Recommending Participants for Collaborative Merge Sessions

Catarina Costa, Jair Figueirêdo, João Felipe Pimentel, Anita Sarma, and Leonardo Murta

Abstract—Development of large projects often involves parallel work performed in multiple branches. Eventually, these branches need to be reintegrated through a merge operation. During merge, conflicts may arise and developers need to communicate to reach consensus about the desired resolution. For this reason, including the right developers to a collaborative merge session is fundamental. However, this task can be difficult especially when many different developers have made significant changes on each branch over a large number of files. In this paper, we present TIPMerge, an approach designed to recommend participants for collaborative merge sessions. TIPMerge analyzes the project history and builds a ranked list of developers who are the most appropriate to integrate a pair of branches (Developer Ranking) by considering developers' changes in the branches, in the previous history, and in the dependencies among files across branches. Simply selecting the top developers in such a ranking is easy, but is not effective for collaborative merge sessions as the top developers may have overlapping knowledge. To support collaborative merge, TIPMerge employs optimization techniques to recommend developers with complementary knowledge (Team Recommendation) aiming to maximize joint knowledge coverage. Our results show a mean normalized improvement of 49.5% (median 50.4%) for the joint knowledge coverage with the optimization techniques for assembling teams of three developers for collaborative merge in comparison to choosing the top-3 developers in the ranked list.

Index Terms— Version Control, Branch Merging, Collaborative Merge, Developer Recommendation, Optimization

1 Introduction

The software development process often requires parallel work made by multiple people because of a host of reasons, such as time-to-market, maintenance of old versions, and implementation of a subsystem. Such parallel work is usually isolated from each other by using branches. A branch provides an isolated area where changes can be made, designs explored, and code tested in parallel with other teams working in other branches [1].

Changes made in branches need to be reintegrated periodically through a merge operation. The effort involved in such an integration is usually dependent on how much work went on in the branch and also in the other branches in the intervening time [2], especially in situations when conflicts occur [3].

In the case of conflicts, developers often need to discuss their choice of conflict resolution with other developers since conflicts also suggest differences in opinions and strategies [4]. In such situations, some developers may not have enough knowledge to make the right decision. Therefore, the collective knowledge of project members can help such conflict resolutions, with all involved parties participating in a collaborative effort [4], [5]. For instance, a survey of 164 developers [6] showed that when performing a merge, people frequently (75%) contacted other developers to resolve the conflicts together.

Inviting the right set of developers to a collaborative merge session is a key requirement for dealing with this problem. However, this requires knowledge about the project to prioritize among developers [3]: who are aware of the project history, the dependencies in the project, and the changes in the branches. Inviting all involved developers to a merge session is infeasible due to cost and developer availability. Moreover, some developers may have similar knowledge, as they may have changed the same files. In this case, recommending such developers could lead to an overlap in knowledge. Therefore, a global view of knowledge distribution among developers is fundamental.

Existing approaches do not address how to select developers who are the best suited to perform a collaborative merge. Some proposals focus on supporting collaborative model merging [4], [7] and collaborative real-time editor [5], [8]. These studies aim at enabling all involved participants to work in a collaborative fashion. We also found works that assign developers to software activities, mainly assigning developers to issues [9]–[16] or pull request [17]–[23]. However, assigning developers to a collaborative merge session brings additional challenges due to the number of developers, the time interval between branch synchronizations, and the number of commits per branch. Finally, dependencies among files across branches add further complexity to the merge process.

In this paper, we propose TIPMerge, an approach designed to recommend developers for collaborative merge sessions. TIPMerge builds a ranked list of developers who are the most appropriate to integrate a pair of branches (the Developer Ranking). As the top developers in the ranking may have similar knowledge, this ranked list is then used to detect knowledge coverage of the artifacts. TIPMerge checks the developer's knowledge coverage of changed files and methods and shows which set of developers have the highest joint coverage (the Team Recommendation). As the number of developers in the ranking can be high, we use an optimization algorithm to find which developers make the best team to deal with a specific merge case.

In previous a paper [3], we presented the first version of TIPMerge, which builds a ranked list of suitable candidates

- *Catarina Costa, João Felipe Pimentel, and Leonardo Murta are with the Computing Institute, Fluminense Federal University, Niterói, RJ, Brazil. E-mail: (ccosta, jpimentel, leomurta}@ic.uff.br.*
- *Jair Figueirêdo and Catarina Costa are with the Federal University of Acre, Rio Branco, AC, Brazil. E-mail: {jjcfigueiredo, catarina}@ufac.br.*
- *Anita Sarma is with School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA. E-mail: anita.sarma@oregonstate.edu.*

xxxx-xxxx/0x/$xx.00 © 200x IEEE Published by the IEEE Computer Society

to perform the merge based on a medal count system. Although useful for conventional merge, in which just one developer is selected for the task, picking the top developers from a ranked list is not appropriate for collaborative merges. This work complements our previous work by introducing the concept of joint knowledge coverage and providing an optimization strategy to select developers that form the most suitable team for the merge. As shown by our experiments, optimizing the joint knowledge coverage when assembling teams of three developers for collaborative merge leads to a mean normalized improvement of 49.5% in the knowledge coverage when compared to choosing the top-3 developers in the ranked list.

In summary, our main contributions are:

Approach - We present an approach that analyzes change history in branches, file dependencies, and the past history to build a ranked list of developers who are the most appropriate to integrate a pair of branches. Using this ranked list, our approach is able to recommend the most suitable developers in terms of *joint knowledge coverage* to team up in a collaborative merge session.

