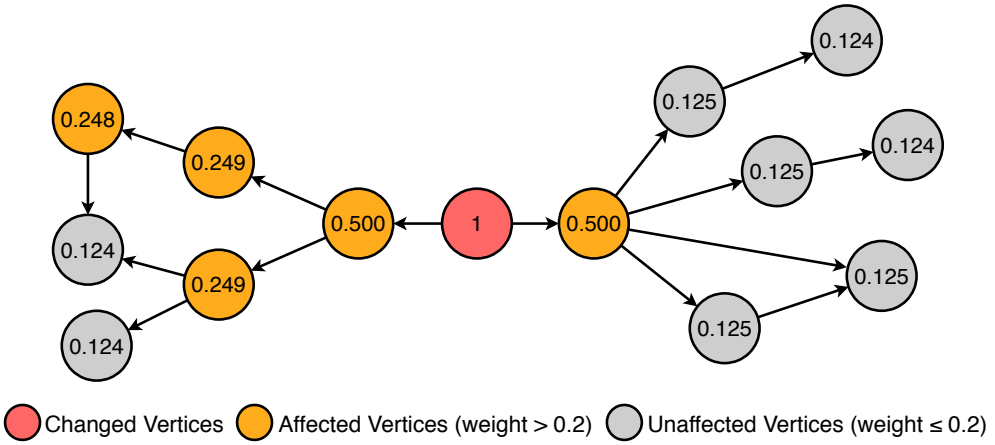

Figure 4.3: Example of a dependency graph with weights and a tolerance of 0.2

The dependency graph in Figure 4.3 displays how the weight assignment works. The changed vertex gets assigned the weight 1.The two children of the changed vertex get assigned the approximated weight of 0.5, because d has a value of 0.001 in our case. Now there can be seen a big difference between what is happening next on the left and on the right side of the changed vertex. On the left side the vertex with weight 0.5 only has two children, so they both have a higher value than the four children of the vertex with weight 0.5 on the right side of the changed vertex. On the left side there can be seen three affected vertices in a row that all have a weight higher than the chosen tolerance of 0.2. On the right side the weight is already smaller than 0.2 after one step. Therefore, Figure 4.3 shows very well that both the distance from the changed vertex and the *OutDegree* of the parent node influence the weight of a vertex. The assumption that was made by Chien et al. is, that the smaller the weight is, the less their PageRank value is influenced by the changed vertex [7].

The pseudo code below shows the approximated incremental PageRank algorithm, which is very similar to the incremental PageRank algorithm. The only section that is different from the incremental PageRank is the one where the affected vertices are computed (Lines 12-37 in the *approxIncrPageRank* function). In the lines 12-15 every changed vertex gets assigned the weight 1. After that all subsequent children are added to the changed vertex list, if their weight is greater than the tolerance (lines 23-27 in the *approxIncrPageRank* function). Lines 28-32 deal with the special case that the child node has already been removed from the unchanged vertices set. If the new weight is greater than the old weight of the vertex, the vertex gets assigned the new value. If the vertex is already removed from the changed vertices set it is added again, because all subsequent vertices will also have a higher weight, which may be greater than the tolerance.

```
1: function approxIncrPageRank(Graph, newVertices, newEdges, delEdges, scaling-
   Factor, tolerance)
2:     for each edge (I,J) in newEdges or delEdges do
3:         changedVertices.add(I)
4:         changedVertices.add(J)
5:     end for
6:     changedVertices.addAll(newVertices)
7:     for each vertex N in Graph.getVertices() do
8:         if not(changedVertices.contains(N)) then
9:             unchangedVertices.add(N)
10:        end if
11:    end for
12:    for each vertex N in changedVertices do
13:        N.setWeight(1)
14:    end for
15:    while not(changedVeritces.isEmpty()) do
16:        N ← changedVertices.pop()
17:        if N.getChildren().isEmpty() then
18:            danglingVertices.add(N)
19:        else
20:            for each vertex C in N.getChildren() do
21:                weight ← calculateVertexWeight(N);
22:                if unchangedVertices.contains(C) then
23:                    if weight > tolerance then
24:                        C.setWeight(weight)
```

```
25:                         unchangedVertices.remove(C)
26:                         changedVertices.add(C)
27:                     end if
28:                 else if C.getWeight() < weight then
29:                     C.setWeight(weight)
30:                     if not(changedVertices.contains(C)) then
31:                         changedVertices.add(C)
32:                     end if
33:                 end if
34:             end for
35:             recalculateVertices.add(N)
36:         end if
37:     end while
38:     for each vertex N in recalculateVertices, danglingVertices, unchangedVertices do
39:         if N.pageRankValue.exists() then
40:             N.pageRankValue ← N.pageRankValue · scalingFactor
41:         else
42:             N.pageRankValue ← 1 ÷ Graph.getVertexCount()
43:         end if
44:     end for
45:     IterativePageRank(Graph, recalculateVertices)
46:     for each vertex N in danglingVertices do
47:         PageRank(Graph,N)
48:     end for
49: end function
```

The function *calculateVertexWeight(parentVertex)* is called by the *approxIncrPageRank* function above and contains the formula by Chien et al. [7] reformulated in pseudo code. Because the formula does not need any information of the vertex itself it just gets called with its *parentVertex* as parameter.

```
1: function calculateVertexWeight(parentVertex)
2:     weight ← (parentVertex.getWeight() - 0.001) ÷ parentVertex.getOutDegree()
3:     return weight
4: end function
```

5 Evaluation

In the evaluation we want to measure at first the runtime improvement of the incremental approaches. We use different software projects that vary in size, programming language and functionality. After checking the runtime, we evaluate how precise the computation is by comparing the rankings of the standard with the incremental PageRank by utilizing the Spearman's rank correlation metric. Finally, we apply the incremental approaches to problems of software analysis and compare the results with the standard PageRank.

5.1 Research Questions

In this bachelor's thesis the following research questions will be proposed and their results are presented and discussed.

RQ1: How much faster is an incremental approach compared to the standard PageRank algorithm? The first question deals with the runtime improvement of an incremental approach. We developed two algorithms for this evaluation: One of them is incremental and the other one approximates the incremental approach. We compare the incremental algorithm and several different approximated incremental approaches with different thresholds in terms of their runtime.

