FINDING EXPERT DEVELOPERS USING ARTIFACT TRACEABILITY GRAPHS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
İdil Hanhan
September 2024

Finding Expert Developers Using Artifact Traceability Graphs
By İdil Hanhan
September 2024

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Eray Tüzün(Advisor)

Uğur Doğrusöz

İsmail Sengör Altıngövde

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

FINDING EXPERT DEVELOPERS USING ARTIFACT TRACEABILITY GRAPHS

İdil Hanhan
M.S. in Computer Engineering
Advisor: Eray Tüzün
September 2024

Mentoring is a commonly used practice in the software industry where mentors and mentees are matched to ease the onboarding process of the mentee, who is a newcomer. Also, during a project's life cycle, developers work on sections of the codebase that are unfamiliar to them. Both cases raise the task of finding an expert developer to contact for possible questions. With this study, we aim to construct an algorithm that recommends expert developers for a specific part of the codebase, namely folders, files, and methods, based on previous developer activities such as commits and code reviews. We construct an artifact traceability graph using commit history, method change history, code review history, and issue history. The relationships in the graph are weighted according to recency and a weight coefficient we determine intuitively. Utilizing this graph, we calculate a score representing the developer's expertise level on a folder, file, or method, and recommend developers with the highest expertise. To evaluate the success of our algorithm, Expert Developer Finder, we compare its recommendation with the developers who commented on related issues. We run our algorithm on three open-source projects - Nutch, OpenNLP, and Curator. On average, for weighted recommendations, we reached up to 84% accuracy for folders, 82% accuracy for files, and 88% accuracy for methods. On average, for unweighted recommendations, we reached up to 84% accuracy for folders, 84% accuracy for files, and 93% accuracy for methods. We believe that our results show that the Expert Developer Finder algorithm is able to recommend experts by utilizing the historical data of projects. However, further work is required to fine-tune the weights set in the artifact traceability graph.

Keywords: key developer, artifact traceability graph, expert developer, mentor recommendation, expert recommendation, developer recommendation.

ÖZET

YAPI İZLENEBİLİRLİK ÇİZGELERİNİ KULLANARAK UZMAN GELİŞTİRİCİLERİ BULMA

İdil Hanhan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray Tüzün
Eylül 2024

Mentörluk yazılım sektöründe yaygın olan bir uygulamadır. Bu uygulama ile mentör işe yeni başlayan bir danışan ile eşleşir ve danışanın işe başlama süreci kolaylaşır. Daha sonra, yazılım geliştirme sürecinde, geliştiriciler kod temelinin aşina olmadıkları bölümlerinde çalışabilirler. İki durumda da olası sorular için doğru geliştiriciyi bulmak gerekmektedir. Bu çalışma ile kod temelinin bir bölümü, örneğin klasör, dosya ve method, için katkı (commit) ve kod inceleme geçmişini kullanarak uzman geliştirici öneren bir algoritma oluşturmayı amaçlıyoruz. Yapı izlenebilirlik çizgesini katkı (commit) geçmişi, method değişiklik geçmişi, kod inceleme geçmişi ve sorun raporları geçmişini kullanarak inşa ediyoruz. Çizgede bulunan ilişkileri, güncellik ve sezgisel olarak belirlediğimiz bir ağırlık katsayısı ile ağırlıklandırıyoruz. Çizgeyi kullanarak geliştiricilerin bir kod bölümundeki (klasör, dosya ve method) uzmanlığını simgeleyen bir skor hesaplıyoruz ve uzmanlığı en yüksek olan geliştiricileri öneriyoruz. Uzman Geliştirici Bulucu algoritmamızın başarısını ölçmek için algoritmanın önerilerini ilgili sorun raporlarına yorum bırakan geliştiriciler ile karşılaştırıyoruz. Algoritmamızı üç açık kaynaklı projede (Apache Nutch, OpenNLP ve Curator) çalıştırdık. Ağırlıklı önerilerde, ortalama olarak, klasörler için %84, dosyalar için %82, methodlar için %88 doğruluğa ulaştık. Ağırlıklı olmayan önerilerde, ortalama olarak, klasörler için %84, dosyalar için %84, methodlar için %93 doğruluğa ulaştık. Sonuçlarımızın Uzman Geliştirici Bulucu algoritmasının proje geçmişini kullanarak uzman geliştirici önerebildiğini gösterdiğine inanıyoruz. Ancak, yapı izlenebilirlik çizgesinindeki ağırlıkların ince ayarını yapmak için daha fazla çalışma gerekmektedir.

Anahtar sözcükler: anahtar geliştirici, yapı izlenebilirlik çizgesi, uzman geliştirici, mentör önerme, uzman önerme, geliştirici önerme.

Acknowledgement

I would like to express my deepest gratitude to my advisor, Asst. Prof. Eray Tüzün, for his unwavering support, guidance, and encouragement throughout this research. His insights and expertise were invaluable in shaping this work and my academic and professional growth.

I also extend my sincere thanks to my esteemed jury members, Prof. Uğur Doğrusöz and Prof. İsmail Sengör Altıngövde, for their time and consideration in reviewing this work.

I am grateful to Selin Bahar Gündoğar and other BILSEN members for their valuable suggestions during my research.

A heartfelt thanks to my parents, Deniz and Bilgi, sister İrem and friends for their unwavering love and patience. Their support was vital during the course of this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Developers are the critical resource for the success and continuity of a software project. During a project's life cycle, developers often work on a section of the codebase that is unfamiliar to them. This occurs more frequently for newcomers - who may or may not be experienced developers. Since the codebase is almost completely unfamiliar to newcomers, they are more likely to have questions. So developers, newcomer or not, find themselves looking for an expert developer to contact for possible questions.

A report published by the Chartered Institute of Personnel and Development (CIPD) in 2020 recommends the mentorship system for newcomers and states that it allows a learning ground for them [1]. Mentorship is a guidance commonly used in the software industry to help with adapting to a new workplace. There are a number of studies about the advantages of mentorship in this industry. A study by Fagerholm et al. [2] shows that mentorship positively affects the efficiency of developers via a study conducted on 120 students who worked on Meta's, formerly known as Facebook, open-source projects as a part of Meta's Education Modernization Program. Another study by Britto et al. [3], conducted in Ericsson, shows that the mentorship system is a key aspect. In 2018, Britto et al. [4] showed that mentorship is not only beneficial for new developers but is also recommended to be utilized for a longer period. However, studies on

recommending the ideal mentor for newcomers are limited.

Rodeghero et al. [5] show that new developers who work remotely, due to the COVID-19 pandemic, may have trouble finding experienced developers to ask questions about the code base. They estimated that such problems that arose during the pandemic will persist as remote working policies are implemented. X, formerly known as Twitter, has remote working as a permanent option -first announced in 2020 [6] and is still available [7] (even though it was temporarily removed after the ownership change of the company). Also, in 2020, Meta stated that half of their workers will be able to work remotely in the upcoming years [8], and in 2023 they adopted a hybrid working model with three days of working in the office - though it did not affect the existing remote workers [9]. As the mega technology firms continue to implement such measures, it is foreseeable that remote working opportunities will persist. Therefore, we believe there is no better time to investigate how to aid developers in finding an expert developer.

The aim of this work is to develop an automated recommendation approach for a code segment (folder, file, or method) to solve such problems. We believe this tool will enable newcomers to quickly contact the most knowledgeable developers about specific code segments. In addition, it will help developers, regardless of their experience level, become more familiar with unfamiliar code segments with the help of experts.

One potential existing solution to find the expert developer is using the *git blame* functionality of Git, which finds the latest author of the code segment in question [10]. However, results from *git blame* may not always be useful. The developer, who is pointed to by *git blame*, is the one who worked on the code segment most recently but that does not necessarily mean that they are the one who worked on it most frequently or the one who has the most expertise - since the purpose of *git blame* is not to find the expert. German et al. [11] also argue that *git blame* might leave the wrong impression when the purpose is to find an expert. Additionally, only one author's name is outputted by it even if multiple developers have worked on the code, which limits the number of developers that can be contacted. Our algorithm differs by listing at least three most qualified

developers by default for the chosen code segment. When *git blame* is not useful, the developer might go on to review the history of the file or get in contact with multiple developers to find the expert developer. Reviewing the file history and comparing different versions might become tedious as the scope of the search increases. Contacting multiple developers is, oftentimes, inefficient as developers might respond late, and the number of developers to contact might be unknown or open-ended. Even though the developer might eventually find the expert developer, this process introduces an overhead to the developer's work.

There have been several studies that introduce an algorithm that recommends a developer. In some studies, the aim is to find a developer that will be assigned to a task [12][13][14][15] or to a project [16] or to a PR review [17][18][19][20]. In others, the purpose is to find a developer who can be a mentor [21][22][23]. There have also been studies that aim to define and/or quantify the expertise of developers in a project [24][25][26][27][28][29][30][31][32][33][34]. Many, such as [22][23][31] and more, utilize the historical data of software projects, from version control systems such as GitHub, to develop their algorithm. Some have also considered data from email history [35] or online platforms for developers, such as Stack Overflow [16][28].

Considering the studies found that recommend mentors, the majority do not provide a tool that can be used during the project life cycle. The only study found that finds mentors and includes a tool is by Canfora et al. [21]. Their tool utilizes email and version history to recommend mentors for a file or any component/topic mentioned in an open-ended question. Kintab et al. [22] state their intention to provide a plug-in for their Expert Recommender System Framework (ERSF). Another important aspect to consider is the granularity of the component for which experts are found and recommended. Kintab et al. [22] recommends experts for code fragments, Akter [23] recommends experts for methods and Canfora et al. [21] recommends experts for files. However, it should be noted that Canfora et al. [21] also account for textual help requests which might provide more granularity. Therefore, to the best of my knowledge, there has been little prior work that recommends experts for a range of components with varying granularity.

In this study, the objective is to develop an algorithm that suggests expert developers for a specific component, namely folder, file, or method, of a software project. The research objective is as follows:

Research Objective: To what extent can the history of a software project be utilized to accurately find expert developers for a software project component?

Using the historical data of the software project, an artifact traceability graph is constructed and used to find the experts for a specific component. The algorithm is accessible via a tool, Software Artifact Analyser (SAA) [36] initially developed by Merdol et al. [37], that also offers visualization and other algorithms for the project, such as finding anomalies. We believe that with our algorithm, developers can escape the hurdle of finding an expert with the existing solutions, have a tool that they can use during the project life cycle, and have the option to find experts for a range of components. We construct and evaluate this algorithm in light of the following research questions:

RQ1: To what extent can the history of a software project be utilized to accurately find expert developers for a software folder?

RQ2: To what extent can the history of a software project be utilized to accurately find expert developers for a software file?

RQ3: To what extent can the history of a software project be utilized to accurately find expert developers for a software method?

The first step of developing our algorithm is constructing an artifact traceability graph that represents the relationships between different artifacts in the project, such as developers, files, methods, etc. The artifact traceability graph

is constructed from data such as commits, code reviews, and issues. The second step of our development is utilizing this graph to calculate the expertise of a developer for a folder, file, or method. We represent the expertise of a developer with a know-about score, which is a modified version of the score introduced in [20].

We run our algorithm on three open-source projects, Nutch, OpenNLP, and Curator - all from Apache. To evaluate the success of our algorithm, we retrieve the developers who authored commits and commented on the issues related to a component and consider them as the ground truth for experts. We believe commenting on issues signals knowledge of the artifacts related to the issues since they contain information about the project's requirements. Then, we compare the expert developers recommended by our algorithm to the top issue commenters of issues related to the artifact in question. We collected the data from these open-source projects in July and August 2024 from both GitHub and Jira. We utilize the expanded version of the SAA tool for data retrieval. Then, we developed queries to run the algorithm to find the experts independently from the tool and extract related issues and retrieve top commenters.

1.1 Thesis Contribution

The main contributions of this thesis are the following:

1. We develop an algorithm that recommends expert developers for components with varying granularity using artifact traceability graphs constructed from the version and issue history of a project.
2. The algorithm is integrated into a tool that can be utilized during the project life-cycle.
3. Provides an opportunity to extend the presented artifact traceability graph structure.

1.2 Organization

In the following section, Section 2, we share background information on the usage of graph analysis to gain insight into developers in software projects, on the works that develop algorithms to quantify expertise and recommend developers for issue handling, PR reviews, project assignment, and mentorship. Section 3 describes the underlying tools we utilize in our work. In Section 4, we present our methodology to retrieve software project data, construct an artifact traceability graph, and develop an algorithm to recommend experts. Section 5 details how we chose the open-source projects to test our algorithm with and describes our validation strategy. Section 6 shares our results, and Section 7 shares statistics about the previous work we believe to be most similar to ours and discusses our work and results. In Section 8, we share the threats to the validity of our work and, finally, in Section 9, we conclude the paper and share our future work.

Chapter 2

Background Information

In this section, we share background information on works utilizing graph analysis, quantifying expertise, and developing algorithms to recommend developers. Works utilizing graph analysis are presented in Section 2.1. Then in Section 2.2, we present works that define and/or quantify expertise. Works recommending developers for issue handling are presented in Section 2.3, and works that recommend developers for reviewers are detailed in Section 2.4. Then works that recommend developers for projects and mentors are detailed in Sections 2.5 and 2.6 respectively.

2.1 Graph Analysis

Graph analysis is one of the methods that have been used to represent and/or analyze software projects. For example, Bhattacharya et al. [38] use a graph-based approach in the context of software evolution. Graph analysis is used to detect plagiarism in both [39] and [40] and Meneely et al. [41] use network analysis to detect failures. For this study, we focus on the usage of graph structures and analysis to gain insight into developers, whether that is related to their expertise, community, evolution, etc.

Hu et al. [42] utilize network analysis to investigate the reasons behind a positive evaluation a developer gives to another. They focus on the effect of the following metrics on this evaluation: past reputation, shared affiliation, and homophily in location, nationality, programming language, and community status. They source their data from an online platform called *Ohloh*, currently known as *Black Duck Open Hub*. In this platform, developers can send each other a positive evaluation, by sending a *Kudos*, and this is the information that is used to construct a network of developers. As a result of their analysis, they found that all three metrics mentioned above are effective.

Aljembi et al. [43] investigate the evolution of a developer social network (DSN), specifically a bug-tracking system based DSN. They focus on the following perspectives: social network analysis, developer social network as an ecosystem, community evaluation patterns, and the core-periphery structures. From the social network analysis perspective, they found that DSN follows a law degree distribution and is a "small world" in each period of time. From the ecosystem perspective, they used biodiversity measurements and found 55% diversity with 75% of evenness between the developers to contribute to different OSS projects in the same environment. From the community evaluation patterns perspective, they found that DSN manifests all evolution patterns. Lastly, for the core-periphery structure perspective, they found 20% of the developers were core members while about 80% were peripheral members and only 10% of the developers changed their position over time.

Hou et al. [44] develop an algorithm for community detection in software ecosystems, such as GitHub, according to developer cooperation intensity. For the calculation of cooperation intensity, the authors rely on both topology information, which is the information that is gathered from the network itself such as node degree, and developer interaction information such as the repositories that developers share and the frequency of this action. Their algorithm, ABDCI, draws from the hierarchical clustering idea of the Louvain algorithm. They evaluate ABDCI on five network structures constructed from GitHub repositories and then compare their algorithm to existing community detection algorithms. They found that their algorithm produces more clear community structures.

The graph structure we utilize in this study is referred to as an *artifact traceability graph* which has been used in [45], [20] and [32]. An artifact traceability graph is a representation of the software project where the graph nodes correspond to various artifacts, such as developers, issues, files, etc. and the links represent a type of relationship between the artifacts, such as development, inclusion, relation, etc.

Sülün et al. [20] utilize artifact traceability graphs to develop an algorithm that recommends reviewers. Based on this graph, a metric referred to as the *know-about score* is calculated - first introduced in their previous work [45]. This metric considers the distinct paths between a developer and a software file. They also expand their previous work by taking into account the recency aspect of artifacts. They compare the recommendations given by their algorithm to recommendations given by other approaches, such as Naive-Bayes [46], RevFinder [18], and Profile-based approach [19]. Top-k accuracy and Mean Reciprocal Rank (MRR) are used to validate the results. On average, they observed higher accuracy with the recommendations given by their algorithm. Additionally, they introduce a GitHub bot tool.