Implementation - We implemented our approach in a tool that is able to build a ranked list of suitable candidates to perform the merge based on a medal count system. It also recommends the maximum knowledge coverage for a collaborative merge session. Given the number *n* of developers provided by the user, *TIPMerge recommends the n developers that maximize the joint knowledge coverage*.

Empirical Evaluation - We quantitatively evaluated TIPMerge over 25 real-world projects. We verified the normalized improvement in joint knowledge coverage of the TIPMerge team recommendation for collaborative merge over the selection of the top developers from the ranked list. In summary, our evaluation was designed to answer the following research question: *Does the n-developer TIPMerge* ***Team Recommendation*** *for collaborative merge improve the percentage of joint knowledge coverage over teaming up the top-n developers from the* ***Developer Ranking****?*

2 TIPMERGE

The primary goal of TIPMerge is to recommend teams for a collaborative merge session with the expertise to integrate changes across two branches. Our approach has the following steps, shown in Fig. 1: (1) It extracts data from the repository until the branch tips – the two most recent commits of the two branches that will be merged (see Section 2.2); (2) It detects dependencies among files by identifying files that were frequently committed together. We calculate dependencies from the data before the branch creation (see Section 2.3); (3) It Identifies developers who edited key files – files that were edited in both branches or had dependencies across branches (see Section 2.4); (4) It builds a ranked list of suitable candidates to perform the merge based on a medal count system (see Section 2.5); (5) It recommends a team of developers that have together the maximum knowledge coverage over the key files for a collaborative merge session, considering the ranking and contributions of each developer (see Section 2.6).

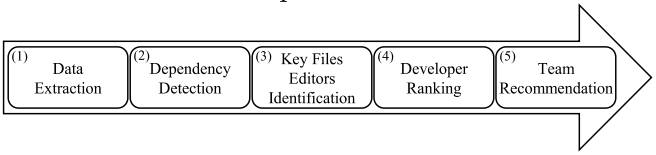


Fig. 1. Steps of the approach

2.1 Scenario

Here is an intentionally simple scenario that we use as a running example to explain how TIPMerge works. Consider a hypothetical project, Calculator, which employs a feature branch in parallel to the master branch to implement advanced operations. Fig. 2 presents a commit history that includes these two branches, and five developers: Alice, Alex, Bob, Peter, and Tom. Let us assume that Bob creates a feature branch from the master (C50) and performs three commits (C51, C54, and C56). Alex and Tom also committed to this branch (C57 and C59, respectively). Alice and Peter continue to work in the master branch in parallel. Alice performs two commits (C52 and C53), followed by two commits from Peter (C55 and C58). Let us further assume that Alice and Bob change the same files, *QuadraticEquation* and *Subtraction*, across the branches (see Table 1 and Table 2). Alex also changed the file *Subtraction* in the feature branch. Peter changed the files *Multiplication* and *Division* in the master branch. Tom changed only the file *IEquation* in the feature branch. However, there is a logical dependency in this scenario: file *QuadraticEquation* depends on file *IEquation*.

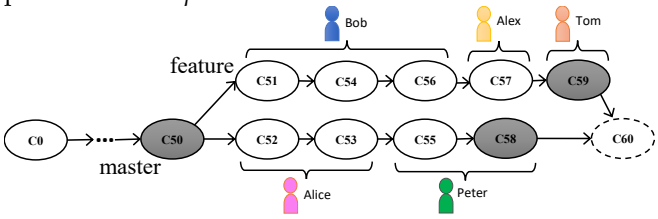


Fig. 2. Example of Merging Branches

TABLE 1
COMMITS IN THE MASTER BRANCH

File Name	Alice	Peter
QuadraticEquation	2 (C52, C53)	0
Subtraction	1 (C53)	0
Multiplication	0	2 (C55, C58)
Division	0	2 (C55, C58)

TABLE 2
COMMITS IN THE FEATURE BRANCH

File Name	Alex	Bob	Tom
QuadraticEquation	0	2 (C51, C56)	0
Subtraction	1 (C57)	3 (C51, C54, C56)	0
IEquation	0	0	1 (C59)

In our example, developers are unaware of changes made in the other branch. Therefore, Alice does not know about the parallel changes in the feature branch made by Bob to *QuadraticEquation* and *Subtraction*, and made by Alex to *Subtraction*. An automatic merge of the branches could fail due to direct conflicts. Further, Tom changed *IEquation* in the feature branch, on which *QuadraticEquation* depends, and was changed by Alice in the master branch. A merge of these branches could also generate build or test failure due to indirect conflicts.

Additionally, Table 3 shows a hypothetical change history of the project files before the branching. Alex had edited all the five files and Anna four of the five files.

We analyze information about changes across branches and those in the previous history since both are relevant. Developers who have made changes in the branches know about recent changes that need to be integrated. Developers who have modified files in the past may know about the history and goals of the implementation.

TABLE 3
CONTRIBUTIONS IN HISTORY BEFORE BRANCHING

File Name	Alex	Alice	Anna	Bob	Tom
QuadraticEquation	14	0	4	0	0
Subtraction	3	0	0	2	0
Multiplication	4	0	2	0	0
Division	1	0	3	0	0
IEquation	6	0	2	0	4

2.2 Data Extraction

The first step in our approach is extracting the data about branches from the projects. Formally, we can define a project p as a tuple (F, D, C), where F is a set of files, D is a set of developers, and C is a set of commits. Each commit $c_i \in C$ is a tuple (F_i, a_i, P_i), where $F_i \subseteq F$ is the set of files changed (add, remove, or edit) by c_i; $a_i \in D$ is the author of c_i; and $P_i \subset C$ is the set of parent commits of c_i.