RQ2: How much do the incrementally calculated ranks deviate from the standard PageRank algorithm? It is not enough to have a runtime improvement. More importantly, the approximated incremental values need to be close to the exact ones. This research question investigates deviations when calculating PageRank values in an incremental software analysis using spearman's rank correlation metric.

RQ3: How much do the results of code centrality analysis based on PageRank deviate when applying an incremental approach? The first two questions deal with the runtime improvement as well as with the accuracy of the values. Now, the only thing left to discuss is, whether this is now truly applicable in the software analysis field or not. To find out if that is the case we apply our approach in two different use cases, namely the unnecessary code identification [4] and the central class recommendations [3].

5.2 Study Objects

We evaluate our approach with different source code projects which are mostly taken from Github. The chosen projects are: Teamscale[1], Jabref[2], MySQL Server[3], Mozilla[4], Apache Ant[5] and an anonymous project. We selected the projects such that we get different programming languages and different project sizes. To assure the quality of the software projects we chose projects which are well-known and highly rated on GitHub. Two projects are closed source and the development teams of these projects provided both their source code and their history to us so that we can also evaluate our work on these projects. We used projects of different sizes to be able to make statements that apply both for smaller projects like Jabref or Apache Ant, as well as for larger projects like MySQL Server the anonymous project or Mozilla. To have a decentral type of software structure in our study, we included the Apache Ant library project. Library projects have a decentral structure, because they consist of several independent functionalities which are independent of each other. This leads to a low-coupled dependency graph. To determine whether our approach is restricted to one programming language or not, we chose projects written in different languages: Java, C# and C++. We also display the time span for which we evaluated the projects. The time spans vary due to the variation in the number of commits per time unit. For example, it took an analysis time span of 8 years for the Apache Ant project to get 850 commits whereas the Mozilla project had more than 2000 commits in 2 months. Especially the huge projects with many lines of code tend to have a much higher amount of commits during a smaller time span than the small projects.

Project	Lang.	SLOC	Commits	Time span
Jabref	Java	121,600	1,700	3 years
Apache Ant	Java	140,300	900	8 years
Teamscale	Java	513,300	1400	1.5 years
Anon. Project	C#	914,800	4,000	1 year
MySQL Server	C++	2,315,600	3,400	2 years
Mozilla	C++	7,704,100	2,400	2 months

Table 5.1: Metrics of the study objects

[1]https://www.cqse.eu/en/products/teamscale/landing
[2]https://github.com/JabRef/jabref
[3]https://github.com/mysql/mysql-server
[4]https://github.com/mozilla/gecko-dev
[5]https://github.com/apache/ant

In this bachelor's thesis, we deal with the change of the projects during a chosen time span. To get an impression of how big this change is, we present in Table 5.2 the amount of classes and dependencies for the first commit analyzed (First Commit column) and for the last commit (Final Commit column). The history analysis tool we use, called Teamscale, splits up commits with a huge number of changes into several smaller analysis steps to improve the runtime. Therefore especially the first commit is split into several analysis steps. The analysis of the project starts with a small portion of the software project and after every analysis step another portion is added to the already existing one. This process continues until the whole project is analyzed. Therefore, the analysis tool has to deal with a huge amount of change per analysis step until the first commit is completely analyzed. For example the Mozilla project needs about 100 analysis steps to get from 0 classes to 28,000 classes. After that, there are only added 400 new classes during the whole analyzed time span. The last column shows the average amount of classes that are changed in one analysis step relatively to the total number of classes. As stated above, the analysis until the first commit is finished, is just an implementation detail of Teamscale and thats why we did not take those analysis steps into account when measuring the average amount of change. We still mentioned this implementation detail of Teamscale, because it will explain some behaviors of the PageRank values in our explanation. The reason why the average changes are proportionally higher compared to the total amount of classes for the first three projects is that in these cases there exist a smaller amount of total classes. Nevertheless, one can see that the amount of changes is especially for large projects relaitvely small compared to the total number of classes in the project.

	First Commit		Final Commit		
Project	Classes	Deps.	Classes	Deps.	Avg. Change [%]
Jabref	700	3,800	1,500	7,500	5.96%
Apache Ant	1,200	5,000	1,300	5,500	1.74%
Teamscale	7,700	46,700	6,500	43,000	1.14%
Anon. Project	10,000	56,500	11,000	63,400	0.48%
MySQL Server	5,400	21,000	8,000	43,100	1.00%
Mozilla	28,300	122,500	28,700	124,300	0.12%

Table 5.2: Metrics of the study objects showing the change during the time span displayed in Table 5.1

5.3 Study Design

The basis for all three research questions is the history analysis of the software projects. This analysis contains the computation of the dependency graph of every commit. Based on this information the different incremental PageRank algorithms presented in Chapter 4 can be performed. In the first step the PageRank is computed normally. In every following step there is not only computed the standard PageRank, but the two different proposed incremental algorithms as well. In total there are 8 different incremental approaches computed per step: The purely incremental one and 7 different approximated incremental approaches with different thresholds. The chosen thresholds are: 0.001, 0.01, 0.05, 0.1, 0.2, 0.5 and 1.0. These values are getting stored for each approach and are reused in the following analysis step. Every approach reuses the values of their own previous computation.

For the standard PageRank values we rely on a slightly adjusted version of the library of Jung to have a reasonable basis to which the incremental values can be compared to. The small adjustments were only made to guarantee the applicability of the approaches from the papers of Desikan et al. [6] and Chien et al. [6] and do not falsify the results. The parameter dealing with the disappearing potential is used in the Jung library but is left out in both of the used papers. That is why we left this factor out as well.

In the following, we describe our study design to answer the research questions.

5.3.1 RQ1: How much faster is an incremental approach compared to the standard PageRank algorithm?

To get the runtime improvement of the different incremental approaches, we measure the time needed to incrementally compute the new PageRank values for each analysis step and for each PageRank approach. The time needed for the standard PageRank computation is also measured for each analysis step as a reference value. The measured time contains the computation of the affected vertices, the assignment of the values to the vertices and the partial recomputation of PageRank. In case of the standard PageRank the measured time contains solely the computation needed for the PageRank algorithm. The additional computations of the incremental approaches which are not needed by the standard PageRank are also considered for comparing the standard PageRank to the incremental approaches. This makes the results more comparable than only taking the PageRank computation time and the partially recomputation time of the incremental approaches into account.