Çetin and Tüzün [32] utilize artifact traceability graphs to categorize developers into *Jacks*, *Mavens* and *Connectors* and introduce algorithms that evaluate knowledge distribution, identify key developers and suggest a replacement for leaving developers. They also introduce a proof of concept tool that displays their evaluation results. This tool does not give recommendations but displays statistics about developer scores and categories. To evaluate the algorithms proposed, they conducted case studies on six open-source projects. They found that their algorithm is compatible with the top commenters, who left comments in the issue tracking system, by up to 98%. They achieved up to 91% accuracy with their replacement recommendation algorithm and their knowledge distribution algorithm is compatible with 94% of the baseline approaches.

2.2 Quantify Expertise

There have been several studies that aim to quantify the expertise of developers in a software project. This can be accomplished by investigating the factors that cause one to be considered an expert by offering a way to rank experts in a software project, etc.

Schuler and Zimmerman [24] are the first to propose utilizing usage expertise, gained when functionality is used, to define expertise. This is different from implementation expertise, gained by changing an artifact, which is a common measure of expertise.

Ma et al. [25] investigate the utilization of usage expertise for finding experts. They find experts for a method at a point before a commit and check if the author of the commit, who they consider as the expert, is in the Top-k results. They conclude that usage expertise ranks the best for Top-1 compared to implementation expertise. In addition, they investigate the feasibility of recommending developers across projects and found it to be possible - however, they do not check the accuracy of this kind of recommendation.

Constantinou and Kapitsaki [26] develop a system that defines expertise by considering the quantity and continuity of a developer's contribution to software projects -specifically their commit activity. The result is a ranking of programming languages according to the developer's expertise. They use the GHTorrent dataset [47] to gather information about developer activity and also extend it by getting more information from GitHub. They rely on Stack Overflow to evaluate their approach, where they compare their ranking to the programming languages for which the developer has accepted answers. They found that their approach, on average, outlines the developer's main expertise in the top 20%.

Baltes et al. [27] describe expertise as a combination of general and task-specific knowledge and expertise. They note that factors such as personality, skill, and mentoring affect the formation of expertise and the code quality and productivity of the developer.

Huang et al. [28], develop an algorithm called CDPScorer to evaluate the developer's ability using their Community Question Answering (CQA) activity. The algorithm first links users of the CQA to the users in the Open Source Software (OSS) community. Then, they assign programming ability terms to each question and project by using the tags assigned to them. If the CQA does not have tags, the tags in Stack Overflow are extracted. The answers in CQA are scored using features such as upvotes, downvotes, and answer length. The TextRank algorithm is used to mine the keywords from answers and the extracted words are matched with tags. Then, a static code analysis tool is used to assign code-based scores to each project. The ability score is then calculated as the weighted average of question and code-base scores. For code evaluation and answer quality calculations, Linear Regression and Regression Tree M5P and the two combined are applied. They found that applying M5P for answer features and Linear Regression on code metrics has the best result.

Yan et al. [29] generate a Heterogeneous Information Network (HIN), which displays the relationships between objects to profile the expertise of developers. The data is extracted from GitHub and Stack Overflow. Random Walk with Restart (RWR) is applied by finding the proximity of two nodes to eliminate the cold-start problem. The algorithm's results are compared with that of CPDScorer [28] and they found that HIN's performance success rate is 67.88%, whereas CPDScorer's success rate was 55.57%.

Vadlamani and Baysa [30] investigate how an expert is defined and the purpose behind developers contributing to GitHub and Stack Overflow. They conduct an online survey with developers who are active on both platforms mentioned and perform a quantitative and open coding analysis. They found that knowledge and expertise, effective and clear communication, and analytical thinking are the key factors that define expertise. Furthermore, they found that contributions to GitHub are motivated by personal factors, while contributions to Stack Overflow are motivated by professional ones.

Javeed et al. [31] use deep learning to develop an algorithm that identifies the level of coding expertise. They gather project code from GitHub and they use

SonarQube to measure the quality of the code. The following metrics are used to categorize the projects into novice and expert: complexity, reliability, security, maintainability, duplication, and complexity. They test their approach with three deep learning models, Long Short-Term Memory (LSTM), convolution1D, and a hybrid of the two. They found LSTM to be the better predictor with 98.27% training accuracy and 96.25% test accuracy along with 0.4 loss reduction.

As mentioned in the subsection 2.1, Çetin and Tüzün [32] utilize artifact traceability graphs to categorize developers into *Jacks*, *Mavens* and *Connectors*. Since their work includes the categorization of developers, we believe their work can be categorized under the purpose of quantifying expertise.

Sun et al. [33] propose with their CValue algorithm that multidimensional information can be used to calculate developer contribution for every commit. The algorithm works by calculating four different aspects: AST(Abstract Syntax Tree) difference, complexity metrics, call graph, and program dependence graph. AST difference is calculated to see how the quantity of the code changes and what kind of changes are made to the code. Complexity is calculated by assigning weights to lines of codes that are more central, and therefore more complex. Complexity is composed of four parts: lines of codes of a function, Halstead Volume, the percentage of comments and Cyclomatic complexity. Call Graph assigns a higher weight to the modifications to the core logic of the code. The program dependence graph assigns a higher weight to the modifications to the core statements or variables of a method. These four parts are normalized and added to create a score. The results are compared with the baseline values, which are two other metrics: ELOC and COCOMO [48]. CValue performed better than ELOC by 12.16% and COCOMO by 19.59%.

Mockus et al. [34] model developer productivity at Meta by utilizing Software Supply Chain (SSC) networks and using Kars centrality and PageRank on the networks. The graph weights are timestamps that indicate when the node relationships were created. This paper takes into account the aspect of a developer who does management work in addition to a regular developer and introduces a new node called community to indicate when existing code is reused somewhere

else. Furthermore, it has unique relationship types such as part-of-team and part-of-commit.

2.3 Issue Handling Recommendation

In some studies, the authors aim to quantify expertise and find experts for the purpose of assigning them to an issue or bug.

Wu et al. [12] propose recommending developers for bugs by using K-Nearest-Neighbour search and utilizing frequency and social network metrics such as In-degree, Outdegree, Degree, PageRank, Betweenness, and Closeness. When given a new bug report, the algorithm first uses the K-Nearest-Neighbor search to find the past bug reports related to the new one and the developers who worked on them. Then frequency and social network metrics are used to rank the developers. In conclusion, they found that their algorithm, DREX, outperforms the traditional bug-assigning method, which is based on text classification, and that frequency and Outdegree metrics outperformed the other metrics.

Zhang et al. [13] propose three steps to recommend developers when a new bug report is made. The first step is to build a concept profile that contains topic terms and documents created from past bug reports. The bugs are categorized with a K-Means Clustering algorithm and the cosine measure is used to measure textual similarity. Then, topic terms are extracted from the created clusters and are used as bug concepts. To find the closest match, the frequency distribution of the topic terms in a bug report is calculated and compared to that of previous bug reports. The next step is creating a social network where each node represents a developer and the links represent their cooperative relationship. The numeral in the nodes represents the number of bug reports that were fixed by a developer. Then, the developers were ranked and they found that their algorithm outperformed all other similar algorithms, including the DREX algorithm from [12].

Li et al. [14] explore matching developers with tasks posted in crowdsourcing mediums. They first construct a developer social network so that the relationship between active and inactive developers can be monitored. Secondly, social influence is calculated between developers by browsing similarity and task bidding. Then, their social similarity degree is calculated, and the algorithm recommends active developers using developer-task ratings and implicit feedback from friends on the platform. The algorithm recommends inactive developers by combining the recommended tasks of their active developer friends on the platform.

Yadav et al. [15] compute a developer expertise score by using priority, versatility, and average fix-time. Three similarity measures are used to rank the developers: feature-based, cosine-based, and Jaccard. Hit ratio and reassignment rates are used for the post-assessment of the success of developer ratings. The results show a 20% improvement compared to state-of-the-art studies by achieving a 90% accuracy rate.

2.4 Reviewer Recommendation

Another aim for quantifying expertise and finding developers is to recommend them as reviewers.

Çetin et al. [17] conduct a literary review of code reviewer recommendation studies. They look into 29 exemplary studies conducted between 2009 and 2020. They found that the majority of studies use machine learning approaches, which are then closely followed by heuristic approaches. They stated that most studies evaluate their models using open-source projects and they preferred incremental training set validation techniques. Among the problems that arise in these papers, they found that most studies have reproducibility issues. Furthermore, the most common validity threat was related to generalizability and dataset integrity. For future work, they found that studies mainly discuss refining models and conducting additional experiments.

Thongtanunam et al. [18] investigate using a file location-based code-reviewer recommendation tool, RevFinder, to enhance distributed software development's code review procedure. They believe that code reviewers with comparable levels of experience are likely to evaluate files with comparable directory structures. They use the Longest Common Prefix (LCP), Longest Common Suffix (LCS), Longest Common Substring (LCSubstr), and Longest Common Subsequence (LCSubseq) string comparison techniques to measure the similarity between file paths. Then, a similarity score is calculated by comparing a new review with a closed review and then distributing these scores to the code reviewers who participated in the earlier reviews using a Code-Reviewers Ranking Algorithm. The paper also compares the results of another tool called REVIEWBOT [49]. The results show that RevFinder has an accuracy range from 20% to 86% over four open-source software projects.

Fejzer et al. [19] presents a method for recommending code reviewers based on profiles derived from the history of their code reviews. To facilitate quick similarity calculations, these profiles are stored as multisets of file path segments in hash tables. The technique compares new commits with reviewer profiles using the Tversky index, taking into account the decay factor to account for reviews' recentness. Their results showed that the new approach improved the accuracy of reviewer recommendations compared to existing methods. For instance, the method achieved a Top-1 recall of 0.5492 while the Top-10 recall was 0.9066 in one of the software projects.

As explained in Section 2.1, Sülün et al. [20] utilize artifact traceability graphs to develop an algorithm that recommends reviewers.

2.5 Project Recommendation

In some studies, the aim is to quantify expertise and find developers to recommend them as possible developers for projects.

Zhang et al. [16] recommend expert developers for both small and large open source projects on GitHub by utilizing DA (development activity) based and KS (knowledge sharing) based recommendations from Stack Overflow and their hybrid model with different coefficients. The results show that the hybrid model performs the best for unpopular GitHub projects and the DA-based approach works better for popular projects. Furthermore, different coefficients result in different accuracy. A decrease in the coefficient results in a more accurate hybrid result for unpopular projects with 5-10 developers compared to the DA-based approach. However, the DA-based approach is better for popular projects.

2.6 Mentor Recommendation

Similar to our work, there have been past studies that aim to quantify expertise and find developers for mentorship.

Canfora et al. [21] investigate finding mentors for a new developer in OSS projects by using both developer mailing patterns and project version history. They also integrate their algorithm, YODO, as an Eclipse plugin. The authors investigate the emailing patterns between users and code contributions. Then, among the selected mentors, an IR (information retrieval)-based approach is used to find mentors who have helped other developers with similar problems. The newcomer's textual request for help and potential list of mentors are used as inputs. The algorithm supports queries based on the source code files a developer is working on and the textual help request entered by the developer. In addition, they visualize mentor and mentee relationships. The algorithm is tested on five open source projects and the results conclude that it can identify mentors with an 80% accuracy and recommend mentors with a 77% accuracy.

Kintab et al. [22] develop an expert recommender framework that suggests the correct developer to assist another developer - referred to as the *active developer*. They take into account both technical and social heuristics. Technical heuristics include degree of code similarity, most recent modification etc. Social heuristics

include the number of shared files, the previous responses the *active developer* gave to the framework's suggestions etc. Each heuristic is weighted differently according to results from an experiment conducted with human judges. This framework uses data from the SimCad clone detection tool, which finds similar code fragments to the original one that is provided, GitHub, and the information they keep in the framework. For evaluation, they compare their recommendations to the recommendations given by human judges and machine learning algorithms such as NaiveBayes, J48, and NaiveNet. Their accuracy results for testing the heuristics above ranged from 23% to 100%.

Sharmin Akter [23] proposes a new algorithm that utilizes usage expertise, implementation expertise, and the two combined to recommend expert developers. The usage expertise is computed by detecting every instance that a developer made a call to a method and the implementation expertise is calculated by detecting new code additions to the method or the method declaration. Different weights are assigned (between 0 and 1) to both the usage and implementation expertise's raw data to calculate the optimal contribution data. The algorithm allows users to find the expert developers of a method when the user inputs the method name, its class name, and its package name. The algorithm then outputs a list of the expert developers for the method. The accuracy of the recommendations is checked by computing the average accuracy and Mean Reciprocal Rank (MRR). Both individual and combined expertise models are evaluated with Top-1, Top-5, and Top-10 accuracy. The results of the study show that the combined version does not outperform the individual expertise model, the largest group with the smallest number of developers has the best results, prioritizing usage expertise over implementation expertise in weights (0.75 for usage and 0.25 for implementation) produce the best results, and that their individual and combined expertise models outperform the recommendations given by [25]. The accuracy was tested for three software projects: the method accuracy ranged from 65.93% to 97% and the commit accuracy ranged from 74.70% to 92.75%. In conclusion, among the three different types of models, the usage expertise model was found to be the most successful and the implementation expertise model performed the worst.

Chapter 3

Underlying Tools

In this section, we explain the two underlying tools we use in our work, Software Artifact Analyser (SAA) [36] and CodeShovel [50]. SAA is the tool we built upon to include our algorithm. While changes are made to the tool to fulfilll our purpose, we believe it is beneficial to explain the current state and purpose of the tool before going into our methodology. CodeShovel is a tool we utilize to retrieve method histories for a software project. We believe it is important to explain this tool independently of our work so that the benefits and limitations that come with using it are better understood.

3.1 Software Artifact Analyzer

Merdol et al. [37] propose a framework for an analysis tool that uses artifact traceability graphs and implement a tool, SAA, in light of this framework. This section explains the initial state of SAA while our expansion of the tool is explained in later sections.

SAA is a tool that researchers and practitioners can utilize to visualize and gain insight into their software projects. The tool has two major parts. In the first part, the data is retrieved and cleaned, and an artifact traceability graph

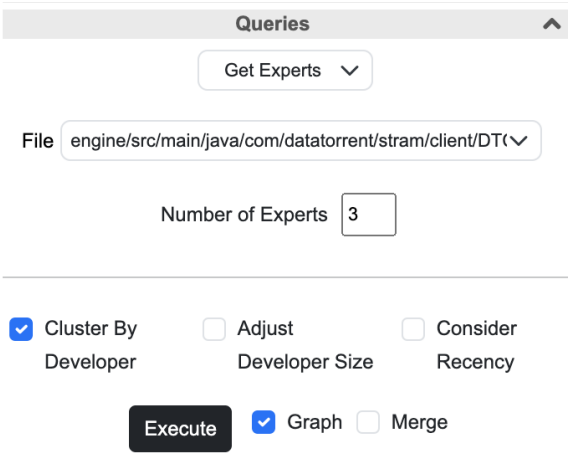

Figure 3.1: The menu for running the *Get Experts* algorithm

is constructed. This part is written in Python and utilizes Perceval [51]. Data retrieval is extended in our research to gather method history information via CodeShovel [50]. The second part of the SAA tool handles the visualization of the graph and the algorithms that can be run on it. This part is written in Java, and Neo4j graph database [52] is used to store the graph. Cypher queries are used to construct the graph and retrieve any information that is needed from the graph. Some of the algorithms offered in SAA are:

- Get Commits of a Developer
- Get Anomalies
- Get Anomaly Statistics
- Get Recommended Reviewers
- Get Experts

The algorithm that is related to our work is *Get Experts*. In its initial version, this algorithm is triggered by clicking on a file node in the graph and selecting the *Get Experts* algorithm from the offered queries. The related menu can be seen in Figure 3.1. The algorithm results are then displayed - as seen in Figure 3.2.

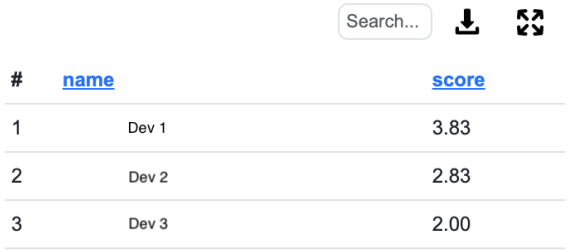

Figure 3.2: The results of the *Get Experts* algorithm (Names removed for privacy)

The initial version of SAA's *Get Experts* algorithm can only be used for software files. Our work extends the algorithm to get experts for methods and folders. The *Get Experts* algorithm has the following steps:

1. Developers who are candidates for being experts in the selected artifact are gathered using ***subgraphAll***.
2. A custom procedure, ***findNodesWithMostPathBetween***, is called to get expert developers and their know-about score.
3. The results are gathered and displayed.