Commits are organized in a directed acyclic graph (e.g., Fig. 2), where the first commit of the project has no parent (e.g., commit C0 in Fig. 2), *revision* commits have only one parent (e.g., commit C53 in Fig. 2), and *merge* commits have two or more parents (e.g., commit C60 in Fig. 2). All reachable commits from c_i form its history, including c_i itself and the transitive closure over its parents. In Fig. 2, {C0, …, C51, C54, C56, C57, C59} is the history of commit C59. The *history* of $c_i \in C$ is $H_i = \{c \in C | c = c_i \vee \exists p_j : (p_j \in P_i \wedge c \in H_j)\}$.

Two commits $c_i, c_j \in C$ that do not reach each other (i.e., $c_i \notin H_j \wedge c_j \notin H_i$) are called *variants* (e.g., commits C58 and C59 in Fig. 2). Variants may have a common history, which comprises all commits that exist in both histories. In Fig. 2, {C0, …, C50} is the common history of commits C58 and C59. The *common history* of $c_i, c_j \in C$ is $CH_{i,j} = H_i \cap H_j$.

The history of each variant also comprises commits that do not belong to the common history, forming an independent line of development called branch history. For example, {C51, C54, C56, C57, C59} is the branch history of C59 when merging with C58; and {C52, C53, C55, C58} is the branch history of C58 when merging with C59 (Fig. 2). As branches can be created from other branches, the branch history may vary depending on the opposing branch, as a consequence of different common histories. The *branch history* of $c_i \in C$ when merging with $c_j \in C$ is $BH_{i,j} = H_i \backslash H_j$.

Each branch history comprises a set of files changed by its commits. The files changed in the branch history of $c_i \in C$ when merging with $c_j \in C$ is $F_{i,j} = \bigcup_{c_k \in BH_{i,j}} F_k$.

In addition, *files changed in the common history* (i.e., $\bigcup_{c_k \in CH_{i,j}} F_k$) are extracted to determine expertise over the key files. Currently, we collect data of all past commits, but the approach can be easily adapted to only consider changes in a given time frame (e.g., past release) to accommodate decay in expertise [24].

2.3 Dependency Detection

Next, TIPMerge identifies dependencies among files that are edited across branches. This is vital since parallel changes to dependent files can cause indirect conflicts when the branches are integrated. The majority of software projects often use a combination of different programming languages. Therefore, we use logical dependencies [25] instead of structural coupling [26] in TIPMerge.

TIPMerge uses the change history of the project (before branching) to determine dependencies between pairs of files. Of course, it is possible that these dependencies might change based on edits in the branches. However, the past history provides us a baseline of these dependencies.

We assume that all commits within the same branch have already been integrated: in our scenario, since Peter and Alice are working on the same (master) branch, we assume that Peter has integrated his changes with Alice's prior to his commits.

In order to understand how we compute the *logical dependencies across files*, let's assume that each file $f_l \in F$ has a set of dependencies $Dep_l \subset F$ that are obtained by using an *association rule* mining technique. An association rule is a pair (X, Y) of two disjoint sets $X, Y \subset F$. In the notation $X \rightarrow Y$, X is called antecedent and Y is called consequent [27]. It means that, when X occurs, Y also occurs, even if they are not structurally related [28]. However, its probabilistic interpretation is based on the evidence in the transactions [25], which is determined by two metrics [27]: (1) support, the joint probability of having both antecedent and consequent, and (2) confidence, the conditional probability of having the consequent when the antecedent is present.

The confidence value can range from 0 to 1, where 1 means that every time that the antecedent is changed, the consequent is also changed. In this case, the use of a threshold is necessary because low confidence implies low probability that changing a file causes impact in the dependent file. Therefore, the use of confidence (instead of support) allows us to define the direction in the dependencies. Development teams have the freedom to decide the threshold above which a dependency becomes relevant. Our approach parameterizes the threshold and uses the value set by the user. Here, after some empirical tests, we have chosen a confidence threshold of 0.6 to determine dependency. We wanted a considerable value, a little above the middle, thus we tested thresholds of 0.6 and 0.8. We found that 0.6 provided higher normalized improvement of the Developer Ranking accuracy in contrast with the top-k developers who performed most of the merges in the past.

In our scenario, we have dependencies between the files *QuadraticEquation* and *IEquation*. *IEquation* was changed in 12 commits. Let us assume that of these 12 commits, 8 also included changes to *QuadraticEquation* (Table 3). The confidence of the association rule (*IEquation* →*QuadraticEquation*) is $8/12 = 0.66$. Based on a threshold of 0.6, we say that *QuadraticEquation* depends on *IEquation*. As confidence is not symmetric, the confidence of the rule *QuadraticEquation* →*IEquation* can be different. In our scenario, *QuadraticEquation* was changed in 18 commits, and of these 18 commits, 8 also included changes to *IEquation*. The confidence of this rule is $8/18 = 0.44$. Therefore, *IEquation* does not depend on *QuadraticEquation*.