5.3.2 RQ2: incrementally calculated ranks deviate from the standard PageRank algorithm?

In the software analysis applications that use the PageRank, the actual PageRank values themselves are not that important. The PageRank values will be transferred to rankings. This means that the incremental approaches do not need to have the same values as the standard PageRank. The only thing that matters are the ranks that the incremental PageRank values produce. The ranks of the standard PageRank and the incremental PageRanks should be as similar as possible. For each incremental PageRank approach the spearman's rank correlation metric is computed with the standard PageRank ranks as target values to measure the equality of the rankings.

5.3.3 RQ3: How much do the results of code centrality analysis based on PageRank deviate when applying an incremental approach?

To find out, how useful the incremental approach is in software analysis, the incremental PageRank approaches are applied to the already mentioned problems central class recommendation and unnecessary code identification. In the case of the central class recommendations we compare the top 10 classes of each incremental approach with the ones of the standard PageRank. To measure the applicability of the incremental approaches for the unnecessary code identification we compare the results the approach of Haas [4] returns for the standard PageRank algorithm and the different incremental PageRank algorithms.

5.4 Results

For every research question we collected the runtimes and the rankings for all different incremental PageRanks as well as for the standard PageRank in each analysis step and in the following, we present the results.

5.4.1 RQ1: How much faster is an incremental approach compared to the standard PageRank algorithm?

In Figure 5.1 the time needed for the whole PageRank computation of all analysis steps is displayed for every PageRank approach. To make a comparison between the different projects easier we did not show the absolute time needed in seconds. The runtimes of the different projects differ a lot depending on the size of the projects as well as on the number of commits that were analyzed during the selected time spans. To get rid of all these differences, we present the runtimes proportional to the time needed

by the calculations of the standard PageRank. This means the standard PageRank runtime is always set to 100% as reference value and the other runtimes are presented proportionately to this one.

The figures show that the different incremental approaches lead to a runtime improvement. This improvement is different for the projects. The incremental PageRank without any approximation takes the longest. For the MySQL Server project the incremental PageRank needs 80% of the original PageRank time and in the Apache Ant project and the Mozilla project, it is even worse than the standard PageRank runtime. This shows how important it is to approximate the incremental computation. With an approximation threshold of 0.001 the computation is already 3 times faster than without any incremental approach. The runtime difference between the thresholds is the smallest for Jabref and for the Apache project.

In the case of a much bigger project like the MySQL Server or Mozilla, the computation of the affected nodes is not too time consuming compared to the time needed by the standard PageRank, at least when looking at the approximated PageRank values in Figure 5.1. That the size of the project influences the runtime improvement of different thresholds can be seen when comparing Jabref and MySQL Server. Whereas a threshold of 0.05 is in the case of MySQL Server already big enough to make it 10 times faster than the original PageRank, the threshold for Jabref needs a significantly less precise threshold of 0.2 to make the computation 10 times faster which can also been seen in Figure 5.1.

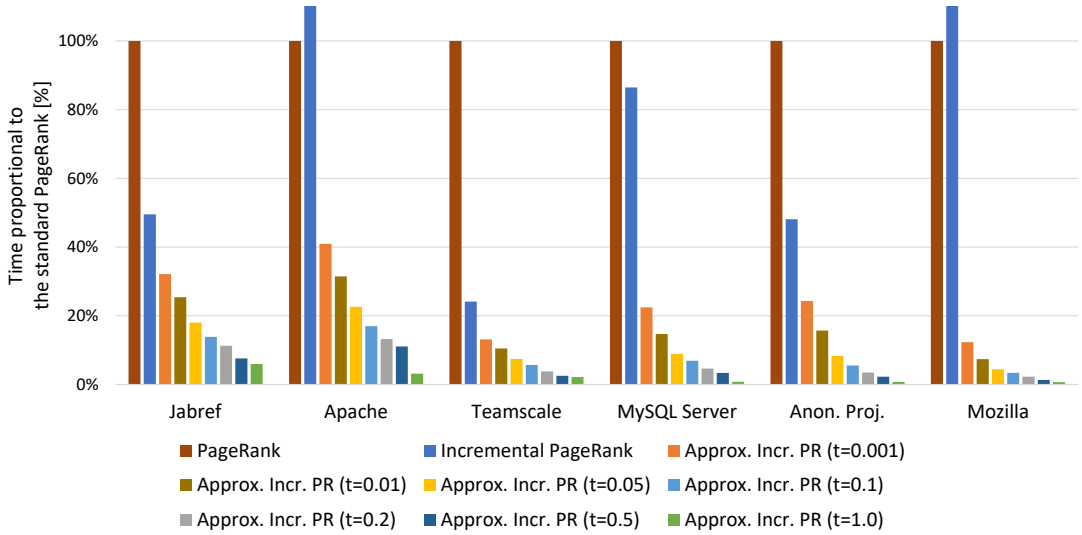

Figure 5.1: The summed up runtime of the different PageRank approaches

To get a better feeling for the runtimes of the single analysis steps, Figure 5.2 and Figure 5.3 each display 500 commits for Jabref and MySQL Server, respectively. Here again, we did not choose the absolute values, but the runtimes proportional to the standard PageRank. That is why the standard PageRank has always a value of 100%. The figures show how much the incremental approaches improve the runtime. The higher the threshold of the approximated incremental PageRank, the better the runtime. Nevertheless there are still some commits where the runtime of the PageRank gets closer to the runtime of the standard PageRank, but mostly the improvement is very high. In the case of the incremental PageRank in Figure 5.3 this is not the case, but it is already noticeable in Figure 5.1 that the runtime of the incremental approach is not significant better for the MySQL Server project.