In the first step, the ***subgraphAll*** procedure from Neo4j Awesome Procedures (APOC) plugin [53] is used to gather the developers who are a specific length away from the artifact in question. This step uses a breadth-first approach to find the possible developers. With this procedure, we are able to focus on a specific portion of the traceability graph and not the entire graph.

In the second step, a procedure created in [37], is utilized to get the experts. The signature for ***findNodesWithMostPathBetween*** can be seen below.

findNodesWithMostPathBetween Signature

$findNodesWithMostPathBetween(elementIds, ignoredTypes, terminalType, weightProperty, lengthLimit, nodeLimit, isDirected, pageSize, currPage, filterTxt, isIgnoreCase, orderBy, orderDir, timeMapping, startTime endTime, inclusionType, timeout, idFilter)$

The most important parameters of this procedure are the following:

- *elementIds*: The source node in a path.
- *ignoredTypes*: Any edge that can not be in a path.
- *terminalType*: The target nodes in a path.
- *weightProperty*: Any weight attached to the edges that should be taken into account.
- *lengthLimit*: The length limit of a path between source and target.
- *nodeLimit*: The limit for the number of results returned.

The values we use for these parameters in our version of the algorithm will be explained in Section 4.

Inside the ***findNodesWithMostPathBetween*** procedure, the ***Common Target*** algorithm is utilized. This algorithm was implemented in [54] and is also utilized in [37]. The ***Common Target*** algorithm uses a depth-first approach to find paths, at most of a specific length, between a set of source and target nodes in a graph. Each target node has a score which is calculated by the multiplication of weight values divided by the length of the path score. This is equivalent to our know-about score, which will be explained in Section 4. Then the algorithm returns the top target nodes, together with their score.

In the final step, the results returned by the procedure are reformatted to be displayed. The results also trigger some visual changes, but we will not go into detail about these as they were not introduced in this work.

3.2 CodeShovel

We use CodeShovel [50] to gather information about the change history of methods. CodeShovel traces method-based history and can be used for both open

Table 3.1: CodeShovel and Expert Developer Finder Algorithm Category Mapping

Change Case	**Explanation**	**CodeShovel Category**	**Expert Developer Finder Category**
I-A	New line to an existing method	bodychange	Update
I-B	New method to existing file	introduced	Initialisation
I-C	New file	N/A	N/A
I-D	New folder	N/A	N/A
I-E	Blank line added	formatchange	Ignore
I-F	Comment added	bodychange, docchange	Update
II-A	Line deleted from a method	bodychange	Update
II-B	Method deleted	N/A	N/A
II-C	File deleted	N/A	N/A
II-D	Folder deleted	N/A	N/A
II-E	Blank line deleted	formatchange	Ignore
II-F	Comment deleted	bodychange, docchange	Update
III-A	Method updated	bodychange	Update
III-B	Method renamed	rename	Update
III-C	Method exception/ parameter/ modifier/ return type changed	returnchange, paramchange, modifierchange, exceptionchange	Update
III-D	Method carried over inside the same file	N/A	N/A
III-E	Method moved to a different file	movefromfile	Move
III-F	File renamed	filerename	FileRename
III-G	File moved to a different folder	filerename	FileRename
III-H	Folder renamed	filerename	FileRename
III-I	Comment updated	bodychange, docchange	Update
Combo	Any combination of the above cases is possible	multichange	Combination of the corresponding individual categories

and closed-source projects. We chose CodeShovel as it outperforms Git log and FinerGit [55], which are its most widely used competitors. The tool supports API calls that list all of the files in a software project, list all the methods in a file, and retrieve the history of a method. To get the method history, file path, method name and method start line is needed. The output is a JSON file containing the history of the method.

For method history retrieval, CodeShovel works by searching backwards, starting from the commit pointed to by HEAD until it finds the first commit where the method was created. It is important to note that a commit hash can be specified as the starting point of the backwards traversal. CodeShovel measures the body and signature similarity between methods using The Jaro-Winkler distance algorithm to detect the method over the life-cycle of the project.

Each change in the method history is categorized into the following by CodeShovel:

- introduced
- bodychange
- filerename
- formatchange
- movefromfile
- returnchange
- paramchange
- modifierchange
- exceptionchange
- docchange
- multichange

We rely on this categorization when deciding on a course of action when processing a method change. The change cases these categories correspond to and the actions we take can be seen in Table 3.1. The change categories in our work and their effect on the graph will be detailed in the *Construction* subsection of Section 4. Since we rely on this categorization, we are only able to include the method changes detected by CodeShovel. For example, a deleted method is not detected by CodeShovel and therefore the deletion action on a method is not tracked on the method level by our algorithm.

Chapter 4

Methodology

We divide the methodology into three sections. The first section details the data retrieval, then we explain how an artifact traceability graph is constructed, and finally, we share our Expert Developer Finder algorithm.

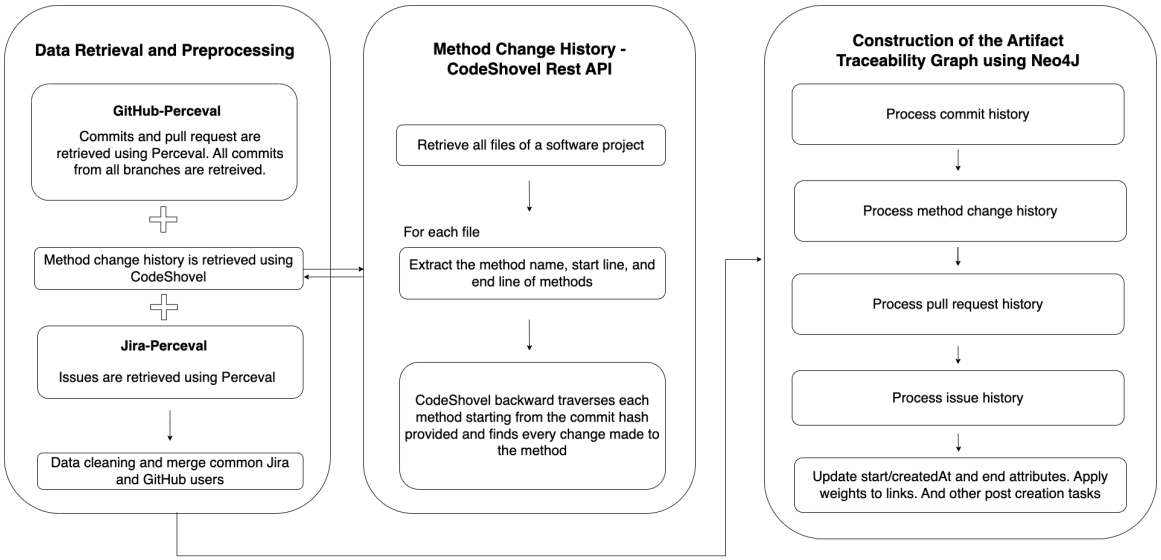

Figure 4.1: Flowchart of the creation of the artifact traceability graph

The flowchart of the methodology can be seen in Figure 4.1 and Figure 4.2, representing creation and usage respectively. All steps in the flowcharts take place inside the Software Artifact Analyzer tool.

Figure 4.2: Flowchart of the usage of our Expert Developer Finder algorithm.

As seen in Figure 4.1, the initial step of creation is data retrieval of the version and issue history of the software project. The retrieved data include commit, pull request, issue, and method history. All of the retrieved data is saved and later used for the construction of the artifact traceability graph. Then the developer names in the data are cleaned. In the second step, the retrieved data is used to create a representation of the software project in the form of an artifact traceability graph.

In the first step of the usage of our algorithm, a method, file, or folder name is retrieved from the user, as seen in Figure 4.2. Using this information, the expertise score of developers for the artifact in question is calculated. A metric called know-about score is used to quantify the expertise and it is calculated using the artifact traceability graph. In the last step of this portion, developers are ordered according to their expertise and displayed to the user.

4.1 Data Retrieval

The data retrieval process consists of four steps. These are retrieval of commit, pull request, issue history, and method change history data. For commit, pull request, and issue history, Perceval [51] is used, and for method change history, CodeShovel [50] is used. The pseudocode of the data retrieval can be seen in Algorithm 1.

Algorithm 1 Data Retrieval Process

$commits = Git(...).fetch()$
for `<commit in commits>` **do**
    $recordHEADCommitHash(commit)$
    $saveToFile(commit)$
end for

$prs = GitHub(...).fetch(category =' pull\_request')$
for `<pr in prs>` **do**
    $saveToFile(pr)$
end for

$issues = Jira(...).fetch()$
for `<issue in issues>` **do**
    $saveToFile(issue)$
end for

$filePaths \leftarrow getFilesFromCodeShovel(...)$
for `<filePath in filePaths>` **do**
    $methods \leftarrow getMethodsInFileFromCodeShovel(...)$
    for `<method in methods>` **do**
        $methodHistory \leftarrow getMethodHistory(...)$
        $saveToFile(methodHistory)$
    end for
end for

The commits and pull requests are retrieved from Git and GitHub via Perceval. For the retrieval of the commits, no branch is specified, and as a result, commits from all branches of the project are fetched. From the commit history, we record the commit hash that the HEAD points to. The issues are retrieved from Jira via Perceval.

The method change history is retrieved via CodeShovel. The REST API of CodeShovel is used to first retrieve all of the files in the software project. Then, for each file, the details of all of the methods inside the file are retrieved. Finally, the method change history is retrieved for each method found. In this final call, CodeShovel does a backward history traversal to find every change done to the method in question. The starting point of the traversal is the commit hash we found while retrieving the commit history.

The method history data we find is limited to the methods that exist in the software project at the time of data retrieval. This is caused by the fact that we first make calls to retrieve information about the files and methods and then use the information returned to retrieve the change history of a specific method. The information in question is the path to the method, the name of the method, and the start line of the method. As a result, the connection between past commits and any method that no longer exists is lost. However, it is important to mention that the connection between past commits and any file that no longer exists is not lost. The case is the same for deleted folders since the deleted files inside the folder are not lost.

Right after data retrieval is done, we utilize a functionality of SAA [36] that automatically groups developers with similar names. It is also possible to add more developers to a group or create new ones. This ensures developers who have different names in different data sources (GitHub and Jira) are not considered as unique developers multiple times.

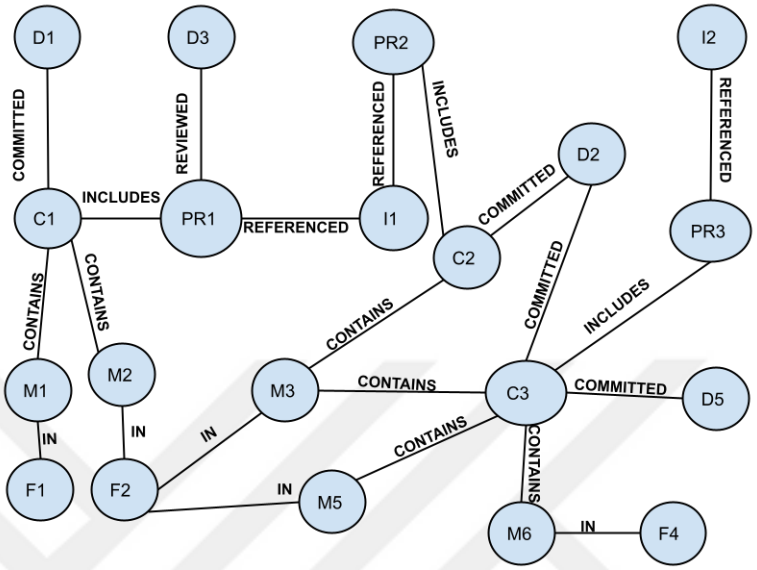

Figure 4.3: Sample artifact traceability graph. (C: commit, D: developer, I: issue, PR: pull request, F: file, M: method)

4.2 Artifact Traceability Graph

The artifact traceability graph is a data structure that represents the relationship between the artifacts in a software project. The nodes of the graph represent the various artifacts in a project. These artifacts are issues (I), commits (C), pull requests (PR), files (F), methods (M), and developers (D). The links represent the different relationships between the artifacts - such as referenced, committed, includes, etc.

A sample artifact traceability graph can be seen in Figure 4.3. The software project represented with this graph has two issues (I1 and I2). PR1 and PR2 reference the first issue and PR3 references the second one. PR1 is reviewed by D3 and includes C1. C1 is authored by D1 and contains changes to M1 and M2. M1 and M2 are in F1 and F2 respectively. F2 also includes M3 and M5. C2, which is included in PR2 and committed by D2, contains M3. D2 also committed C3, alongside D5, which contains M3, M5 and M6 (in F2, F2 and F4 respectively). C3 is included in PR3 which is referenced by I2.

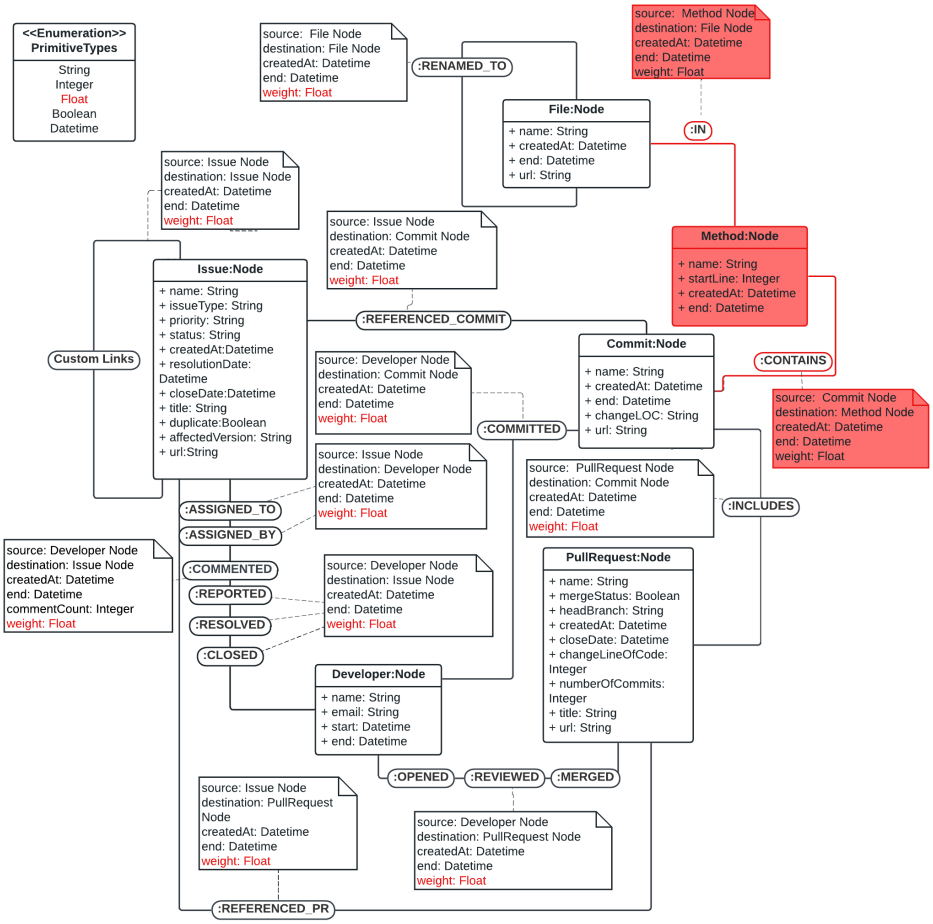

Figure 4.4: The nodes and links of an artifact traceability graph

4.2.1 Nodes and Links

All possible nodes and links in the artifact traceability graph presented in a class diagram can be seen in Figure 4.4. The classes, relationships, and attributes written in black were already proposed by the SAA framework and implemented in the example tool [36]. Everything else, written in red, is added in this work.

As seen in Figure 4.4, the following are the possible node types:

- **Developer:** They are authors of the code, reviewers of pull requests or creators and commenters of issues for the project, etc. The start attribute corresponds to the date when the developer first appeared in the project's

history. The end attribute is supposed to represent the end of the developer's involvement in the software project. In the current implementation, we assume that the developer might work on the project in the future so the end attribute stores a future date which is displayed as "-" in the UI.