2.4 Key File Editors Identification

The next step in our approach is to identify the developers who have modified files that are relevant to the merging of the branches. We term these files as key files:

$$KF_{i,j} = \left\{ f_k \in F \;\middle|\; \begin{array}{c} (f_k \in F_{i,j} \cap F_{j,i}) \vee \\ (f_k \in F_{i,j} \wedge Dep_k \cap F_{j,i} \neq \emptyset) \vee \\ (f_k \in F_{j,i} \wedge Dep_k \cap F_{i,j} \neq \emptyset) \vee \\ (f_k \in Dep_l \cap F_{i,j} \wedge f_l \in F_{j,i}) \vee \\ (f_k \in Dep_l \cap F_{j,i} \wedge f_l \in F_{i,j}) \end{array} \right\}$$

Key files are files changed in parallel in both branches (e.g., *Subtraction* and *QuadraticEquation*) – captured in the first line of the equation – or files that were changed in one

branch (e.g., *IEquation*), but have a dependency with files that were changed in the other branch (e.g., *QuadraticEquation*) – captured in the second and third lines of the equation. As both the dependent and the dependency are considered key files, the fourth and fifth lines of the equation are necessary to capture the dependencies. Changes to the former class of files can cause a merge failure (direct conflicts), whereas changes to the latter class can potentially lead to build or test failures (indirect conflicts). Only key files are relevant for us, as all other files can be automatically merged safely.

Once we have identified the key files, we identify the developers who have changed these files: (1) in a branch, which signals expertise in the change, or (2) in the previous history, which signals expertise in the file.

In our scenario, the key files are *QuadraticEquation*, *Subtraction*, and *IEquation*. According to Table 1 and Table 2, Alice changed *QuadraticEquation* twice and *Subtraction* once in the master branch. Bob changed the same files in the feature branch: two and three times, respectively. Alex changed *Subtraction* once in the feature branch. Moreover, Tom changed *IEquation* once in the feature branch. According to Table 3, Alice did not change any file in previous history, Bob changed *Subtraction*, and Tom changed *IEquation*. Further, Alex changed all the key files and Anna changed two of them (*QuadraticEquation* and *IEquation*).

2.5 Developer Ranking

Next, TIPMerge uses an algorithm that counts the number and type of contribution – changes in a branch or in the previous history – to build a ranking of suitable candidates for performing the merge. We use a medal system to rank developers. This is analogous to how countries are ranked in the Olympic Games, based on medal counts. The following rules define when developers receive gold, silver, or bronze medals.

A *gold medal* is awarded when a developer changes a key file in a branch. The rationale is that developers who changed a key file are the most knowledgeable about the change and its implications. They probably are also well versed with the file in general, and therefore, likely to be able to perform additional edits during a merge if necessary. *Gold medals* are defined as:

$$G_{i,j}(d) = \left| \bigcup_{c_k \in BH_{i,j} \wedge a_k = d} F_k \cap KF_{i,j} \right| + \left| \bigcup_{c_k \in BH_{j,i} \wedge a_k = d} F_k \cap KF_{i,j} \right|$$

A *silver medal* is awarded when a developer has changed a key file in the past. Developers who created or edited files in the past likely possess knowledge about the goals and requirements of these files, which can be helpful. *Silver medals* are defined as:

$$S_{i,j}(d) = \left| \bigcup_{c_k \in CH_{i,j} \wedge a_k = d} F_k \cap KF_{i,j} \right|$$

A *bronze medal* is awarded when a developer changes a file that depends on another file. We assume that developers who have changed a dependent file may have learned about the API of the file that they are using. Consequently, they may know the goals and expectations of such a file, which may help in determining the impact of a change. *Bronze medals* are defined as:

$$B_{i,j}(d) = \left| \bigcup_{c_k \in BH_{i,j} \wedge a_k = d} \bigcup_{f_l \in F_k} Dep_l \cap F_{j,i} \right| + \left| \bigcup_{c_k \in BH_{j,i} \wedge a_k = d} \bigcup_{f_l \in F_k} Dep_l \cap F_{i,j} \right|$$

TIPMerge assigns a medal for each edited file, irrespective of the number of commits made. In our scenario, Alice, Alex, and Bob each get one gold medal for *Subtraction*, even though Alice committed the file once in the master branch, and Bob committed it three times and Alex once in the feature branch. Similarly, Bob and Alex each get one silver medal for *Subtraction*, because of their past changes (before branching). As commits may have different granularities [29], counting commits for distinguishing the relevance of the contribution over files could lead to incorrect interpretations. Moreover, in our approach we assume that when a developer edits a file, that developer has knowledge about the entire file. While our approach can support a finer-grained expertise calculation at the method level, we leave it for future work as such fine-grained analysis would be language dependent.