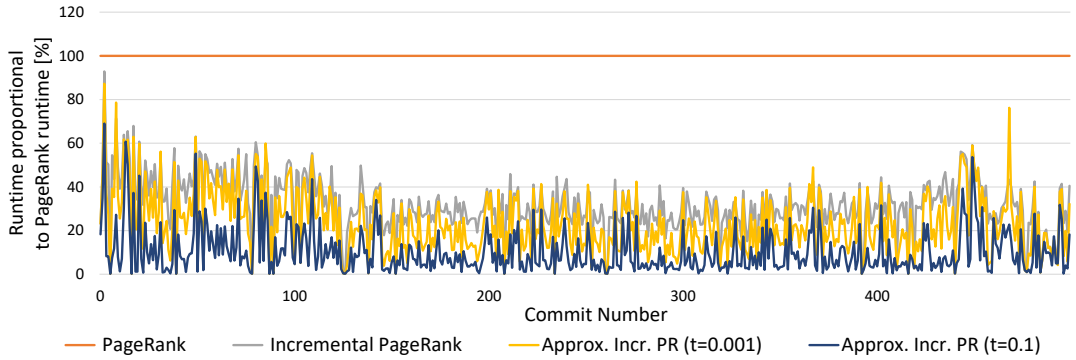

Figure 5.2: The Runtime of Jabref for every commit

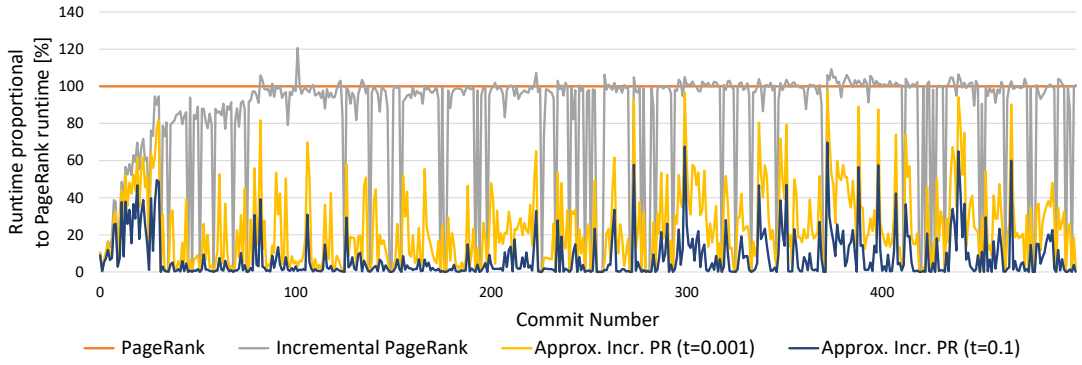

Figure 5.3: The Runtime of MySQL Server for every commit.

The two figures also show that the runtime improvement of the approximated

PageRank is higher for the MySQL Server project than for the Jabref project compared to the standard PageRank times.

In Figure 5.4 the different amounts of classes that have to be recomputed on average are displayed. Due to comparison reasons we again do not use the absolute values. In this case we use the amount of classes proportional to the total number of classes in each project. When comparing the different projects with each other it can be seen that the relative amount of classes that have to be recomputed is higher for the smaller projects than for the big projects. Therefore, especially the improvement of the approximated incremental PageRank approaches is getting higher with an increasing size of the projects which can be also seen in Figure 5.1. The improvement of the incremental PageRank is not necessarily getting higher depending on the size of the project. A threshold of 1.0 of the approximated incremental PageRank means that only the changed values and no affected values at all are getting recomputed.

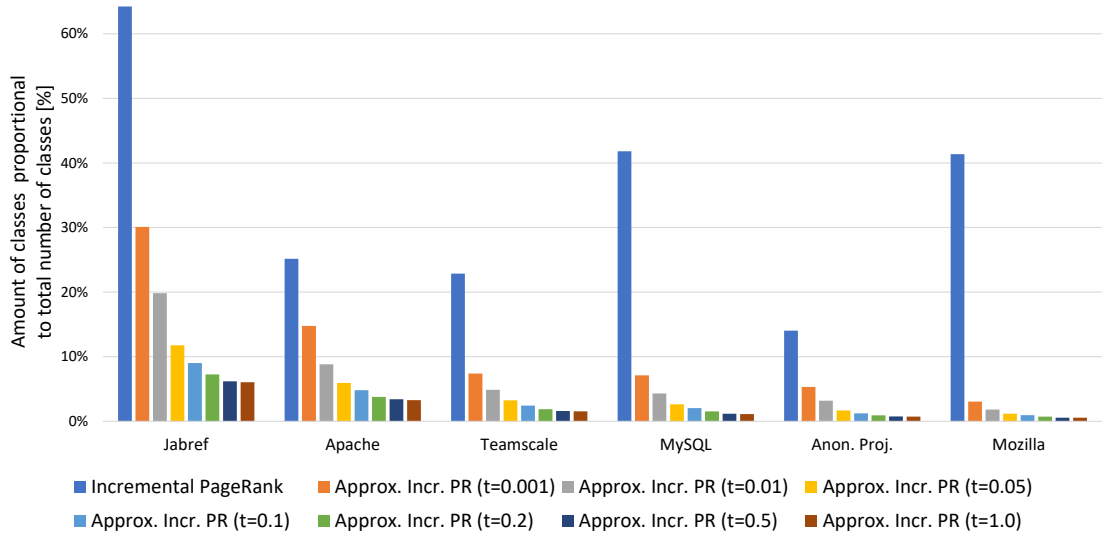

Figure 5.4: The amount of classes that get recalculated

5.4.2 RQ2: How much do the incrementally calculated ranks deviate from the standard PageRank algorithm?

In Figure 5.5, the spearman's rank correlation metric is displayed to measure the equality of the ranks of the incremental and the standard PageRank. This metric was computed for all projects after analyzing the given number of commits for all incremental approaches, captured in Table 5.1. The best result in terms of accuracy is

reached with the Jabref project. For the Apache Ant project the spearman's metrics are the worst.