- **Issue:** Contains information about a feature that will be implemented, a bug that needs to be updated, a piece of code that needs to be refactored, etc. It is possible for issues to be linked to other issues. For example, an issue might be linked to another issue that blocks the work needed for the first one. It is also possible for the issue to reference a commit. The createdAt date is the start date of the issue's lifecycle, and the closeDate is the end of its lifecycle.

- **Pull Request:** Displays the changes made on a branch in comparison to another branch. Pull requests are opened, merged and reviewed by developers and include multiple commits. The createdAt date is the start date of the pull request's lifecycle, and the closeDate is the end of its lifecycle.

- **Commit:** This is the set of changes made to the software project. They are committed by developers and multiple developers can be the author of a single commit. In this data structure, commits are directly linked to the methods they have changed and indirectly linked to the files they have changed (through the method node). The createdAt attribute corresponds to the date of the commit and is equal to the end attribute since they are instantaneous.

- **File:** Represents the files in a software project and they may include multiple methods. The name of a file node is the file path in the software project. In this data structure, when a file is renamed, it is linked to its older version (via RENAMED_TO). The createdAt attribute corresponds to the date the file first appeared in the project's history. The end attribute is supposed to represent the removal or renaming of the file. For files that still exist, it stores a future date which is displayed as "-" in the UI.

- **Method:** Represents the methods/functions in a software project. The name of a method node is the combination of the path of the file it is in, the

name of the method and the start line of the method. This name reflects the latest state of the method. The createdAt attribute corresponds to the date the method first appeared in the project's history. The end attribute is supposed to represent the removal of the method from the software project. In the current implementation, we are not tracking the deletion of methods and therefore we store a future date in the end attribute.

There is no separate node that represents a **Folder** in the software project. Instead, the folder information is accessible in the name attribute of both file and method nodes.

The possible links are:

- Source: **Developer** to Destination: **Issue**
  - COMMENTED
  - REPORTED
  - RESOLVED
  - CLOSED
- Source: **Developer** to Destination: **Pull Request**
  - OPENED
  - REVIEWED
  - MERGED
- Source: **Developer** to Destination: **Commit**
  - COMMITTED
- Source: **Issue** to Destination: **Developer**
  - ASSIGNED_TO
  - ASSIGNED_BY

- Source: **Issue** to Destination: **Issue**
  - Any custom link created in the issue management system.
- Source: **Issue** to Destination: **Pull Request**
  - REFERENCED_PR
- Source: **Issue** to Destination: **Commit**
  - REFERENCED_COMMIT
- Source: **Pull Request** to Destination: **Commit**
  - INCLUDES
- Source: **Commit** to Destination: **Method**
  - CONTAINS
- Source: **Method** to Destination: **File**
  - IN
- Source: **File** to Destination: **File**
  - RENAMED_TO

All links are directed from the source to the direction, as mentioned in the list above. However, it is possible to run queries on the graph without any direction restriction. Additionally, each link has a creation and an end date (createdAt and end respectively).

Links are weighted according to recency and a value we refer to as the weight coefficient. Recency is relevant for all links and represents the recency of the link's creation. The weight coefficient is relevant for the REVIEWED and COMMITTED relationships and is meant to represent the effect of these links on the expertise of the developer connected to them. This was implemented because, while the review action might increase the expertise of a developer for related

artifacts, we believe that it should not be equivalent to the expertise gained by directly committing a change to them. In addition, we believe that both links should contribute more to the expertise than any other link. The default value for the weight coefficient is 1.50 for COMMITTED, 1.25 for REVIEWED, and 1 for every other link. The weight of a link is the product of the weight coefficient and recency.

4.2.2 Change Cases

In this section, we present the change cases that may exist in a commit. We will mainly focus on the effect of these change cases on the commit, method, and file nodes and will not detail the creation of the pull request, developer, or issue nodes in detail. For simplicity, all of the graphs in this section will exclude the link labels that can be seen in Figure 4.3. The notations in the graph which are C, D, I, PR, F, and M correspond to commit, developer, issue, pull request, file, and method respectively. It is important to mention that any combination of the cases below may exist in a commit. For simplicity, we will be going through them one by one.

The initial state of the artifact traceability graph can be seen in Figure 4.5.

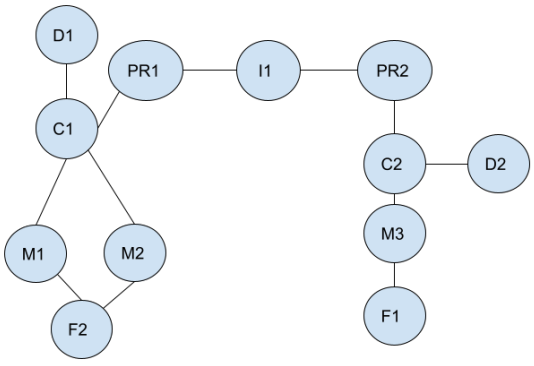

Figure 4.5: The initial state of an artifact traceability graph.

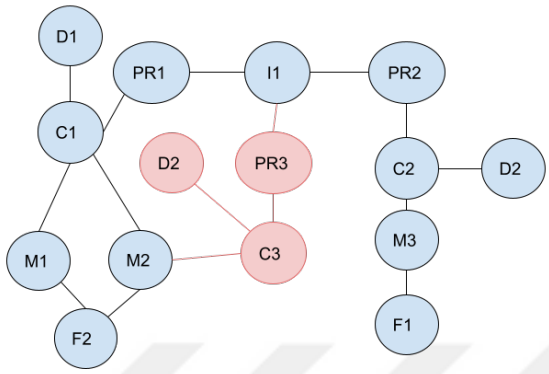

Figure 4.6: Resulting graph when new lines are added to M2 in F2 with C3.

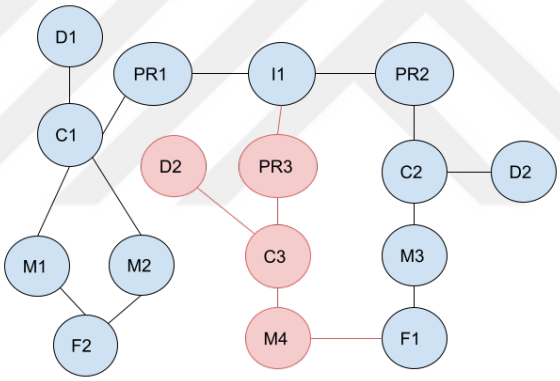

Figure 4.7: Resulting graph when the new method, M4, is added to F1 with C3.

I Only Addition

(a) *New line to an existing method.* An example can be seen in Figure 4.6.

(b) *New method to an existing file.* An example can be seen in Figure 4.7.

(c) *New file.* This is the case where a new file is added to the project without any method. So these are not detected in the method change history but in the commit history. We use the dummy method node to link the commit to the newly created file. An example can be seen in Figure 4.8. When a new method is later added to this file, the dummy method will still be linked to the initial commit and the file, and the new method will be linked to the second commit and the file.

(d) *New folder.* Since Git only tracks files and not directories, this case

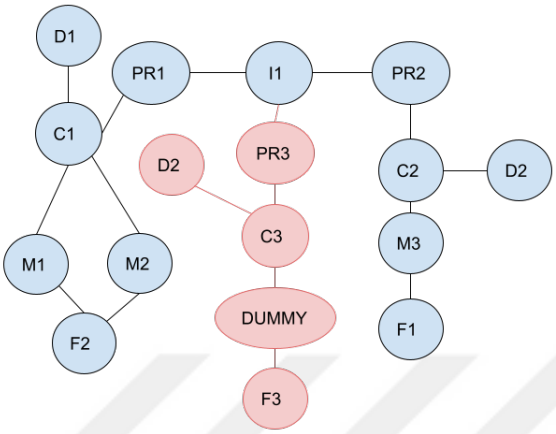

Figure 4.8: Resulting graph when the new file, F3, is added with C3.

is not detected until a file is also created within the folder. At that stage, the change case becomes Case I-C instead.

(e) *Blank line added.* We consider the addition of blank lines to a method as a trivial change that does not require any expertise in the method. However, we are not able to distinguish this case for files and therefore some expertise is gained for the affected file. An example can be seen in Figure 4.9.

(f) *Comment added.* Developers might add comments to better explain a piece of code, even if they were not the ones who wrote the code. The addition of such comments implies an investigation of the related method by the developer. Therefore expertise in the method is gained with this action. For example, if comments were added to M2 in F2, the resulting graph would be Figure 4.6. If the comment is not added to a specific method, the commit that introduced it is linked to the corresponding file via the dummy method.

II Only Deletion

(a) *Line deleted from a method.* Handled the same way as in Case I-A.

(b) *Method deleted.* The deletion of a method is not picked up as a change case by our algorithm. Since we rely on the current state of the project

to retrieve information about method changes, any method that currently does not exist is missing from the graph.

(c) *File deleted.* The deletion of a file is not picked up as a change case by our algorithm. The nodes for deleted files are present in the graph and are linked to the commits that worked on them via the dummy method. The deletion of any methods inside the file is handled the same way as in Case I-B.

(d) *Folder deleted.* As explained before, Git does not track directories, therefore the deletion of an empty folder is not detected. If the folder does contain files, their deletion is handled the same way as in Case I-C.

(e) *Blank line deleted.* Handled the same way as in Case I-E.

(f) *Comment deleted.* Handled the same way as in Case I-F.

III Modification

(a) *Method updated.* Handled the same way as in Case I-A.

(b) *Method renamed.* Handled the same way as in Case III-A. In addition, the name attribute is updated.

(c) *Method exception/parameter/modifier/return type changed.* Handled the same way as in Case III-A.

(d) *Method carried over inside the same file.* Moving a method inside the same file is not picked up as a change case in our algorithm.

(e) *Method moved to a different file.* When the method is moved to a different file, the method node is unlinked from the old file node and linked to the new one. Additionally, the name attribute of the method node is updated to include the path of the new file. An example can be seen in Figure 4.10.

(f) *File renamed.* An example can be seen in Figure 4.11. Note that, the name attribute of the method node is updated to include the new file name instead of the old one. Additionally, the method's link to the old file is removed when the change case is detected for that method.

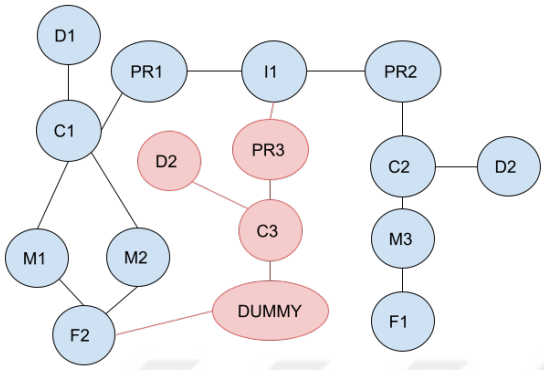

Figure 4.9: Resulting graph when a blank line is added to F2 with C3.

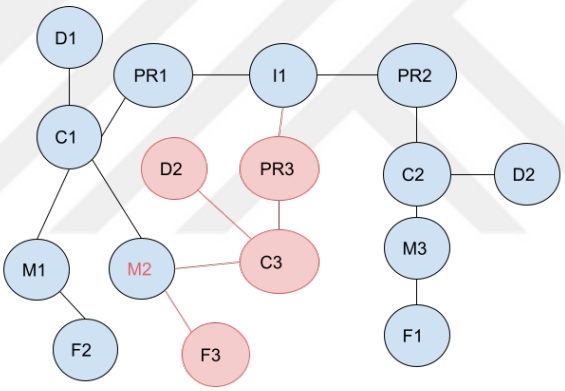

Figure 4.10: Resulting graph when M2 in F2 is moved to F3 with C3.

(g) *File moved to a different folder.* This change case is detected as a file rename in our algorithm. Therefore it is handled the same way as in Case III-F.

(h) *Folder renamed.* This change case is detected as a file rename in our algorithm. Therefore it is handled the same way as in Case III-F.

(i) *Comment updated.* Handled the same way as in Case I-F.

IV Other: Style Changes or Typo fix. In its current state, our algorithm does not distinguish these cases from any other update to a file or method. This is caused by the fact that no in-depth analysis is done on the intent of the change. Therefore, these changes are considered equal to an intentional implementation change.

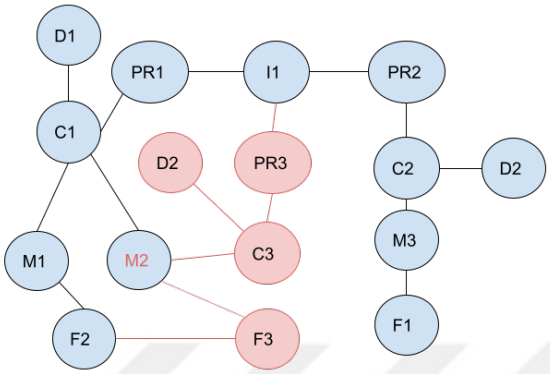

Figure 4.11: Resulting graph when F2 is renamed to F3 with C3.

4.2.3 Review Cases

In this section, we present the cases that exist in the review process of a pull request. We will mainly focus on the effect of these review cases on the developer and pull request nodes and will not detail the creation of any other node.

In this study, a developer who participates in the review process for an artifact gains expertise on the said artifact. However, we do not believe this expertise is equivalent to the expertise gained via actively working on the artifact. Therefore, as mentioned before, the COMMITTED and REVIEWED links have different weights.

The review cases are the following:

I Request to review is sent to the developer.

II The developer approved the PR.

III The developer requested changes to the PR.

IV The developer commented on the PR.

We consider Cases II, III and IV as **review** actions. If one, or more, of them are present then the developer is considered a reviewer of the pull request. It should

be noted that we do not consider simply opening or merging a pull request as a review action.

4.2.4 Construction

The retrieved data is processed to construct the artifact traceability graph in the following order:

1. commit history from Git,
2. method change history from CodeShovel,
3. pull request history from GitHub and
4. issue history from Jira.

For commits, pull requests and Jira issues the source class packages introduced by Merdol et al. [37] are used. These classes include functions that allow us to quickly retrieve information such as the files that were modified in a commit, the developers who reviewed a PR or developers who commented on an issue. We expand these source classes to reflect the change cases presented in the previous sections. For the method change history, we developed a new source class called *CodeShovelMethodHistory*. This class represents the entire history of a single method and contains a list of *CodeShovelMethodChange* instances. For each change, we have functions that determine the type of the change and get the method's file path, name, and start line.

The first step of graph construction is processing the commit history. In this step, the commit nodes are created and linked to their respective developer. Then, the files that are modified in each commit are created and linked to the dummy method which is then linked to the respective commit. This dummy method node is necessary because in our structure files are not directly linked to the commit, and at this stage, we are not processing any data about the methods themselves. In addition, the files that were renamed are linked to their older version. If

the commits reference an issue, these issue nodes are created and linked to the respective commit.

The second step of the construction is processing the method change history. For each method change the following are the relevant details for our algorithm:

- type (classified by CodeShovel [50])
- commitDate
- commitName (in other words commitHash)
- path
- functionStartLine
- extendedDetails (includes the old values for method name and file path and etc.)

For change history categorized as multichange, the data is structured differently but the information above is still accessible. As mentioned before, this is not an exhaustive list of the information provided, but this is the information that we utilize.

When processing the method change history, we first unlink the commit and file in question from the dummy node. Since these commit and file nodes will now be linked to an actual method in the project. The next steps vary according to the change type. Table 3.1 provides the change type returned from CodeShovel for each change case and also presents the action we take.

The action categories for method changes are:

- **Initialization:** This is the first occurrence of a method. The commit and file node should already exist. The method node is created and linked to the relevant commit and file (via CONTAINS and IN respectively).

- **Update:** This corresponds to any update that is made to a method that does not change its location. The method, commit, and file node should already exist. The commit node is linked to the method node (via CONTAINS). The name attribute of the method is updated in case the name or start line has changed with the update.
- **Move:** This corresponds to moving the method to a different file. The method, commit, and file node should already exist. The commit node is linked to the method node (via CONTAINS). The method node is unlinked from the old file node and linked to the new file node (via IN). Additionally, the name attribute of the method node is updated.
- **FileRename:** The creation of the link between renamed files is handled while processing the commit history. For the method, the commit node is linked to the method node (via CONTAINS) and the name attribute of the method node is updated.
- **Ignore:** No change should be registered for the method.