Our algorithm prioritizes developers with gold medals since: (1) they are the expert on the change, and (2) they have the most recent knowledge about the changed file. In the case of a tie in the number of gold medals, we use the number of silver medals to break the tie. This is because, everything being equal, a developer who has more experience overall is likely to be more suitable in merging changes. Finally, when there is a tie in the number of silver medals, we consider bronze medals. The notion is that if two developers have an equal knowledge of the changes and an equal knowledge of the project history, a developer who has additional knowledge about another file is more suitable for the merge. This *medal ranking* is formally defined as:

$$R_{i,j} = \left(d_r \in D \middle| \begin{array}{c} G_{i,j}(d_r) + S_{i,j}(d_r) + B_{i,j}(d_r) > 0 \wedge \\ \left(\begin{array}{c} G_{i,j}(d_r) > G_{i,j}(d_{r+1}) \vee \\ \left(\begin{array}{c} G_{i,j}(d_r) = G_{i,j}(d_{r+1}) \wedge \\ \left(\begin{array}{c} S_{i,j}(d_r) > S_{i,j}(d_{r+1}) \vee \\ \left(\begin{array}{c} S_{i,j}(d_r) = S_{i,j}(d_{r+1}) \wedge \\ B_{i,j}(d_r) > B_{i,j}(d_{r+1}) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)$$

Table 4 shows that Alice and Bob changed *QuadraticEquation* in the master and feature branches, respectively – earning them gold medals. Alex and Anna also changed it in the previous history, each receiving a silver medal. Alice, Alex, and Bob changed *Subtraction* in the branches, earning them a gold medal each. Alex and Bob get silver medals for editing *Subtraction* file in the previous history. Only Tom modified file *IEquation* in the feature branch (earning a gold medal), and Tom, Alex, and Anna changed this file in the previous history (earning silver medals). Alice receives a bronze medal for *IEquation*, because she edited *QuadraticEquation*. Remember, file *IEquation* is a key file because it was changed in the feature branch and *QuadraticEquation*, changed on the master branch, depends on it. Moreover, our assumption is that to be able to understand and edit the dependent file (*QuadraticEquation*), the developer must have some knowledge about the dependency (in this case the interface *IEquation*).

TABLE 4
MEDALS (GOLD | SILVER | BRONZE)

File	Alex	Alice	Anna	Bob	Tom
QuadraticEquation	0 | 1 | 0	1 | 0 | 0	0 | 1 | 0	1 | 0 | 0	0 | 0 | 0
Subtraction	1 | 1 | 0	1 | 0 | 0	0 | 0 | 0	1 | 1 | 0	0 | 0 | 0
IEquation	0 | 1 | 0	0 | 0 | 1	0 | 1 | 0	0 | 0 | 0	1 | 1 | 0

By counting the medals and tie-breaking when necessary, TIPMerge generates a Developer Ranking. In our scenario, Bob has the same number of gold medals as Alice,

but he has a silver medal, which places him in the first position (Table 5). Here, the first four candidates (Bob, Alice, Alex, and Tom) all have gold medals. For the ones that know about equal "amounts" of recent changes performed in the branches (Bob and Alice with two gold medals each, and Alex and Tom with one gold medal each), the tiebreakers involving past changes or dependency information differentiate them.

TABLE 5
DEVELOPER RANKING

Developer	Gold Medal	Silver Medal	Bronze Medal
Bob	2	1	0
Alice	2	0	1
Alex	1	3	0
Tom	1	1	0
Anna	0	2	0

2.6 Team Recommendation for Collaborative Merge

Collaborative merge recommendation considers the ranking, the contributions of each developer, and the team size to recommend a team of developers that together have the maximum knowledge coverage over the key files to perform the merge.

TIPMerge identifies developers who have complementary expertise based on changes made in the key files in the branches and previous history. Although the ranking detailed in Section 2.5 sorts developers based on their expertise, the top developers in the ranking may have similar knowledge, as they may have changed the same files. In this case, selecting these developers could lead to an overlap in knowledge and, consequently, in an ineffective collaborative merge session.

Recommending the optimal team of developers is a combinatorics problem with the complexity of $O(2^n)$ for n developers. Since the number of developers who have made some changes to the project (and are therefore in the ranking) can be high, it is desirable to have solutions that do not require trying all the combinations to optimize the combination of developers that TIPMerge recommends. Meta-heuristics can achieve it and speedup the process at the expense of having a chance of not finding the optimal solution. In this work, we implemented a Tabu search metaheuristic algorithm [30]. However, other metaheuristics are also suitable for this problem.

The user can parametrize the algorithm with a minimum coverage threshold. The algorithm ignores all solutions with less coverage than the threshold, speeding up the process. For example, a threshold of 40% indicates that teams with coverage of less than 40% of the total absolute coverage considering all developers would be discarded. On the other hand, a threshold of 0% indicates that all possible teams would be considered during the optimization.

Given n, the number of desired developers that a user wants to invite to a collaborative merge, TIPMerge creates an initial solution with the top n developers from the developer ranking. Then, the Tabu search looks for solutions in the neighborhood that optimizes the current best solution until the execution time reaches the maximum duration or the algorithm performs a number of iterations without changes. Both stop conditions are provided by the user as configuration parameters.

The algorithm updates the current best solution as soon as it finds a new solution with a higher coverage, given by a fitness function. We keep older solutions in a temporary Tabu list to avoid recalculations. The fitness function is $f_{i,j}(S) = W_G \times G'_{i,j}(S) + W_S \times S'_{i,j}(S) + W_B \times B'_{i,j}(S)$, where

$$G'_{i,j}(S) = \left| \bigcup_{c_k \in BH_{i,j} \wedge a_k \in S} F_k \cap KF_{i,j} \right| + \left| \bigcup_{c_k \in BH_{j,i} \wedge a_k \in S} F_k \cap KF_{i,j} \right|$$

$$S'_{i,j}(S) = \left| \bigcup_{c_k \in CH_{i,j} \wedge a_k \in S} F_k \cap KF_{i,j} \right|$$

$$B'_{i,j}(S) = \left| \bigcup_{c_k \in BH_{i,j} \wedge a_k \in S} \bigcup_{f_l \in F_k} Dep_l \cap F_{j,i} \right| + \left| \bigcup_{c_k \in BH_{j,i} \wedge a_k \in S} \bigcup_{f_l \in F_k} Dep_l \cap F_{i,j} \right|$$

This value is based on the number of unique medals that the combination of developers has and the weight of these medals. That is, developers with medals obtained from the same files in the same branch contribute only once to the solution. TIPMerge also explores solutions with fewer developers with the same fitness value, as fewer developers needed for a collaborative merge is considered better.