Another interesting aspect is the approximated incremental PageRank with a threshold of 1.0. The only values that get recomputed in that case are those which have been changed since the last commit. This means that no affected values are recomputed. For the Jabref project this is still a metric with comparatively good results. It is not much worse than the approaches with smaller thresholds that also recompute affected classes. When looking at the bigger projects like Mozilla one can see that there is a significant loss of accuracy with a threshold of 1.0 as well as for a threshold of 0.5. This shows that the selection of the threshold is not independent of the size of the project. The bigger the project, the smaller the threshold has to be to still get sufficiently exact values. This is the case, because the number of recalculated vertices is in both cases very similar for the approximated incremental PageRank. This makes the proportional amount of recalculated vertices smaller for the bigger projects and leads to a smaller overall precision for the bigger projects if the threshold gets too high.

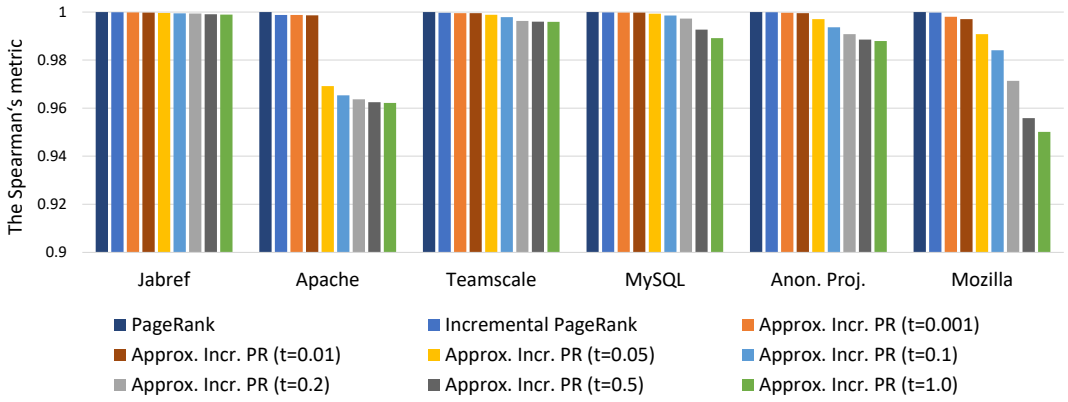

Figure 5.5: The spearman's rank correlation metric for all projects.

To get a better feeling for the spearman's metric, it is also interesting to look at a particular ranking of one project. In Figure 5.6 the ranks of the different PageRank approaches are displayed for one particular commit of Teamscale. This is the last commit analysed in the history analysis, so before this ranking is reached there were about 1400 commits already computed incrementally. For every PageRank approach the classes were sorted and ranked according to their PageRank values. The x-axis of the Figure 5.6 shows the number of different classes which are sorted according to their standard PageRank ranks. The purely incremental PageRank ranks are almost identical to the standard PageRank ranks. The only difference is, that at rank 4000 the

incremental PageRank ranks start to grow faster, but the sortation of the ranks is not lost. The incremental PageRank and the approximated incremental PageRanks with a threshold of 0.001 and 0.01 are very close to the purely incremental line, with only a few ranks that are scattering. We did not display these here, because one can not differentiate between these and the incremental PageRank. For every higher threshold like 0.05 or even 0.2, the scattering of the values is getting more and more, but still the number of scattering values is very small compared to the total number of classes ranked.

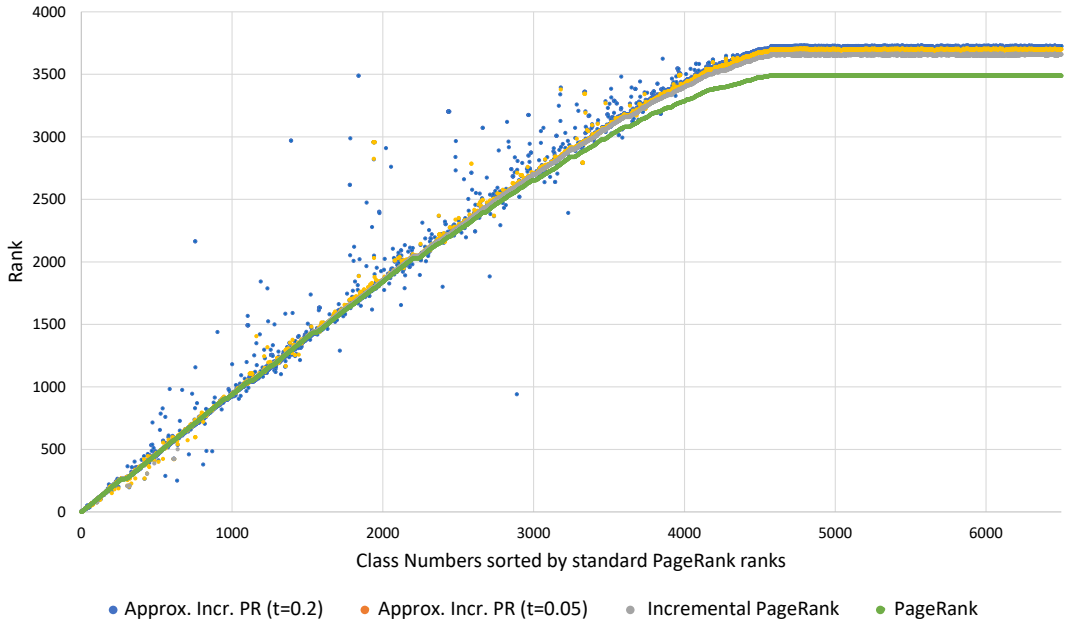

Figure 5.6: The ranks of the classes of the Teamscale project.

Figure 5.7 presents how the spearman's rank correlation evolves over two years. The commits were selected from the MySQL Server project to analyze a project with an average size compared to the other study objects. The first noticeable thing is the significant drop in the figure that gets bigger with an increasing size of the approximation. After this initial drop all metrics grow again and start to converge to a specific value. There is still a very small decline noticeable, but considering the overall time of 2 years and a decline of less than 0.001 during this timespan this can be neglected. The metrics of the incremental approaches with a threshold of 0, 0.001 and 0.01 can not be easily identified, because they are all very close to 1.