Next, the pull request history is processed. Here, the pull request node is created and linked to the developer/s who opened, closed, and reviewed the PR. Then, the PR is linked to the commits that are included in it. If the PR is referenced by an issue, the issue node is created and linked to the respective pull request.

Then, the Jira issue history is processed. First, the attributes of any issue node that was created in the previous steps are filled, and the issue is linked to the developer who reported and, if relevant, closed the issue. Additionally, the issue is linked to the developers who are assigned to it, assigned somebody to it, resolved it, and commented on it. Again, if the issue references any pull request, they are also linked to it. Finally, any link between issues (such as blocks, fixes, duplicates, etc) is created.

Once all nodes and the links between them are created the following steps are taken:

- The start/createdAt and end attributes of developer, file, and method nodes are updated.
- Any developer without a name is removed from the graph.
- Weight is added to all links.
- The renamed files are updated to point to the most recent successor.

This concludes the construction of the artifact traceability graph.

4.3 Expert Developer Finder

The Expert Developer Finder algorithm recommends the developers with the most knowledge about a code segment. The actions that contribute to the expertise of the developer are the following:

- Commit to the code segment
- Commit to the previous version of the code segment
- Review of the Pull Request that includes a commit to the code segment

It is important to note that issues are excluded from contributing to the developer's expertise. In addition, contribution to an older version of the code segment (other than the very last version) does not contribute to the developer's expertise, and reviews of older versions of the code segment are also excluded.

In this section, we detail the score we use to quantify the developer's expertise, how the Expert Developer algorithm works, and, finally, how the algorithm can be used via the SAA tool.

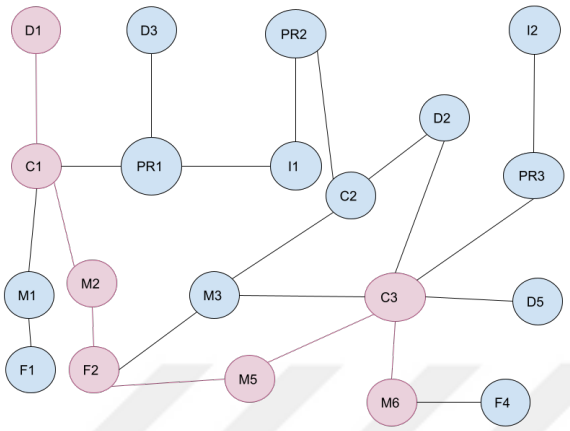

Figure 4.12: A distinct but invalid path from D1 to M6.

4.3.1 Know-About Score

The metric we use to quantify the expertise of a developer is the know-about score, first introduced by Sülün et al. [45]. The know-about score is calculated by finding the distinct paths between a source (S) and a target node (T) in the artifact traceability graph and summing the inverse of the length of the paths. The source node could be a file or a method node and the target node is a developer. The formula for the know-about score can be seen in Equation 4.1.

$$KA_{S,T} = \sum_{i} \frac{1}{Length(Path_i(S,T))} \tag{4.1}$$

When the code segment in question is a method, there is a single source node in the artifact traceability graph. In this case, a valid distinct path between a developer and method can not contain another method node. A path that is invalid according to this restriction can be seen in Figure 4.12. This is a problematic path because it makes it look like D1 does have some knowledge of M6 even though they never worked on it directly. Therefore, such paths need to be ignored while calculating the know-about score.

When the code segment in question is a file, there is again a single source node in the artifact traceability graph. In this case, a valid distinct path between

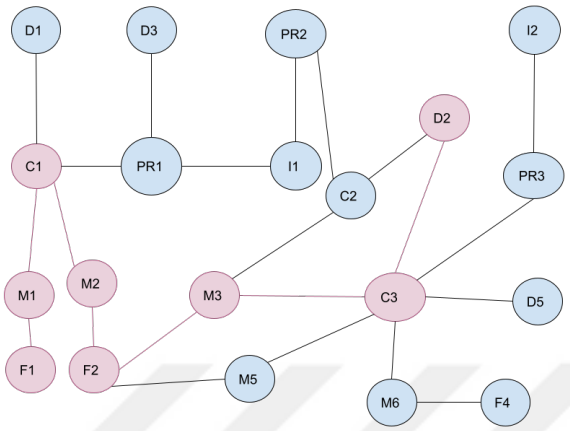

Figure 4.13: A distinct but invalid path from D2 to F1.

a developer and a file can not contain another file node. A path that is invalid according to this restriction can be seen in Figure 4.13. This is a problematic path because it makes it look like D2 does have some knowledge of F1 even though they never worked on it directly. Therefore, such paths need to be ignored while calculating the know-about score.

To ensure that the invalid paths mentioned in the previous paragraphs do not affect the know-about score, we specify a length limit which limits the length of the distinct path between a developer and a file/method. In accordance with the list of actions that contribute to the expertise, explained at the beginning of this section, and the invalid paths explained before, we select the length limit as three for methods and four for files.

As it can be seen in Figure 4.14 and Figure 4.15 any direct contribution via a commit or PR review to the file and method in question is covered by the length limit. In addition, as shown in Figure 4.16 a direct commit to the previous version of the artifact is also covered. However, due to the length limit, the path from a developer to an artifact that changed twice is considered as an inapplicable path even though it is technically distinct and valid. In addition, the path from a developer who is related to the artifact via a PR and an older version of the artifact is inapplicable. These cases can be seen in Figure 4.17 and Figure 4.18 respectively.

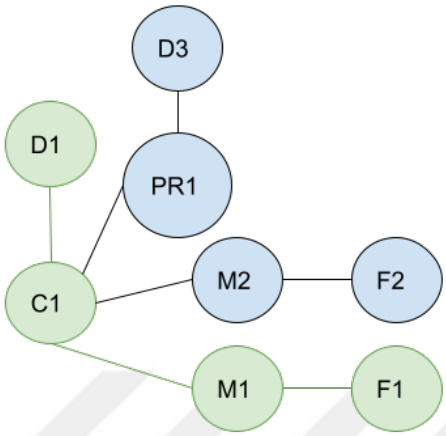

Figure 4.14: A distinct, valid, and applicable path from D1 to M1 and D1 to F1.

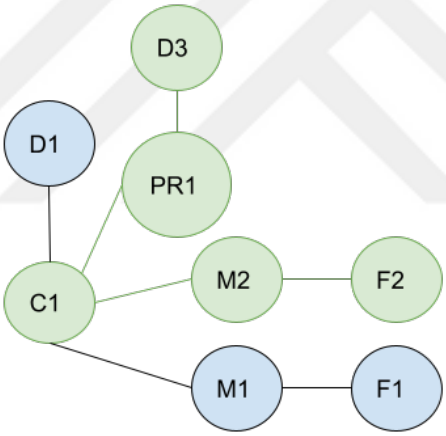

Figure 4.15: A distinct, valid, and applicable path from D3 to M2 and D3 to F2.

When the code segment in question is a folder there is no single source node. For folders, the know-about score is calculated for each file in the folder and the know-about score for the folder is the average of file scores. The files in a folder are retrieved via the name attribute of the file node.

The weight attribute of the links affects the know-about score according to the formula seen in Equation 4.2.

$$KA_{Weight} = \sum_{i} \frac{\prod_{j} Weight(Link_{ij})}{Length(Path_i(S,T))} \tag{4.2}$$

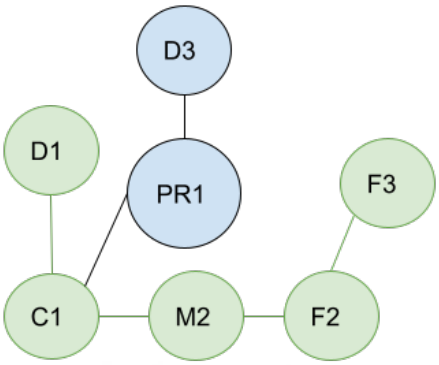

Figure 4.16: A distinct, valid, and applicable path from D1 to F3.

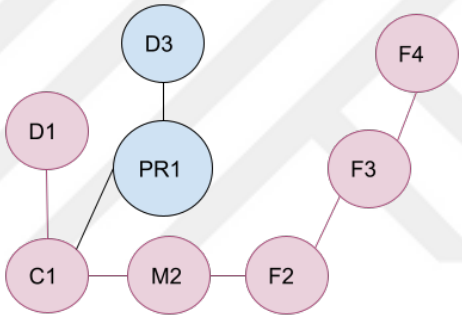

Figure 4.17: A distinct and valid but inapplicable path from D1 to F4.

As explained before, the weight in Equation 4.2 corresponds to the product of recency and weight coefficient. The weight coefficient is 1.5 for COMMITTED, 1.25 for REVIEWED and 1 for every other link. The recency is calculated in the same fashion as Sülün et al. [20], as seen in Equation 4.3.

$$Recency = 1 - \frac{T_{creation}}{T_{total}} \tag{4.3}$$

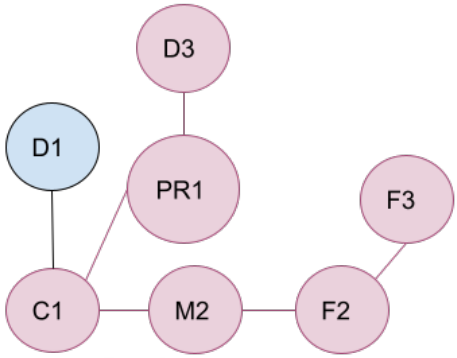

Figure 4.18: A distinct and valid but inapplicable path from D3 to F3.

4.3.2 Expert Developer Finder Algorithm

The algorithm differs slightly depending on the artifact for which experts are found. The pseudocode when the artifact chosen is a file or method can be seen in Algorithm 2.

Algorithm 2 Expert Developer Finder Algorithm for Files and Methods

$devIds = runQuery(" MATCH(a: \quad Artifact) \quad WHERE \quad elementId(..)..")$

$results = runQuery(" CALL \quad findNodesWithMostPathBetween...")$

$experts = results.getExperts()$
$scores = results.getScores()$

Finding experts for folders differs from finding experts for files/methods because folders are not represented as nodes in the graph. Therefore, in this case, the experts for each file in the folder are found, and the expert's score for each file is summed up and divided by the number of files. Then the top developers according to the average score is returned. The pseudocode can be seen in Algorithm 3.

The first query in Algorithm 2 finds the developers who are candidates for being experts, so these are the developers who are within the length limit of the artifact. For folders, this is the second query in Algorithm 3. The details of this step were previously explained in Section 3.1.

Algorithm 3 Expert Developer Finder Algorithm for Folders

$files = runQuery("MATCH(f:File)\ WHERE\ f.name$
$\quad CONTAINS\ folderName..")$

for file in files **do**

$\quad devIds = runQuery("MATCH(f:File)..")$

$\quad results = runQuery("CALL\ \ findNodesWithMostPathBetween...")$

$\quad experts = results.getExperts()$

$\quad scores = results.getScores()$

for $i = 0; i < experts.length; i++$ **do**

$\quad\quad devsAndScores[experts[i]] += scores[i]$

end for

$calculateAverage(devsAndScores)$

end for

As seen in both algorithms, the ***findNodesWithMostPathBetween*** call is where the know-about score is calculated and the experts are returned with their score. The specific parameters of this call can be seen below.

findNodesWithMostPathBetween Call

$CALL\ \ findNodesWithMostPathBetween([artifactId],\ [OPENED,$
$\quad MERGED, ASSIGNED\_BY, ASSIGNED\_TO, BLOCKS, CLOSED,$
$\quad COMMENTED, DEPENDS\_UPON, DUPLICATES, FIXES,$
$\quad INCORPORATES, IS\_A\_CLONE\_OF, REFERENCED, RELATES\_TO,$
$\quad REPORTED, RESOLVED, SUPERSEDES], possibleDevIds, weight,$
$\quad lengthLimit, nodeLimit, false, pageSize, currPage, null, false, orderBy,$
$\quad orderDir, timeMapping, startTime, endTime, inclusionType, timeout, idFilter)$

All relationships that are related to the Issue node and OPENED and MERGED relationships are specified as the *ignoredTypes* in this procedure call. This means that these relationships will not exist in the paths discovered. So if a developer is related to an artifact only through these relationships, they will not be considered an expert. For example, a developer who commented on an issue

that references a commit that changed the artifact will not be recommended as an expert for that artifact. As mentioned before, the *lengthLimit* is three for methods and four for files. The *nodeLimit* corresponds to the number of results returned and is, in most cases, determined by the user. To ensure any common bots, such as dependabot, are not returned as experts, we run the query with the *nodeLimit* specified by the user plus one. In case a bot is found in the results, they are removed from the ranking and the other developers below them are moved up. If no bot is found, developers in the amount specified by the user are recommended. The default value for the nodeLimit is four, meaning three developers are recommended by the algorithm by default. The parameters after the *nodeLimit* are related to user preferences, formatting the results, etc., and will not be explained in more detail.

The details of ***findNodesWithMostPathBetween*** were previously explained in Section 3.1.

4.3.3 Usage

The algorithm to find experts is triggered from the SAA application. To find experts for files or methods, the user first clicks on a file or method in the graph. This displays a menu, seen on Figure 4.19 and 4.20 respectively, which includes the queries that can be run on the selected artifact. The users can specify the number of experts that should be returned by the algorithm. Additionally, the users can select another artifact, of the same type, from this view to run the query for. This list of files or methods is retrieved from the graph directly.

After the algorithm is run and results are found, the experts are displayed with their score, as in Figure 3.2. For files and methods, the graph is also updated to visualize the experts and their connection to the artifact.

Since folders are not represented in the graph, the *Get Experts* query is run from the *Database* tab of SAA. This can be seen in Figure 4.21. The users are able to select the folder to run the query on from the dropdown menu. The list

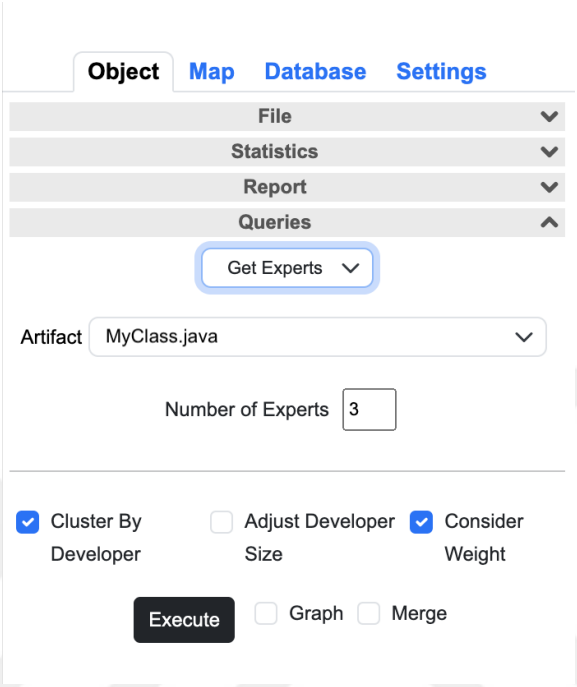

Figure 4.19: The menu to run *Get Experts* for files.

of folders is retrieved by extracting folder names from the files in the graph. As before, the users can specify the number of experts to be recommended. After the algorithm is run and results are found, the experts are displayed with their score, as in Figure 3.2. For folders, no visual changes are applied when presenting the results.

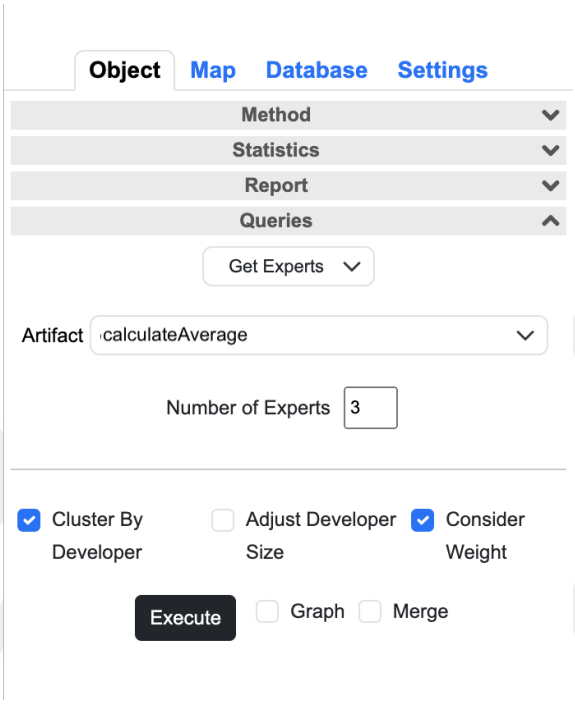

Figure 4.20: The menu to run *Get Experts* for methods.