For our running example we used $W_G = 1, W_S = 0.8, and\ W_B = 0.3$, but users can parameterize weights according to their needs.

When exploring the solution space, TIPMerge first tries to increase the size of the team to improve the knowledge coverage if the current solution has less than n developers, where n is the user-specified number of developers to attend the merge session. Then, it tries to decrease the number of developers but keeping the same coverage. Finally, it generates mutations of the solution by replacing developers in the solution with other developers.

We now describe the TIPMerge team recommendation process to select two developers ($n = 2$) for the merge session presented in Section 2.1. Since the top two developers are Bob and Alice (see Table 5), they comprise the initial solution.

Bob and Alice. In the master branch, Alice has two gold medals because she changed two key files. In the feature branch, Bob has two gold medals because he changed two key files. Considering the history, Bob has a silver medal. Finally, Alice has a bronze medal. By combining their medals (as there is no intersection among changed files) Bob and Alice have four gold medals (two in each branch), a silver medal, and a bronze medal. Thus, their fitness value is $1 \times (2 + 2) + 0.8 \times 1 + 0.3 \times 1 = 5.1$.

With this solution, TIPMerge tests to see if the number of developers should be increased. It stops since the solution already has the maximum allowed number of developers ($n = 2$ in this example). Then, it tries to reduce the number of developers, generating the following solutions:

Bob has two gold medals in the feature branch and a silver medal. His fitness value alone is $1 \times 2 + 0.8 \times 1 = 2.8$.

Alice has two gold medals in the master branch and a bronze medal. Her fitness value is $1 \times 2 + 0.3 \times 1 = 2.3$.

Both solutions are worse than **Bob and Alice** together. Thus, TIPMerge mutates the current best solution. TIPMerge follows the ranking order for the mutations. Thus, it replaces Bob with Alex, creating the following mutation:

Alex and Alice. Alice has two gold medals in the master branch and a bronze medal. Alex has a gold medal in the feature branch and three silver medals. Together, they have three gold medals, three silver medals, and a bronze medal. Thus, their fitness value is $1 \times (2 + 1) + 0.8 \times 3 + 0.3 \times 1 = 5.7$, which is higher than the current solution. Thus, this solution becomes the current best solution.

Since there is a better solution, TIPMerge stops the mutations and restarts the algorithm. Thus, it tries to reduce the number of developers, creating the following solution:

Alex has a gold medal in the feature branch and three silver medals. His fitness value is $1 \times 1 + 0.8 \times 3 = 3.4$. This value is lower than the current best solution.

Note that TIPMerge does not generate a new solution for Alice since her solution is already on the temporary Tabu list. Then, TIPMerge tries to mutate the solution, generating the following solutions:

Bob and Alex. Alex has a gold medal in the feature branch, but this medal is the same as one of the two gold medals that Bob has in the feature branch. Similarly, Bob has a silver medal, but it is the same as one of the three silver medals that Alex has. Thus, together they have 2 gold medals and 3 silver medals, and their fitness value is $1 \times 2 + 0.8 \times 3 = 4.4$. This value is lower than the current best solution.

Alex and Tom. Alex has a gold medal in the feature branch and three silver medals. Tom has a gold medal in the feature branch and a silver medal. Since Tom and Alex have the same silver medal, together they have two gold medals and three silver medals. Their fitness value is $1 \times 2 + 0.8 \times 3 = 4.4$. This value is lower than the current best solution.

Alex and Anna. Alex has a gold medal in the feature branch and three silver medals. Anna has two silver medals that Alex also has. Thus, Anna does not contribute to this solution and their solution together has the same fitness function as Alex's solution alone (3.4).

At this moment, the algorithm already tried all combinations that replace Alice. Therefore, it tries to replace Alex. The first solution is **Bob and Alice**, but it is already in the temporary Tabu list. It continues the mutations as follows:

Alice and Tom. Alice has two gold medals in the master branch and a bronze medal. Tom has a gold medal in the feature branch and a silver medal. Together, they have three gold medals, a silver medal, and a bronze medal. Their fitness value is $1 \times (2 + 1) + 0.8 \times 1 + 0.3 \times 1 = 4.1$. This value is lower than the current best solution.

Alice and Anna. Alice has two gold medals in the master branch and a bronze medal. Anna has two silver medals. Together, they have two gold medals, two silver medals, and a bronze medal. Their fitness value is $1 \times 2 + 0.8 \times 2 + 0.3 \times 1 = 3.9$. This value is lower than the current best solution.

At this point, it is not possible to perform other mutations to improve the current best solution. The algorithm recognizes this after a number of iterations without changes and stops. The ranking of candidates for a collaborative merge with a team of two developers is in Table 6. Alice and Alex, at the second and the third positions in the developer ranking (Table 5), respectively, have together the maximum knowledge coverage for this example.