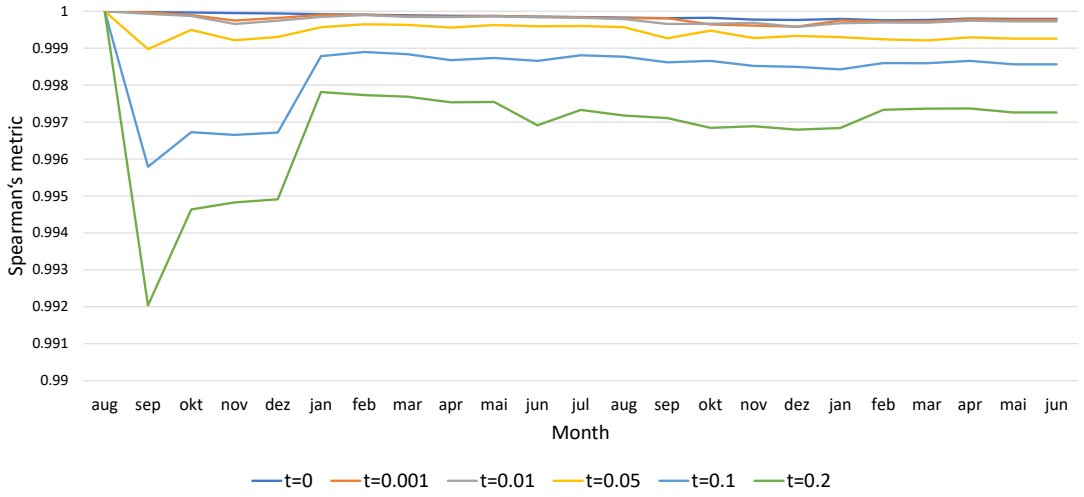

Figure 5.7: The spearman's metric of the incremental approaches of the MySQL Server project over time.

5.4.3 RQ3: How much do the results of code centrality analysis based on PageRank deviate when applying an incremental approach?

Central Class Recommendations

The central class recommendations problem of Steidl et al. [3] show the importance of an incremental PageRank approach, because otherwise the PageRank values would be recomputed for every commit from scratch which leads to high computational costs. In Figure 5.8 the 10 most important classes of the incremental approach with a threshold of 0.01 are compared to those of the standard PageRank. The green bars are representing the amount of classes that have the same rank as in the standard PageRank case. The yellow bars represent those classes which do not have the same rank, but are still part of the ten classes identified as most important. The amount of classes that is not in the top 10 anymore is visualized by the red bars. When leaving out the Apache Ant project which already lead to bad results in RQ2 one can see that with an increasing size of the project the result is getting worse. In the case of Jabref all classes are found and even have the right rank. When looking at the Teamscale project, there are two classes which have the same rank as in the standard PageRank case. Though, in total there is no class that does not belong to the most central classes, which is still good. For the Mozilla project, we have 2 classes that do not belong to the 10 most central classes and only two classes are left that have the same rank.

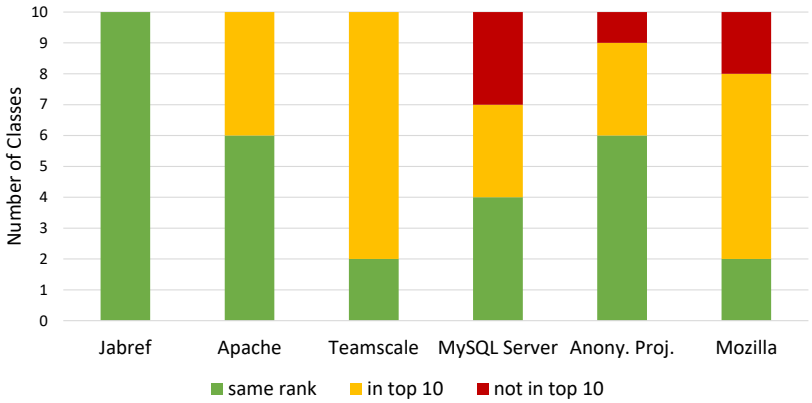

Figure 5.8: The 10 most central classes of the software projects.

Unnecessary Code Identification

The unnecessary code recommendation of Haas [4] is another problem where the incremental PageRank approaches would benefit in terms of runtime. The runtime improvement was already described in RQ1, but the precision of the result has to be validated as well. In this case, there are not recommended single classes, but buckets of classes. Each buckets contains classes of the same directory. In total, there are always 10 buckets recommended and in the Figure 5.9 one can see how many of those are found utilizing an incremental approach instead of the standard PageRank.

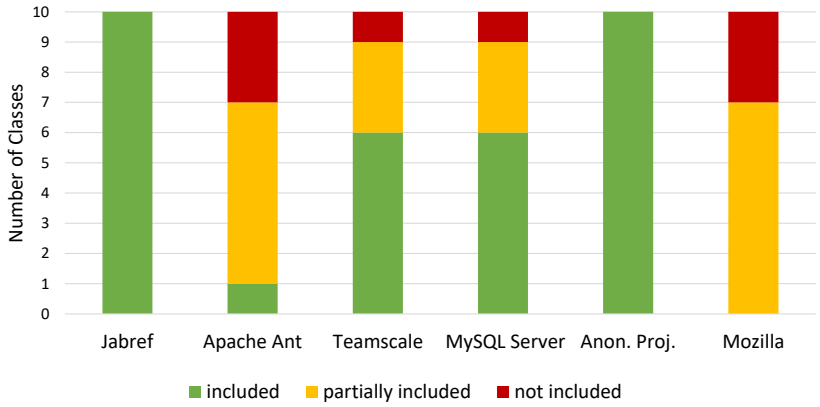

Figure 5.9: The unnecessary code buckets recommended utilizing the incremental PageRank.

The green section displays the buckets that were found. The yellow section represents those buckets which are only partially found and the red ones are the ones which were not recommended at all. A very interesting observation is, that every incremental approach returns the same results for every project. This is why we do not have to specify which incremental approach we used in this case. The best results are scored again with Jabref. With an increasing size of the project, the results are getting worse but the amount of wrong buckets still stays small. For the Mozilla project there was no bucket completely found, but still there only 3 buckets wrong. The only project that does not fit in is Apache Ant. In that case the most wrong buckets are recommended.