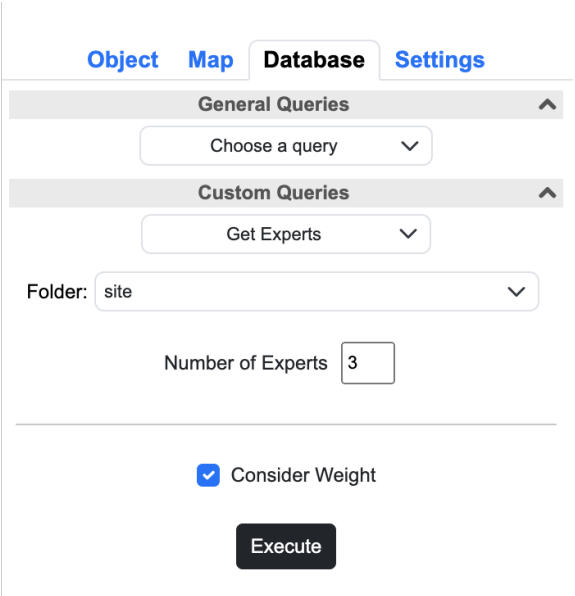

Figure 4.21: The menu to run *Get Experts* for folders .

Chapter 5

Validation Strategy

In this section, we explain the open-source projects selected for validation and the validation strategy of our work.

5.1 Open Source Project Selection

Expert Developer Finder algorithm is evaluated with three open-source projects. In this section, we explain our criteria for choosing them. We made our selection from Apache's open-source projects since a significant portion of their projects use Java. The main points of our criteria are the following:

1. The project uses GitHub for version control and code review.
2. The project uses Jira to keep track of their issues.
3. The project's main language is Java, which constitutes at least 90% of the repository code.
4. The project has 1,000 or more stars.
5. The project has 500 or more pull requests (opened and closed combined).

6. The project has more than 2,000 commits.

7. The project has more than 50 contributors.

In addition, after constructing the artifact traceability graph of the candidate projects, we checked the relationships in the graph. If either REFERENCED_PR (from Issue to Pull Request) or the REFERENCED_COMMIT (from Issue to Commit) relationship did not exist in the graph, the project in question was not selected. This is due to our usage of issue comments during the validation process, which will be explained in the next section.

In light of these criteria, here are the projects selected for validation:

- Apache Nutch [56]
- Apache OpenNLP [57]
- Apache Curator [58]

The properties of these projects, retrieved in July 2024, can be seen in Table 5.1.

Table 5.1: Properties of the open-source projects selected for validation (July 2024)

Project Name	**Stars**	**PR (open and closed)**	**Commits in Master Branch**	**Contributors from Github**
Apache Nutch	2.9K	808	3445	50
Apache OpenNLP	1.4K	641	2166	53
Apache Curator	3.1K	504	2838	128

5.2 Validation

Issues include information about enhancement requests, refactoring, tasks, bugs in a code piece, etc. In our study, we believe that they can be used to validate

the results of our algorithm. We propose that developers who leave comments under issues are knowledgeable about the software artifacts related to the issue, and therefore, we can assume they have expertise on the artifact. For validation, we compare the ranking of commenters on issues related to the artifact to the recommendations given by our algorithm for that artifact. It is important to note that in our algorithm any contribution to an issue does not contribute to the developer's expertise and therefore their recommendation. Thus, we believe it is feasible to use commenters to validate the algorithm results.

We implement a simple Python application to automatically run the validation for every folder, file, and method in a project. At the stage where this application is run, we assume that the artifact traceability graph of the project is already created.

The first step of the validation process is finding the artifacts for which validation will be run. For methods, we get the names of all method nodes that exist in the artifact traceability graph, other than the dummy method. For files, we utilize CodeShovel's *listFiles* API [50] to get all Java files that exist in the project at the time of validation. For folders, we again utilize the *listFiles* API and extract the folders from the names of the related file nodes in the artifact traceability graph. The query we run for the second step can be seen in Algorithm 4.

Algorithm 4 Get Folder Names from Artifact Traceability Graph

```
MATCH (f:File) WITH split(f.name, '/') as fileName, elementId(f) as id
WHERE size(fileName) >1 AND f.name IN fileList
WITH collect(fileName) as splitFileNames UNWIND splitFileNames as name
WITH DISTINCT name[0..size(name)-1] as folders
WITH apoc.text.join([i IN folders], '/') as name
RETURN name
```

Then for each artifact, we find the weighted and unweighted experts. For methods and files, we use Algorithm 2 and for folders we use Algorithm 3.

To find related issue commenters, we first query the software artifact traceability graph to get the issues related to a specific artifact. Then, we retrieve the authors of the comments made under the issues in question. The developers retrieved in this manner are ranked according to the number of times they commented on issues. The COMMENTED relationship between an issue and a developer also keeps track of the number of comments left on that issue by the developer. So both the number of comments they left on one issue related to an artifact and the number of different issues related to the artifact they commented on affect the ranking. In addition, since we use Jira to obtain the issues, and Jira can also be utilized by non-technical team members, we believe the commenters who have not contributed to the code base should be removed for validation. To ensure this we require the issue commenters to have a COMMITTED relationship to a commit for **any** artifact in the graph.

For example, let us consider a software project with three members: Alice, Jane, and Joe, and an artifact linked to Issues 1 and 2. The software artifact graph of such a project can be seen in Figure 5.1. For simplicity, the graph does not contain the link labels that can be seen in Figure 4.3. The ranking of developers according to issue comments for F1 during the validation purposes, is

1. Joe
2. Jane

As expected, Alice is not present in the ranking because she does not have a COMMITTED relationship in the graph. And Joe is ranked above Jane because he has commented on both Issue 1 and Issue 2 while Jane has only commented on Issue 1.

First, we run a query to find all the issues connected to a file. This query for files can be seen in Algorithm 6 in Appendix A. Then, we find all developers who commented on the issues found with the previous query and rank them according to their comment counts. This query can be seen in Algorithm 5 which is the same for all artifacts. For simplicity, we have only shared the query for files.

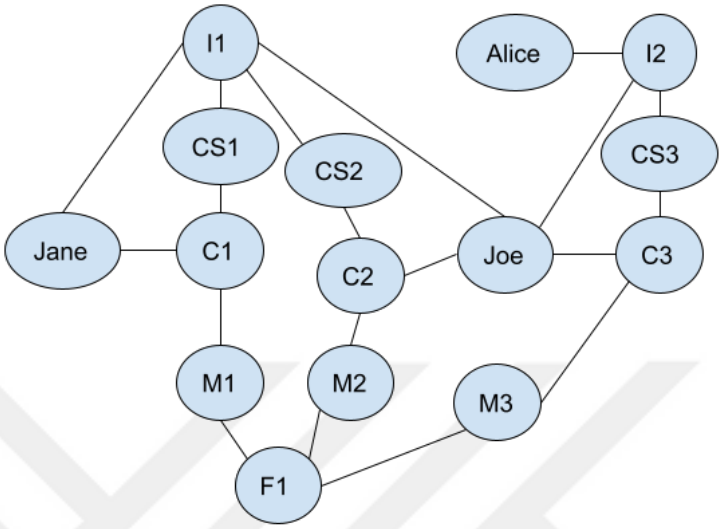

Figure 5.1: Sample artifact traceability graph. (C: commit, D: developer, I: issue, PR: pull request, F: file, M: method)

Once the weighted and unweighted experts and the issue commenters are found, they are logged into a file in the form of a dictionary with the keys: artifact, algo_weight, algo_without_weight, issue_ids, issue_comments. Respectively, they correspond to the artifact name, weighted experts, unweighted experts, related issue IDs, and the ranking of issue commenters.

5.3 Accuracy Calculation

We use a Top-k accuracy approach to measure the correlation between the rankings (top issue commenters vs. experts found by our algorithm - weighted and unweighted).

We calculate Top-1-Top-1, Top-1-Top-3, Top-1-Top-5, Top-3-Top-3, Top-3-Top-5, and Top-5-Top-5 accuracy for the rankings. The first Top-k in the pair corresponds to the issue commenters and the second Top-k corresponds to the experts found by our algorithm. For example, for Top-1-Top-3 accuracy, we compare the top issue commenter to the Top-3 developers recommended by our algorithm. Another example is for Top-3-Top-5, where we compare the Top-3

Algorithm 5 Get Developer Comment Count Per Issue from Artifact Traceability Graph

```
MATCH (i:Issue) - [r:COMMENTED] - (d:Developer) WHERE elementId(i) IN $issue_ids
WITH r, collect(d) as devList
CALL {
  MATCH (d2:Developer) - [r2:COMMITTED] - ()
  RETURN collect(d2) as committedDevList
}
WITH r, [dev IN devList WHERE dev IN committedDevList — dev] as commentators
UNWIND commentators as commentator
WITH r, commentator.name as dev_name
WHERE startNode(r).name = dev_name
RETURN sum(r.commentCount) as comment_count, dev_name
ORDER BY comment_count DESC
```

issue commenter to the Top-5 developers recommended by our algorithm. The formula for this calculation can be seen in Equation 5.1. In this equation, RC is the ranked issue commenters and RD is the list of ranked developers. The Top-k calculation constant is represented with k and N is the number of issue commenters for the current instance. The calculation process is the same for experts found with and without weight.

$$Accuracy(k_1, k_2) = \sum_{i=1}^{N} \frac{\mathrm{RC}_{k_1} \cap \mathrm{RD}_{k_2}}{N} \tag{5.1}$$

It should also be noted that the Top-k-Top-k accuracy calculation is done if the number of developers recommended by our algorithm is more than or equal to k and the same goes for issue commenters. The cases where this is not satisfied are ignored during calculation.

Chapter 6

Results

In this section, we share the results of our validation presented under the research question they relate to.

We tested our algorithm with three open-source projects - Nutch, OpenNLP, and Curator. Information about the artifact traceability graph constructed for each project can be seen in Table 6.1. The average number of nodes in the graph is 14830 and the average number of relationships is 40893.

Table 6.1: Number of nodes and relationships in the artifact traceability graphs created for validation

Project	Number of Nodes in Graph	Number of Relationships in Graph
Nutch	17694	49962
OpenNLP	14581	38801
Curator	12214	33917

For validation, we compared the experts we found for an artifact with the top commenters on issues related to the artifact in question. We calculated the Top-1-Top-1, Top-1-Top-3, Top-1-Top-5, Top-3-Top-3, Top-3-Top-5, and Top-5-Top-5 accuracy of each artifact type (folder, file, method) and both for experts recommended with and without the effect of weights. The average accuracy for each artifact type can be seen in Table 6.2.

Table 6.2: Average accuracy of three open source projects for recommendations given for each artifact type

		Weighted Recommendations			Unweighted Recommendations		
Project	**Issue Commenters**	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Folder	Top-1	0.12	0.55	0.84	0.32	0.71	0.84
	Top-3	-	0.46	0.66	-	0.54	0.68
	Top-5	-	-	0.56	-	-	0.57
File	Top-1	0.14	0.56	0.82	0.36	0.75	0.84
	Top-3	-	0.46	0.65	-	0.52	0.69
	Top-5	-	-	0.52	-	-	0.57
Method	Top-1	0.27	0.66	0.88	0.39	0.74	0.93
	Top-3	-	0.52	0.72	-	0.59	0.78
	Top-5	-	-	0.56	-	-	0.61

6.1 RQ1: To what extent can the history of a software project be utilized to accurately find expert developers for a software folder?

The accuracy results for recommendations given for folders in three open-source projects can be seen in Table 6.3.

The recommendations for Nutch have 92% accuracy for Top-1-Top-5 accuracy with respect to both unweighted and weighted recommendations. The lowest accuracies were 25% and 26% for Top-1-Top-1 weighted and unweighted recommendations respectively. For weighted recommendations, the remaining accuracies ranged from 69% to 89% with 3 out of 4 exceeding 80%. For unweighted recommendations, the range was from 64% to 89% with 3 out of 4 being less than or equal to 70%. Therefore, it can be said that, for this project, the weighted recommendations had higher accuracy.

For folders in the OpenNLP project, the highest accuracy is 85% for Top-1-Top-5 accuracy with weighted recommendations. The accuracy for the corresponding unweighted recommendation is 84%. The lowest accuracy was for Top-1-Top-1 with 8% for weighted recommendations. For unweighted recommendations, the lowest accuracy was for Top-3-Top-3 with 54% accuracy. The

Table 6.3: Accuracy results of recommendations given for folders in three projects

		Weighted Recommendations			**Unweighted Recommendations**		
Project	**Issue Commenters**	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Nutch	Top-1	0.25	0.86	0.92	0.26	0.67	0.92
	Top-3	-	0.82	0.89	-	0.64	0.89
	Top-5	-	-	0.69	-	-	0.70
OpenNLP	Top-1	0.08	0.74	0.85	0.65	0.73	0.84
	Top-3	-	0.40	0.63	-	0.54	0.62
	Top-5	-	-	0.56	-	-	0.55
Curator	Top-1	0.03	0.04	0.76	0.06	0.72	0.77
	Top-3	-	0.16	0.47	-	0.44	0.54
	Top-5	-	-	0.41	-	-	0.46

remaining accuracies ranged from 40% to 74% for weighted recommendations and from 55% to 73% for unweighted recommendations. The most significant difference in the accuracies between weighted and unweighted recommendations is for Top-1-Top-1 accuracy (8% and 65% respectively). The remaining accuracies did not differ this drastically. However, it can still be said that for this project, the unweighted recommendations performed better.

With the Curator project, the highest accuracy for folders is 77% with respect to unweighted recommendations. For weighted recommendations the highest accuracy is 76%. Both are for Top-1-Top-5 accuracy. The lowest accuracies found were for Top-1-Top-1 accuracy with 3% and 6% for weighted and unweighted recommendations respectively. For weighted recommendations, the remaining accuracies ranged from 4% to 47%, and for unweighted recommendations, the accuracy range is from 44% to 72%. As can be seen, for this project, the unweighted recommendations had better accuracy. It should also be noted that the lowest accuracy we recorded for folders and the lowest highest accuracy recorded for folders is for this project. As seen in Table 5.1, this is the project with the most number of developers which might explain the low accuracy for this project in particular.

The highest accuracy we reached for folders amongst all software projects is 92% with Nutch and the lowest accuracy is 3% with Curator. The former is for Top-1-Top-5 and the latter is for Top-1-Top-1. Out of the 36 accuracies we

calculated, 8 of them exceed 80% and 20 of them exceed 60%. 12 out of 36 are under 50% - with 8 of them being from the Curator project. Out of the three projects we evaluated our algorithm with, two of them had higher accuracy for unweighted recommendations. We believe this might be affected by the result strategy we have chosen. Since for top issue commenters we only take into account the number of comments and not their recency, it is understandable that the recommendations that do take recency into account are less related to the top issue commenters.

Referring back to Table 6.1, it can be seen that projects with more nodes and relationships had higher accuracy. In addition, the project with the smallest number of nodes and relationships, Curator, had the lowest accuracies Therefore, we can conclude that the history of a software project can be utilized to find experts for a software folder, and with more historical data higher accuracies can be reached.

On average, the highest accuracy we reached was 84% for Top-1-Top-5 with respect to both weighted and unweighted recommendations. The lowest average accuracy is 12% for Top-1-Top-1 accuracy for weighted recommendations. Considering the averages, the unweighted recommendations had higher accuracy than weighted recommendations - except for Top-1-Top-5 where the accuracy is the same. We believe that reaching 84% average Top-1-Top-5 accuracy shows that our algorithm can recommend accurate expert developers for a given folder.

6.2 RQ2: To what extent can the history of a software project be utilized to accurately find expert developers for a software file?

The accuracy results for recommendations given for files in three open-source projects can be seen in Table 6.4.