TABLE 6
TEAM RECOMMENDATION (TOP-6)

Developers	Gold Medal	Silver Medal	Bronze Medal	Coverage
Alice, Alex	3	3	1	5.7 (74%)
Bob, Alice	4	1	1	5.1 (66%)
Bob, Alex	2	3	0	4.4 (57%)
Alex, Tom	2	3	0	4.4 (57%)
Alice, Tom	3	1	1	4.1 (53%)
Alice, Anna	2	2	1	3.9 (50%)

The coverage is represented as a percentage, which is calculated as follows: before the algorithm starts, the total absolute coverage value is calculated considering all developers available for the merge. This value is used to calculate the percentage of each combination. In the case of Alice and Alex, for example, 74% corresponds to the absolute value of Alice and Alex (5.7) divided by the total absolute value considering all developers (7.7).

3 IMPLEMENTATION

TIPMerge is an open-source project available at https://github.com/gems-uff/tipmerge under the MIT License. It is implemented in Java and analyzes projects versioned on Git independently of their programming language. TIPMerge is language agnostic when analyzing expertise at the file-level. For Java projects, TIPMerge can detect method level changes, which allows for future utilization of our approach. It however requires a parser that can identify method level changes.

We use an adapted version of Dominoes [24], [31] to identify logical dependencies among files across branches. Dominoes organizes data extracted from software repositories into matrices to denote relationships among software entities. For example, $[commit|file]$ denotes the files that were changed by commits in the project. These matrices are combined to depict higher-order relationships, such as logical dependencies among files: $[file|file] = [commit|file]^T \times [commit|file]$. Dominoes runs matrix transformations in GPU, speeding up the analysis process.

To get the recommendations for a collaborative merge, a user first needs to select the two branches to merge via the TIPMerge UI (Fig. 3 (a)). The user can also fine tune the analysis to specific file extensions (Fig. 3 (b)). Once TIPMerge analyzes the project information, it shows the files that each developer has edited and the editing frequency in terms of commits (Fig. 3 (c)). This information is provided for each branch, for the intersection of both branches, and for the previous history. The user can also check the logical dependencies by clicking on the *Get Dependencies* button (Fig. 3 (d)).

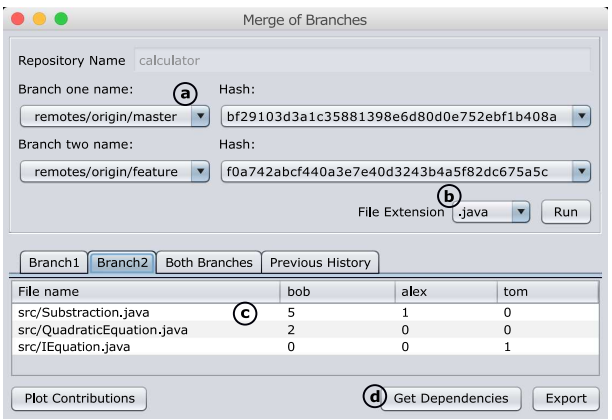


Fig. 3. Information about changed files in the branches and history

In the *Dependencies Analysis* window (Fig. 4), the user can configure the confidence threshold (Fig. 4 (a)) to determine the direction of logical dependencies among files. The user can see the information about dependencies across branches (Branch One -> Branch Two and Branch Two -> Branch One) (Fig. 4 (b)). The Developer Ranking is obtained by clicking on the *Get Ranking* button (Fig. 4 (c)).

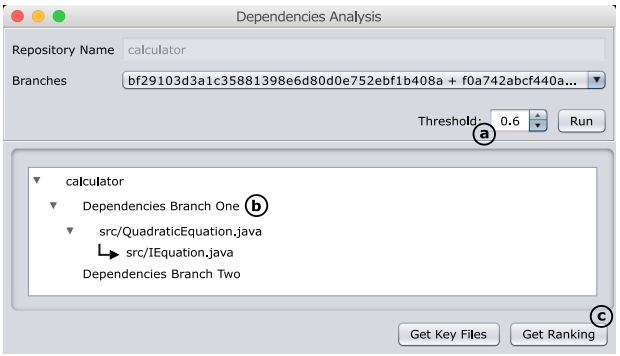


Fig. 4. File Dependencies

TIPMerge presents a ranked list of developers in the *Developer Ranking* window (Fig. 5). For each developer and each file (Fig. 5 (a)), it indicates the gold, silver, and bronze medals. It also shows the branch in which the change was made or if the change was made in the previous history (Fig. 5 (b)). Further information can be obtained through a tool tip, by hovering over the medal count. After getting the ranking, the user can ask for a team recommendation for collaborative merge by clicking on the *Recommend Collaborative Merge* button (Fig. 5 (c)).

In the *Collaborative Merge Configuration* window (Fig. 6), the user can configure the maximum number of developers (Fig. 6 (a)), the minimum coverage threshold, and the weights of each medal (Fig. 6 (b)). The user can also filter developers (Fig. 6 (c)). Finally, the user can set the stop conditions: the maximum time duration and the number of iterations that result in no changes (Fig. 6 (d)).