5.5 Discussion

5.5.1 RQ1: How much faster is an incremental approach compared to the standard PageRank algorithm?

The improvement of the PageRank computation varies depending on the chosen threshold. The incremental PageRank without any approximation takes by far the longest, because the most values have to be recomputed, which can be seen in Figure 5.4. In this case, all successive nodes are marked as affected which leads in a highly coupled project to a recomputation of almost all values. For example, for MySQL Server more than the half of the classes are recomputed on average, which is a lot, considering that only 1% of the total classes are changed on average. The runtime improvement of the small projects is the smallest, because the original PageRank does not take too much time in that case. This makes the incremental approaches less effective due to the additional time they need for computing the affected vertices. For a sufficiently large dependency graph the additional costs of the incremental approaches are negligible compared to the amount of time the original PageRank needs. But this is only the case for the approximated approaches. The incremental approach without approximation needs to recompute too many vertices to make it effective and also the time needed to traverse half of the graph to assign 50% of the graph as affected takes a long time. If only less than 10% of the total classes in MySQL Server have to be recomputed instead of 30% in Jabref, as it is the case for the approximated approach with a threshold of 0.001, this leads to different improvements of the runtime.

In Figure 5.2 and in Figure 5.3 there are still some commits where the runtime of the PageRank is closer or even worse than the standard PageRank computation. Especially for the MySQL Server, the incremental approach is often close to the standard PageRank time. This is the case due to the amount of changes that have to be considered in that specific commit.

The curves of the incremental and approximated incremental approaches are closer

to each other in the Figure 5.2 of Jabref, because the amount of classes that have to be recalculated are also closer to each other (displayed in Figure 5.4). The approximated incremental PageRank with a threshold of 0.001 recalculates half of the classes that the incremental PageRank needs in that case, whereas the incremental PageRank needs 4 times the amount that the approximated approach needs for the MySQL Server project.

5.5.2 RQ2: How much do the incrementally calculated ranks deviate from the standard PageRank algorithm?

The accuracy of the incremental approaches are not independent of the size of the analyzed software project. The best results are scored for the Jabref, due to the comparatively small project size. The Apache Ant project is also small, but in that case the structure of the software project is not ideal for an incremental PageRank computation. In contrast to the other projects, the Apache Ant project is a library and the dependency graph of a library consists typically of several independent subcomponents. This is the case, because a library always provides several different and independent services to the user. This decentral structured software project shows, that the incremental approach is not applicable for all types of software projects.

In Figure 5.5 the different incremental PageRank ranks start to grow faster after the 4000th rank. That is the case due to very small computation errors that occur in the incremental algorithm. If two values have the same value in the standard PageRank approach, there can be already a very small rounding error, which leads to two ranks instead of one when transferring the PageRank values to ranks. Over time the amount of additional ranks gets bigger and that is why the incremental PageRank ranks start to grow faster after a certain amount of time.

In Figure 5.7 the spearman's rank correlation metric is displayed for several consecutive months. At first the metric gets clearly worse, especially for the higher approximation levels. The reason for this is the way the history analysis is implemented in Teamscale, the tool used to analyze the projects. Due to runtime reasons the first commit gets split into several analysis steps and these analysis steps are already computed incrementally after the initial analysis step. This leads to a huge amount of change for every analysis step during the first commit. As already stated, the amount of change drops significantly and the incremental approaches start to deal with smaller portions of the graph and can improve the errors made during the time the changes were as high as in the initial commit phase. The reason for this is that the incremental approaches from the literature were originally designed for only a small amount of changes per time step. Adding a relative high amount of changes for several commits in a row leads to high computational errors. In the case of the Mozilla projects, where the initial commit needs 130 analysis steps to analyze the whole system, there are on

average 10% of all classes modified per step. After the first commit is analyzed, the average rate of changed classes compared to the total number of classes reduces to an average of 0.1%.

After the initial commit the relative amount of changed classes gets significantly smaller. Thus, only small parts of the graph are updated which can correct the computational errors of this part of the graph made during the initial analysis phases of the first commit due to the recomputation.

After the errors are corrected, the metric values converge to different values and do not fluctuate significantly. This shows that the incremental approximations are very stable and can even correct errors made. It can also be seen that the incremental approaches are not applicable, if the amount of change is constantly that high. This behavior is not common for a single commit and even the high amount of change is only due to the implementation used and can be adapted such that we start with the incremental approach not before the initial analysis is finished.

5.5.3 RQ3: How much do the results of code centrality analysis based on PageRank deviate when applying an incremental approach?

Central Class Recommendations

Having not the exact same rank as in the standard PageRank case is a common issue for most of the projects. Since the exact rank of the most central classes is not important, it is still sufficient, if the classes are contained in the 10 most central ones, which is mostly the case. Nevertheless, there exist some wrong classes which should not be contained in the 10 most central classes. Though, considering the 8,000 classes that for example the MySQL Server project contains in total, a success rate of 70% is still very good. Besides, the ranks of the missing three classes are very close to their actual value compared to the total amount: The incremental approach with a threshold of 0.01 assigns those the ranks 14, 19 and 23. This makes it precise enough, because there were 7 out of 10 classes found.

Unnecessary Code Identification

The different incremental approaches return all the same results. The most decentral classes are those classes which are not highly coupled to other classes and the most stable classes are those who are not changed very often. Both, the decentrality and the stability lead to the fact that these classes are never or at least not often reached by the incremental approaches. This is the case, because those approaches only recompute changed vertices and vertices affected by those changes. Due to the stability

of unnecessary classes they are not changed often and due to their decentrality they are rarely part of the affected vertices.

The recommendation of the partial buckets seems bad first, but it still leads to the directory where the unnecessary code was found. This makes it more likely that the programmer will find those classes not recommended by the incremental approach.

5.6 Threats to Validity

5.6.1 Internal Validity

The implementation of the incremental PageRank depends on the correct calculation of the changes made to the dependency graph of the software system and the calculation of the dependency graph itself on the one hand. On the other hand it relies on the correct implementation of the used PageRank library. For the calculation of the dependency graph we use a mature software analysis system, called Teamscale[6] , which makes it very reliable. The used library is the JUNG library[7], the Java Universal Network/Graph framework. This library is often used in software projects, which makes it also very reliable and the fact that it is open source made it easy to check if the internal computations are correct.