Table 6.4: Accuracy results of recommendations given for files in three projects

		Weighted Recommendations			**Unweighted Recommendations**		
Project	**Issue Commenters**	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Nutch	Top-1	0.28	0.86	0.91	0.29	0.78	0.91
	Top-3	-	0.77	0.85	-	0.63	0.85
	Top-5	-	-	0.65	-	-	0.66
OpenNLP	Top-1	0.10	0.60	0.81	0.56	0.72	0.79
	Top-3	-	0.40	0.65	-	0.51	0.65
	Top-5	-	-	0.57	-	-	0.61
Curator	Top-1	0.05	0.21	0.75	0.22	0.75	0.82
	Top-3	-	0.20	0.47	-	0.42	0.58
	Top-5	-	-	0.33	-	-	0.46

The expert developer recommendations for files in the Nutch project had the highest accuracy for Top-1-Top-5, with 91% accuracy for both weighted and unweighted recommendations. The lowest accuracies were for Top-1-Top-1 for both weighted and unweighted recommendations, with 28% and 29% accuracy respectively. The file accuracy for the remaining ranged from 65% to 86% for weighted recommendations and from 63% to 85% for unweighted recommendations. For weighted recommendations, 2 out of 4 accuracies exceed 80% while for unweighted recommendations this is 1 out of 4. Even though the difference is small, the weighted recommendations performed better for Nutch - as was the case for folders.

The recommendations for OpenNLP reach 81% accuracy for Top-1-Top-5 with respect to weighted recommendations and 79% for unweighted recommendations. The lowest accuracy was for Top1-Top1 calculation for weighted recommendations with 10% accuracy. The lowest accuracy for unweighted recommendations is 51% for Top-3-Top-3 accuracy. The remaining accuracies ranged from 40% to 65% for weighted recommendations and from 56% to 72% for unweighted recommendations. Overall, it can be said that, for OpenNLP, our algorithm reached better accuracy with unweighted recommendations.

For the Curator, the highest accuracy for files is 82% for unweighted recommendations. This was for the Top-1-Top-5 accuracy. The highest accuracy for

weighted recommendations is 75% again for Top-1-Top-5 accuracy. The lowest accuracy is 5% with respect to weighted recommendations Top-1-Top-1 accuracy. As mentioned in the previous section, the low accuracy for this project can be explained by the fact that this is the project with the most number of contributors - as seen in Table 5.1. For unweighted recommendations, the lowest accuracy is 22% again for Top-1-Top-1 accuracy. For weighted recommendations, the remaining accuracies ranged from 20% to 47%. For unweighted recommendations, the remaining accuracies ranged from 42% to 75%. As can be seen, for Curator, the unweighted recommendations performed better.

The highest accuracy we reached amongst all software projects is 91% with Nutch and the lowest accuracy is 5% with Curator. The former is for Top-1-Top-5 and the latter is for Top-1-Top-1. Out of the 36 accuracies we calculated, only 7 exceed 80% and 20 of them exceed 60%. 12 out of 36 are under 50% - with 8 of them being from Curator, the project with the smallest graph amongst the projects we selected. As was the case for folders, the unweighted recommendations for files performed better than weighted recommendations. Our explanation for this difference can be seen in the previous subsection.

It can be seen that projects with more nodes and relationships reached higher accuracy, as was the case for folders. Therefore, we can conclude that the history of a software project can be utilized to find experts for a file, and greater accuracy can be achieved with more extensive historical data.

The highest average accuracy we reached concerning three software projects is 84% for unweighted recommendations for files. For weighted recommendations, the highest accuracy reached was 82%. Both were for the Top-1-Top-5 accuracy. The lowest average accuracy is 14% for Top-1-Top-1 accuracy for weighted recommendations. For unweighted recommendations, the lowest accuracy is 36%, again for Top-1-Top-1 accuracy. Considering the average accuracy, unweighted recommendations had higher accuracy than weighted recommendations. Since we reached 84% Top-1-Top-5 accuracy on average, we believe it can be said that our algorithm recommends accurate experts for files.

6.3 RQ3: To what extent can the history of a software project be utilized to accurately find expert developers for a software method?

The accuracy results of recommendations given for methods in three open-source projects can be seen in Table 6.5.

For Nutch, the Top-1-Top-1 accuracy for recommendations with weight is 41% and it is 43% for recommendations without weights. These are the highest accuracies reached for Top-1-Top-1 accuracy amongst the three artifacts. However, these are still the lowest accuracy values for methods. The highest accuracy is 94% for Top-1-Top-5 for unweighted recommendations and 93% for weighted recommendations. The method accuracy for the remaining ranged from 57% to 81% for weighted recommendations and from 58% to 81% for unweighted recommendations. For both only 1 out of 4 exceeds 80% and 2 out of 4 exceeds 70%. Even though the difference is small, the unweighted recommendations for methods in Nutch performed better.

Table 6.5: Accuracy results of recommendations given for methods in three projects

		Weighted Recommendations			**Unweighted Recommendations**		
Project	**Issue Commenters**	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Nutch	Top-1	0.41	0.78	0.93	0.43	0.77	0.94
	Top-3	-	0.59	0.81	-	0.61	0.81
	Top-5	-	-	0.57	-	-	0.58
OpenNLP	Top-1	0.25	0.66	0.89	0.47	0.79	0.96
	Top-3	-	0.56	0.67	-	0.65	0.78
	Top-5	-	-	0.62	-	-	0.70
Curator	Top-1	0.15	0.55	0.82	0.27	0.68	0.89
	Top-3	-	0.41	0.69	-	0.50	0.75
	Top-5	-	-	0.51	-	-	0.54

The expert developer recommendations for methods in the OpenNLP project

had the highest accuracy for Top-1-Top-5 with 96% accuracy for unweighted recommendations. For weighted recommendations, the highest accuracy was again for Top-1-Top-5 accuracy with 89%. The lowest accuracy was for Top-1-Top-1 for both weighted and unweighted recommendations with 25% and 47% respectively. The method accuracy for the remaining ranged from 56% to 67% for weighted recommendations and from 65% to 79% for unweighted recommendations. As can be seen, for the OpenNLP project, the unweighted recommendations performed better than the weighted recommendations.

For the Curator project, the highest accuracy is 89% for Top-1-Top-5 with respect to unweighted recommendations. For the corresponding weighted recommendation, we had 82% accuracy. The lowest accuracies are 15% for weighted recommendations and 27% for unweighted recommendations for Top-1-Top-1. For weighted recommendations, the remaining accuracies ranged from 41% to 69%, and it ranged from 54% to 75% for unweighted recommendations. For both only 1 out of 4 exceeds 80%. But for weighted recommendations, 1 out of 4 exceeds 70% while for unweighted recommendations 2 out of 4 exceeds 70%. Even though the difference is small, the unweighted recommendations performed better for the Curator.

The highest accuracy we reached for methods amongst all software projects is 96% with OpenNLP and the lowest accuracy is 15% with Curator. As was the case for the previous artifact types, the former is for Top-1-Top-5 and the latter is for Top-1-Top-1. Out of the 36 accuracies we calculated, 8 of them exceed 80% and 21 of them exceed 60%. Only 7 out of 36 are under 50% - with 3 of them being from the Curator project. As was the case for folders and files, the unweighted recommendations had better accuracy in comparison to weighted recommendations.

For folders and files, the projects with the largest number of nodes and relationships had the highest accuracy and the smallest project had the lowest accuracy. For methods, however, even though the smallest project had the lowest accuracy, the largest project was not the one with the highest accuracy. We believe it can still be concluded that the history of a software project can be utilized to find

experts for a software method, and, with more historical data, higher accuracies can be reached.

On average, the highest accuracy reached for methods is 93% for unweighted recommendations. This accuracy was calculated for Top-1-Top-5 accuracy. For weighted recommendations, we achieved 88% accuracy again for Top-1-Top-5. The lowest average accuracy is 27% for Top-1-Top-1 accuracy for weighted recommendations. For unweighted recommendations, the lowest accuracy is 39%, again for Top-1-Top-1 accuracy. Similar to folders and files, the unweighted recommendations had better accuracy than weighted recommendations on average. Since 7 out of 12 accuracies exceed 60% and we reached 93% Top-1-Top-5 accuracy, we believe it can be said that our algorithm recommends accurate experts for methods.

Chapter 7

Discussion

In this section, we first compare our work to the related work, previously discussed in Section 2.6. Then, we share the implications of this work for researchers and practitioners.

7.1 Comparison with Related Work

In this section, we share an overview of the papers, whose main purpose is to recommend developers for mentorship, in Table 7.1 and compare our study to them. We share the type of data they utilize and the sources they use to retrieve it, the kind of algorithm they use for recommendations, in addition to the granularity of that recommendation, and whether they provide tool support. Our study is the first entry in this table for comparison.

As seen in Table 7.1, all of the related work utilize commits in their work, as do we. Akter [23] is the only work, amongst the four, that does not use other data types. Kintab et al. [22] utilize information gained from an external tool, SimCad, and their framework. The usage of external tool is similar to our work as we use CodeShovel to gather method change histories.

Table 7.1: Related work for mentor recommendation

Paper	Data Type	Data Source	Algorithm Type	Granularity	Tool Support
Finding Expert Developers Using Artifact Traceability Graphs (our study)	Commits, Code Reviews, Method Change History	GitHub, and CodeShovel	Graph Analysis	Folders, Files, Methods	Yes
YODA: Young and newcOmer Developer Assistant [21]	Emails and Commits	Mailing List Archives and Version Control System	Heuristics and IR(information retrieval) based	Files and Open-Ended Questions	Yes (plugin)
Recommending Software Experts Using Code Similarity and Social Heuristics [22]	Commits, SimCad Tool Outputs which Include Clones of Code Fragments, Information kept on the ERSF which is Based on the Actions Developers Take when Given Recommendations	Git, SimCad and the ERSF itself	Heuristic-based	Code Fragments	No, but a Framework (ERSF) is presented
Recommending expert developers using usage and implementation expertise [23]	Commits	Version History System and Source Code Repository	Metric-based	Methods	No

Amongst the related work we have presented, ours is the only one that uses graph analysis in its algorithm. The most common algorithm type is Heuristic-based with both Canfora et al. [21] and Kintab et al. [22] using it.

Canfora et al. [21] support recommendations for files and topics (asked via open-ended questions). We believe the support of open-ended questions offers a level of flexibility to the granularity where developers are not restricted to a set of options. Kintab et al. [22] support recommendations for code fragments and it can be argued that this is the most granular recommendation among the related work. Akter [23] supports recommendations for methods, which is again a recommendation with higher granularity. In our work, we support recommendations for folders, files and methods. We believe this to be a strong aspect of our work, as it offers the chance to find mentors with a wide range of granularity.

Only 1 out of 3 of the related work seen in Table 7.1 (excluding our work) provide tool support. This work is by Canfora et al. [21] who present an Eclipse plug-in called YODA. We believe the difference in our works, in terms of tool support, lies in the use cases supported by the tool itself. We believe that our expansion of the SAA tool provides a more intact and broad user experience as it includes data retrieval and visualization together with mentor recommendation while YODA allows for a more on-the-go user experience where the plug-in can be quickly accessed while the developer works.

Considering the wide range of granularity of our recommendation algorithm and our tool support, we believe our work to be a novel addition to the works for mentor recommendation.

In terms of accuracy, we reached up to 88% accuracy for folders, 86% accuracy for files, and 93% accuracy for methods. All of these were found for Top-1-Top-5 accuracy. Akter [23] also utilize Top-k accuracy during their evaluation. Specifically, they use Top-1, Top-5, and Top-10 accuracy and their method accuracy ranged from 65.3% to 97%. After investigating the individual software projects tested, we found that only 12 developers contributed to one of the projects up until 2022. This open-source project reached 100% accuracy for the initial commits

when testing Top-10 and may have immensely skewed the results of the general accuracy, always scoring substantially higher than every other software project tested in every other category. Henceforth, we believe it is important to mention that testing with projects with a smaller developer base may skew the accuracy of results.

7.2 Implications for Researchers

In this section, we explore how other researchers can utilize and/or expand our work.

7.2.1 Inclusion of More Languages

The Expert Developer Finder algorithm was implemented for and tested with projects whose main coding language is Java. We focused on Java projects as CodeShovel was evaluated with Java projects, which is one of the underlying tools we use. CodeShovel is also able to extract method change histories for Python projects, which provides an opportunity to advance and evaluate our algorithm with Python projects. In addition, CodeShovel can also be expanded to support other languages [50] which would also expand the languages supported by our algorithm.

7.2.2 Exploring New Artifacts

Our algorithm currently focuses on a limited amount of artifact types for recommendations, and it can be expanded to take into account the existing artifacts in the graph or cover more areas of software, such as non-code files like build files.

As we utilize issues during our evaluation, we excluded the usage of issues during the recommendation process. With another validation method in place,

our algorithm can be expanded to take into account issues during the recommendations process.

In addition, different types of data can be obtained from different data sources which can be used to create new artifacts in the graph. For example, non-source code files can also be incorporated into the graph, or other forms of communication, such as emails, could be analyzed and incorporated. By doing so, the algorithm can look into more cases of relationships between artifacts and developers and output a more sophisticated result.

7.2.3 Exploring Relationship Weights

The artifact traceability graph constructed for our algorithm has relationships weighted according to recency and a weight coefficient. While the recency is relevant for all relationships, the weight coefficient is only relevant for two relationships (COMMITTED and REVIEWED). Other researchers can expand this work to investigate the values we selected for the coefficient and apply the coefficient to other relationships.

7.2.4 Investigation of Efficiency

During our validation, we focused on the accuracy of our algorithm and not particularly on its efficiency. Other researchers can utilize the detailed explanation of our methodology to investigate and, if necessary improve, the efficiency of our algorithm.

7.2.5 Other Validation Methods

This work can be expanded by applying other validation methods to the Expert Developer Finder algorithm. For example, the developers who worked on the

OSS projects we used for validation can be contacted to ask for their opinion on the recommendations given by our algorithm. Another validation method would be to test our algorithm in a company setting as this is another possible usage of the algorithm.

7.3 Implications for Practitioners

In this section, we explore how practitioners from the industry can utilize and/or expand our work.

7.3.1 Tool Support

In our study, we have integrated our algorithm into SAA, which is a tool where developers can visualize the artifacts in their software project and access several types of different algorithms that use the application's internal artifact traceability graph. Practitioners can utilize this tool to gain insight into their project and use our algorithm to find experts for a specific folder, file or method when needed.

7.3.2 IDE Support

Additionally, the algorithm can be integrated as an IDE plugin for developers to use with ease. We believe that developers would find our algorithm more convenient to use if it was integrated into their IDE as an extension. Moreover, using such tools may immensely increase efficiency in the workplace as they are already acquainted with the IDEs they use.

7.3.3 GitHub Extension

Just like how the GitHub interface presents the latest modifiers to a file and this is utilized by developers to find the expert developer, a GitHub extension of our algorithm can be created to allow developers to access our algorithm's recommendations from the GitHub website interface.

7.3.4 Assist With the Onboarding Process

Finding mentors is crucial for the onboarding process of new developers. The mentor finding process must be tailored to each newcomer by focusing on the experts of the newcomer's work. Currently, our algorithm outputs a list of developers for a single artifact and this can be utilized to find the correct mentor for the newcomer's work. In addition, the algorithm can be expanded to select multiple artifacts to suggest expert developers instead of the single artifact approach we have implemented.

7.3.5 Contacting the Expert

Developers may specifically struggle to contact the expert developers, even if they find them using our algorithm. Our study can be expanded to include an interface that allows developers to directly contact the recommended developers.

7.3.6 Progress Tracking

Developers can use our algorithm to gain insight into their own knowledge of a certain artifact. In addition, by calculating the expertise of each developer for all artifacts, a ranking of the top developers in the projects can be found. This information would be useful to team leads when checking the expertise level and progress of different developers.

Chapter 8

Threats to Validity

In this section, we address the threats to the validity of our study. We follow the guidelines specified in [59] and consider construct, internal and external validity, and reliability separately.

Construct Validity is concerned about the suitability of an approach or methodology or operational measures to the research questions being investigated [59]. One threat to the construct validity is related to how we quantify expertise when suggesting developers. We rely on direct changes and/or reviews made to an artifact during the calculation. This means that any second-hand knowledge of a specific artifact, because they worked on the surrounding artifacts, for example, is not considered. Additionally, the quantity of the changes/reviews made by a developer does not affect the expertise that is gained by the change in question. Meaning, the updates made on the artifact traceability graph are the same if the change in question changes 1 or 100 line(s). We did consider weighing changes/reviews differently according to the amount, however, we remain unconvinced that the quantity of the change/reviews is directly correlated with its difficulty or the expertise gained/indicated. For instance, a single line of code can be more vital to the structure of the code compared to lines of unstructured meaningless code. However, future improvements can be considered to calculate the importance of the changes/reviews based on the code syntax.

Some concern might also be raised over our decision to exclude the deleted methods from the artifact traceability graph and therefore the expertise calculation. We tried to address this concern by retrieving method changes for each commit in the repository. This would mean that we would make three calls (retrieve files, retrieve methods, and retrieve method changes) for each commit so that any method that existed at any point is included in the graph. However, the time complexity of this process increases significantly as the commit count increases. Therefore, we decided not to go with this approach. We believe that excluding deleted methods does not cause an issue in accomplishing the goal of our algorithm, since the use case we are interested in is a novice developer seeking expertise on a method they are working on. This request can not be made for deleted methods.

Another threat to the construct validity is related to how we evaluate the accuracy of our algorithm. To test the accuracy, we run the validation algorithm for all methods, files, and folders present in each software project. We base our validation on comments made to issues on Jira, as we have not utilized it to calculate the expertise, and believe that contribution to them is a reasonable signal of expertise. However, there is still a possibility that someone who is an expert on an artifact does not have an equivalent amount of comments on issues or someone who has very little experience has a lot of comments on the issue. To mitigate the latter scenario, we require the commenters we use during validation to at least have one COMMITTED relationship in the graph. Nevertheless, it should be noted that our accuracy results are computed from another type of data we have collected. Another approach would be to contact the developers from the tested software project and ask them to evaluate our rankings.

Internal Validity is concerned about the existence and effect of unaddressed factors on the study/experiment [59]. In our study, the main threat to internal validity is related to the type of data we use. The historical data we rely on to construct the artifact traceability graph, uses version history data, such as commits, and issue tracking data, such as issues or bugs. Later, we only use version history data to recommend experts. As a result, expertise gained via issues and

other types of deliverables in a software project, such as reports, technical analysis, etc., does not affect our recommendations. It is also important to note that the intentions behind a change are unknown, and we can only rely on the change itself and not the purpose/aim behind it to define expertise.

Another threat to mention is that we completely rely on the version history to provide accurate information about the developer's contribution. There might be some loss of expertise if the developer's contribution is missing from the version history - due to squash merges or other unexpected issues. However, we consider these to be issues not unique to or caused by our study.

Another threat to mention is related to the retrieval of method change history. While testing our tool, we realized that a refactor of the method, which included body, name, and parameter change, was missed by CodeShovel. This raises the concern of not having the complete method change history when finding experts for it. However, we believe that the success rate of CodeShovel, 90% of the methods' complete history was retrieved and 97% of method changes were found as given in [50], is acceptable. Therefore, we consider the history retrieved via the CodeShovel tool to be accurate.

One of the internal threats is related to the weights of the graph. We decided on the weights of the graph intuitively rather than testing out different weight coefficient values. We believe this study can be improved by testing it with various weights to find the optimal weights for the relationships between nodes.

External Validity is concerned about the generalizability of a study and its results [59]. One of the main external threats of this study is the limited number of open-source projects being tested. We test our algorithm on three open-source projects and this may not be enough to generalize the results of the study. We mitigate this threat by picking projects that are complex enough to represent large-scale software projects. However, the study can still be expanded by testing with other projects.

In addition, our study can only be generalized to software projects that use Java as their main language.

Reliability is concerned about how much the study depends on the researchers who have conducted it and whether it can be repeated [59]. In our study, the main possible threat to reliability is the updates we make to the artifact traceability graph for each change case. These decisions rely on the authors' collective knowledge about software development. While other authors may take a different approach, we believe our detailed explanation of our approach mitigates this possible threat.

Chapter 9

Conclusion and Future Work

In this study, we developed an algorithm that recommends experts for a specific code segment (folder, file, method) by utilizing an artifact traceability graph. Our work details the steps for data retrieval, the construction of the graph, and the recommendation. All of these steps are integrated into a tool, SAA, first developed in [37].

Our aim with this work is for the Expert Developer Finder algorithm to aid newcomers during the onboarding process by recommending mentors for a specific code segment or aid any developer as they work on a code segment they are unfamiliar with.

For validation, we compare our recommendations for folder, file and methods in three open-source projects (Nutch, OpenNLP, Curator) to top issue commenters of artifacts in question. On average for folders, we reached 88% Top-1-Top-5 accuracy for weighted recommendations and 88% Top-1-Top-5 accuracy for unweighted recommendations. On average for files, we reached 86% Top-1-Top-5 accuracy for weighted recommendations and 85% Top-1-Top-5 accuracy for unweighted recommendations. On average for methods, we reached 91% Top-1-Top-5 accuracy for weighted recommendations and 95% Top-1-Top-5 accuracy for unweighted recommendations. We believe this shows that our algorithm is

able to recommend experts for code segments by utilizing the historical data of a software project. In addition, the project with more nodes and relationships had higher accuracy for both folders and files and the smallest project had the lowest accuracy for all artifact types. Henceforth, it can be argued that more historical data leads to better recommendations.

Our initial expectation was to reach better accuracy with weighted recommendations. Since this was not achieved in this work, we plan to expand it by experimenting with different weights. In addition, we are planning to contact the developers from the open-source projects we tested with to gather their opinions and further validate our algorithm.

Bibliography

[1] CIPD, "Resourcing and talent planning survey 2020," *Chartered Institute of Personnel and Development*, 2020.

[2] F. Fagerholm, A. S. Guinea, J. Münch, and J. Borenstein, "The role of mentoring and project characteristics for onboarding in open source software projects," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10, 2014.

[3] R. Britto, D. Smite, L.-O. Damm, and J. Börstler, "Evaluating and strategizing the onboarding of software developers in large-scale globally distributed projects," *Journal of Systems and Software*, vol. 169, p. 110699, 2020.

[4] R. Britto, D. S. Cruzes, D. Smite, and A. Sablis, "Onboarding software developers and teams in three globally distributed legacy projects: A multi-case study," *Journal of Software: Evolution and Process*, vol. 30, no. 4, p. e1921, 2018.

[5] P. Rodeghero, T. Zimmermann, B. Houck, and D. Ford, "Please turn your cameras on: Remote onboarding of software developers during a pandemic," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 41–50, IEEE, 2021.

[6] E. Dwoskin, "Americans might never go back to the office, and twitter is leading the charge," Oct 2020.

[7] D. G. Tsipursky, "Elon musk is now a fan of remote work," Jan 2023.

[8] J. Horwitz, "Facebook to shift permanently toward more remote work after coronavirus," May 2020.

[9] H. Field and J. Vanian, "Meta employees are back in the office three days a week as part of new mandate," Sep 2023.

[10] Git, "Git - git-blame." Version 2.42.0.

[11] D. M. German, B. Adams, and K. Stewart, "cregit: Token-level blame information in git version control repositories," *Empirical Software Engineering*, vol. 24, pp. 2725–2763, 2019.

[12] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *2011 18th Asia-Pacific Software Engineering Conference*, pp. 389–396, IEEE, 2011.

[13] T. Zhang and B. Lee, "An automated bug triage approach: A concept profile and social network based developer recommendation," in *Intelligent Computing Technology: 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings 8*, pp. 505–512, Springer, 2012.

[14] N. Li, W. Mo, and B. Shen, "Task recommendation with developer social network in software crowdsourcing," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pp. 9–16, IEEE, 2016.

[15] A. Yadav, S. K. Singh, and J. S. Suri, "Ranking of software developers based on expertise score for bug triaging," *Information and Software Technology*, vol. 112, pp. 1–17, 2019.

[16] X. Zhang, T. Wang, G. Yin, C. Yang, Y. Yu, and H. Wang, "Devrec: a developer recommendation system for open source repositories," in *Mastering Scale and Complexity in Software Reuse: 16th International Conference on Software Reuse, ICSR 2017, Salvador, Brazil, May 29-31, 2017, Proceedings 16*, pp. 3–11, Springer, 2017.

[17] H. A. Çetin, E. Doğan, and E. Tüzün, "A review of code reviewer recommendation studies: Challenges and future directions," *Science of Computer Programming*, vol. 208, p. 102652, 2021.

[18] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 141–150, IEEE, 2015.

[19] M. Fejzer, P. Przymus, and K. Stencel, "Profile based recommendation of code reviewers," *Journal of Intelligent Information Systems*, vol. 50, pp. 597–619, 2018.

[20] E. Sülün, E. Tüzün, and U. Doğrusöz, "Rstrace+: Reviewer suggestion using software artifact traceability graphs," *Information and Software Technology*, vol. 130, p. 106455, 2021.

[21] G. Canfora, M. Di Penta, S. Giannantonio, R. Oliveto, and S. Panichella, "Yoda: Young and newcomer developer assistant," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 1331–1334, IEEE, 2013.

[22] G. A. Kintab, C. K. Roy, and G. I. McCalla, "Recommending software experts using code similarity and social heuristics.," in *CASCON*, pp. 4–18, 2014.

[23] S. Akter, "Recommending expert developers using usage and implementation expertise," Master's thesis, University of Lethbridge, Lethbridge, Alberta, Canada, 2021.

[24] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, (New York, NY, USA), p. 121–124, Association for Computing Machinery, 2008.

[25] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, "Expert recommendation with usage expertise," in *2009 IEEE international conference on software maintenance*, pp. 535–538, IEEE, 2009.

[26] E. Constantinou and G. M. Kapitsaki, "Identifying developers' expertise in social coding platforms," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 63–67, IEEE, 2016.

[27] S. Baltes and S. Diehl, "Towards a theory of software development expertise," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 187–200, 2018.

[28] W. Huang, W. Mo, B. Shen, Y. Yang, and N. Li, "Cpdscorer: Modeling and evaluating developer programming ability across software communities.," in *SEKE*, pp. 87–92, 2016.

[29] J. Yan, H. Sun, X. Wang, X. Liu, and X. Song, "Profiling developer expertise across software communities with heterogeneous information network analysis," in *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, pp. 1–9, 2018.

[30] S. L. Vadlamani and O. Baysal, "Studying software developer expertise and contributions in stack overflow and github," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 312–323, IEEE, 2020.

[31] F. Javeed, A. Siddique, A. Munir, B. Shehzad, and M. I. Lali, "Discovering software developer's coding expertise through deep learning," *IET Software*, vol. 14, no. 3, pp. 213–220, 2020.

[32] H. A. Çetin and E. Tüzün, "Analyzing developer contributions using artifact traceability graphs," *Empirical Software Engineering*, vol. 27, no. 3, p. 77, 2022.

[33] Y. Sun, Z. Xu, C. Liu, Y. Zhang, and Y. Liu, "Who is the real hero? measuring developer contribution via multi-dimensional data integration," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 825–836, IEEE, 2023.

[34] A. Mockus, P. C. Rigby, R. Abreu, P. Suresh, Y. Chen, and N. Nagappan, "Modeling the centrality of developer output with software supply chains," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, (New York, NY, USA), p. 1809–1819, Association for Computing Machinery, 2023.

[35] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?," in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, pp. 1–11, 2012.

[36] "Software artifact analyzer."

[37] L. Merdol, E. Tüzün, and U. Doğrusöz, "Software artifact analyzer," *Journal of Systems and Software*, In Review.

[38] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *2012 34th International conference on software engineering (ICSE)*, pp. 419–429, IEEE, 2012.

[39] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, and E. G. Im, "Software plagiarism detection: a graph-based approach," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pp. 1577–1580, 2013.

[40] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 872–881, 2006.

[41] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 13–23, 2008.

[42] D. Hu, J. L. Zhao, and J. Cheng, "Reputation management in an open source developer social network: An empirical study on determinants of positive evaluations," *Decision Support Systems*, vol. 53, no. 3, pp. 526–533, 2012.

[43] M. A. Aljemabi and Z. Wang, "Empirical study on the evolution of developer social networks," *IEEE Access*, vol. 6, pp. 51049–51060, 2018.

[44] T. Hou, X. Yao, and D. Gong, "Community detection in software ecosystem by comprehensively evaluating developer cooperation intensity," *Information and Software Technology*, vol. 130, p. 106451, 2021.

[45] E. Sülün, E. Tüzün, and U. Doğrusöz, "Reviewer recommendation using software artifact traceability graphs," in *Proceedings of the fifteenth international conference on predictive models and data analytics in software engineering*, pp. 66–75, 2019.

[46] J. Lipcak and B. Rossi, "A large-scale study on source code reviewer recommendation," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 378–387, IEEE, 2018.

[47] G. Gousios, "The ghtorent dataset and tool suite," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 233–236, IEEE, 2013.

[48] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: Cocomo 2.0," *Annals of software engineering*, vol. 1, pp. 57–94, 1995.

[49] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 931–940, IEEE, 2013.

[50] F. Grund, S. A. Chowdhury, N. Bradley, B. Hall, and R. Holmes, "CodeShovel: Constructing method-level source code histories," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 1510–1522, IEEE, 2021.

[51] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 1–4, ACM, 2018.

[52] "Neo4j graph database analytics – the leader in graph databases."

[53] A. Library, "Awesome procedures for neo4j 5.21.0.x (extended)," 2024.

[54] U. Dogrusoz, A. Cetintas, E. Demir, and O. Babur, "Algorithms for effective querying of compound graph-based pathway databases," *BMC Bioinformatics*, vol. 10, 11 2009.

[55] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking java methods with git mechanisms," *Journal of Systems and Software*, vol. 165, p. 110571, 2020.

[56] T. A. S. Foundation, "Nutch."

[57] T. A. S. Foundation, "Welcome to apache opennlp."

[58] T. A. S. Foundation, "Welcome to apache curator."

[59] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, pp. 131–164, 2009.

Appendix A

Validation - Finding Related Issues for Files

Algorithm 6 Get Issues Related to a File from Artifact Traceability Graph

MATCH (i:Issue) - [r1:REFERENCED_COMMIT] - (c:Commit) - [r2:CONTAINS] - (m:Method) - [r3:IN] - (f:File)

WHERE elementId(f) = $file_id AND m.name <>"dummy"

RETURN elementId(i)

UNION

MATCH (i:Issue) - [r1:REFERENCED_COMMIT] - (c:Commit) - [r2:CONTAINS] - (m:Method) - [r3:IN] - (f:File) - [r4:RENAMED_TO] - (f2:File)

WHERE elementId(f2) = $file_id AND m.name <>"dummy"

RETURN elementId(i)

UNION

Get Issues Related to a File from Artifact Traceability Graph

MATCH (i:Issue) - [r1:REFERENCED_PR] - (pr:PullRequest) - [r2:INCLUDES] - (c:Commit) - [r3:CONTAINS] - (m:Method) - [r4:IN] - (f:File)

WHERE elementId(f) = $file_id AND m.name <>"dummy"

RETURN elementId(i)

UNION

MATCH (i:Issue) - [r1:REFERENCED_PR] - (pr:PullRequest) - [r2:INCLUDES] - (c:Commit) - [r3:CONTAINS] - (m:Method) - [r4:IN] - (f:File) - [r5:RENAMED_TO] - (f2:File)

WHERE elementId(f2) = $file_id AND m.name <>"dummy"

RETURN elementId(i)

UNION

MATCH (i:Issue) - [r] - (i2:Issue) - [r1:REFERENCED_COMMIT] - (c:Commit) - [r2:CONTAINS] - (m:Method) - [r3:IN] - (f:File)

WHERE elementId(f) = $file_id AND m.name <>"dummy"

RETURN elementId(i)

UNION

MATCH (i:Issue) - [r] - (i2:Issue) - [r1:REFERENCED_COMMIT] - (c:Commit) - [r2:CONTAINS] - (m:Method) - [r3:IN] - (f:File) - [r4:RENAMED_TO] - (f2:File)

WHERE elementId(f2) = $file_id AND m.name <>"dummy"

RETURN elementId(i)

UNION

MATCH (i:Issue) - [r] - (i2:Issue) - [r1:REFERENCED_PR] - (pr:PullRequest) - [r2:INCLUDES] - (c:Commit) - [r3:CONTAINS] - (m:Method) - [r4:IN] - (f:File)

WHERE elementId(f) = $file_id AND m.name <>"dummy"

RETURN elementId(i)

UNION

MATCH (i:Issue) - [r] - (i2:Issue) - [r1:REFERENCED_PR] - (pr:PullRequest) - [r2:INCLUDES] - (c:Commit) - [r3:CONTAINS] - (m:Method) - [r4:IN] - (f:File) - [r5:RENAMED_TO] - (f2:File)

WHERE elementId(f2) = $file_id AND m.name <>"dummy"

RETURN elementId(i)