5.6.2 External Validity

The generalization of a software analysis related project is often very difficult. This is also true for this analysis. Even though different sizes and languages of software projects were considered and both closed- and open-source project were taken into consideration, it can still not be generalized for all possible software projects [13]. The evaluation has shown that the different incremental approaches indeed lead to a runtime improvement whereas the values are still very precise, depending on the chosen threshold. But there was already one software type, the library project, where the precision got drastically worse with an increasing threshold, even though the project itself was very small. Therefore, library projects are already a bad candidate for the incremental approach. However, we think that a software project with a highly coupled dependency graph leads to good results.

[6]https://www.cqse.eu/en/products/teamscale/landing
[7]http://jung.sourceforge.net

6 Conclusion

6.1 Summary

Centrality metrics are not only used for the computation of central pages of the Web graph for search engines, but also for different centrality related topics in software analysis, for example, the recommendation of central software classes and the identification of unnecessary code. Both fields use a graph-based approach for the computation of the centrality metrics, which makes the computation very time-consuming for big graphs. In the first commit the centrality metric has to be computed for the whole graph, but we applied incremental approaches for every following commit due to the relatively small number of changes that occur. The same holds for Web graphs. The amount of change per time step is also very small. In Web graphs there were already many incremental approaches presented and we applied these incremental approaches to the dependency graph of software systems. We computed the nodes that are affected by the change and also applied different levels of approximation. The idea was, that all successor nodes of the changed nodes are those truly affected by the change. We defined a node to be changed, if it has been added to the graph or a dependency between this and another class has been deleted or added. This purely incremental approach results only in a little runtime improvement, because in a highly connected graph this leads to a recomputation of nearly the whole graph. To improve our approach, we combined two approaches from the literature and took the incremental approach from one paper and the approximation formula from the other to gain better results.

We measured the runtime improvement, as well as the precision loss we get for the different incremental approaches with a variation of the approximation. We evaluated both, the runtime metric and the spearman's rank correlation metric, on six different projects of different size, language and purpose.

We investigated three different research questions related to the applicability of the incremental approach in the software analysis field. In the first research question we discovered that there is indeed a runtime improvement if applying an approximated incremental approach. For the incremental approach, there were projects where the overall runtime was even worse than for the standard PageRank. Adding a small approximation to the incremental approach improved the runtime for all sizes and types of projects. The approximated incremental approach with a threshold of 0.01

made the overall runtime at least 3 times better for all analyzed software projects. With the Mozilla project and the Teamscale project, we encountered the best runtime improvements, because they only needed 12% and 13% of the time needed for the standard PageRank calculations.

In the second research question, we dealt with the precision loss with the help of the spearman's rank correlation metric. The result was that there was definitely a loss of information, but this loss was very small compared to the amount of classes that each dependency graph contains. However, we also found out that the incremental approach is not applicable for every type of software project. The library project Apache Ant showed a significant loss of precision for approximation levels, whereas the precision of much bigger projects was still fine. The reason for this is the decentral structure of library projects in general, because the dependency graphs consist of several completely independent subgraphs due to the different independent functionalities a library provides.

The last research question examined how applicable the incremental approaches are for the two discussed software analysis problems. For both problems, the application of an incremental centrality calculation showed promising results. For the most central classes, there were some classes that did not have the exact same rank than they had at the standard PageRank. Still, they mostly were contained in the top 10 classes and even if they were not under those, they were not far away from their correct rank: They were always under the 25 top ranked classes. The unnecessary code identification problem also showed good results. There were mostly only one or two buckets wrong and the rest was at least partially contained. The only bad results came from the Apache Ant project. Though, that the incremental approach is not applicable in that case was already shown in RQ2. There is a restriction for the incremental approach resulting from different dependency graph structures. The connectivity of the graph plays an important role in the PageRank calculation: The higher the connectivity, the better the results of the incremental approaches.

6.2 Future Work

In this bachelor's thesis, we relied for the evaluation on 6 software projects. To guarantee the coverage of all possible software project types and languages, one has to make a much bigger evaluation with a higher amount of projects. Such an enlargement would enhance the reliability of the results.

The first improvement, which is the approximation of the purely incremental PageRank approach, was already presented in this thesis. Another interesting idea that might

improve the approach once more, is to choose a threshold for the approximation depending on the size of the project. As the evaluation has shown: The precision of the results decrease with an increasing project size. However, the runtime improvement gets bigger for bigger projects. To get the best compromise between runtime improvement and accuracy, the threshold has to get smaller the bigger the project gets.

Bibliography

[1] D. Steidl, B. Hummel, and E. Juergens, "Incremental origin analysis of source code files," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*, 2014.

[2] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: Incremental, distributed, scalable," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, 2010.

[3] D. Steidl, "Using network analysis for recommendation of central software classes," Technische Universität München, Tech. Rep., 2012.

[4] R. Haas, "Identification of unnecessary source code," Master's thesis, Technical University of Munich, 2017.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Stanford InfoLab, Tech. Rep., 1999.

[6] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar, "Incremental page rank computation on evolving graphs," in *Special interest tracks and posters of the 14th international conference on World Wide Web*, ACM, 2005.

[7] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D Sivakumar, "Link evolution: Analysis and algorithms," *Internet mathematics*, vol. 1, no. 3, 2004.

[8] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, no. 1, 1904.

[9] Y. Dodge, "Spearman rank correlation coefficient," in *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 502–505.

[10] I. Şora, "Finding the right needles in hay helping program comprehension of large software systems," in *Evaluation of Novel Approaches to Software Engineering (ENASE), 2015 International Conference on*, IEEE, 2015.

[11] I. Şora and D. Todinca, "Using fuzzy rules for identifying key classes in software systems," in *Applied Computational Intelligence and Informatics (SACI), 2016 IEEE 11th International Symposium on*, IEEE, 2016.

[12] M. Kamran, M. Ali, and A. Ahmed, "Generating suggestions for initial program investigation using dynamic analysis," in *Communication, Computing and Digital Systems (C-CODE), International Conference on*, IEEE, 2017.

[13] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015.