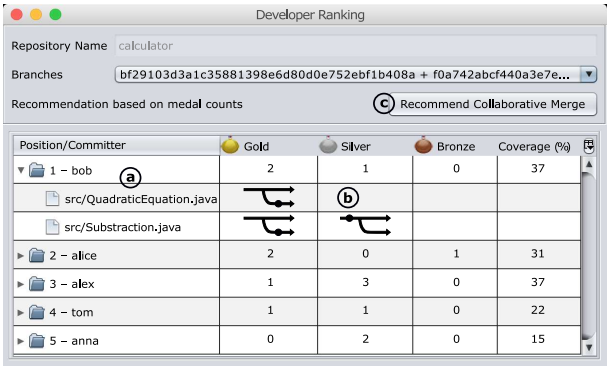


Fig. 5. Developer Ranking

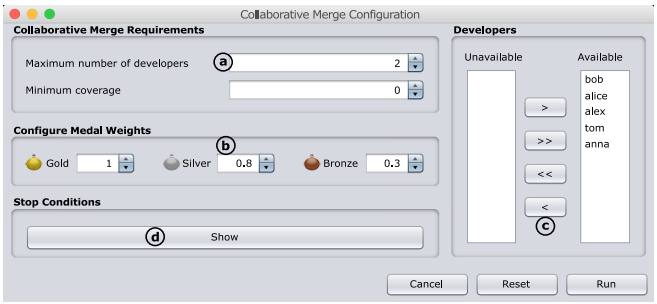


Fig. 6. Collaborative merge configuration

The *Team Recommendation for Collaborative Merge* window (Fig. 7) provides the maximum joint knowledge coverage for n developers (Fig. 7 (a)). Here (Table 6) Alice and Alex are in the first position since they have the highest joint knowledge coverage of 74% (Fig. 7 (b)). This window also shows the files that they changed together (Fig. 7 (c)).

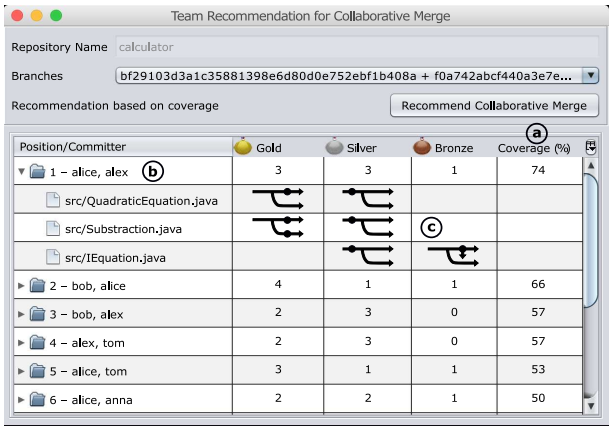


Fig. 7. Team Recommendation for Collaborative Merge

4 Evaluation

In order to assess the recommendation provided by TIPMerge, we answered the research question "*Is the top-n TIPMerge recommendation using the* ***Developer Ranking*** *more accurate than the top-n developers who performed the most merges in the past?*" in a previous paper [3]. The results indicated that 85% of the top-3 developers from the Developer Ranking correctly included the developer who performed the merge. Best (accuracy) results of recommendations were at 98%. Moreover, in 82% of the merges, TIPMerge obtained higher accuracy than selecting the developer who performed most of the previous merges (i.e., the majority class). Our interviews with developers of two projects revealed that in cases where the TIPMerge recommendation did not match the actual merge developer, the recommended developer had the expertise to perform the merge, or was involved in a collaborative merge session.

In this paper, we answer the research question "*Does the n-developer TIPMerge* ***Team Recommendation*** *for collaborative merge improve the percentage of joint knowledge coverage over teaming up the top-n developers from the* ***Developer Ranking****?*". We compared the knowledge coverage of the n-developer in the Team Recommendation (n = 2 & 3) with that of teaming up the top-n developers from the Developer Ranking. We checked the accuracy of n = 2 since pairing developers is a common practice, especially in Agile Methods. We explored n = 3 to increase the team size to the next level.

4.1 Materials and Methods

To form the project corpus for our experiment, we started from the 1,997,541 GitHub projects collected by Menezes et al. [32]. We then selected the top 1,000 unique projects ordered by the number of developers. From this set, we selected projects that were not forks and did not have a clear integrator for analysis. For each project, we checked: (1) whether the project includes merges, and (2) whether it does not comprise a sole developer performing the majority of the merges (majority class > 50%). The first criterion is self-explanatory. The second criterion is used to filter out those projects that either have an integration manager or a small subset of developers who are responsible for performing the merge. For instance, the project Git has a developer who performed 9,385 out of 9,699 total merges (96.76%). Such projects do not need a recommendation system and are filtered out from the dataset. After applying these criteria, we had 27 projects.

Ideally, we would like to select only branch merges. But we cannot identify such merges post hoc, as branches are merely pointers to the tip commit and do not leave traces in prior commits. Thus, we included a selection criterion: merges should include (1) changes performed by two or more developers in each branch and (2) key files. This criterion only includes branch merges; but we may overlook some simple branch merges with only one developer in a branch. This dataset thus errs on the side of being too much selective, eventually leading to false negatives. In two projects no merges fit the criteria. Consequently, this study analyzed 2,040 branch merges in total, from 25 projects (see Table 7), which represent the top 6% most complex merge cases in those projects.

TABLE 7
PROJECT CORPUS

Project	Lang	Dev	Date
activemerchant/active_merchant	Ruby	402	11/23/15
akka/akka	Scala	201	08/14/15
dschmidt/amarok	Ruby	196	05/07/12
angular/angular	TS	155	09/22/15
astropy/astropy	Python	142	09/30/15
apache/Cassandra	Java	103	06/01/15
mozilla/releases-comm-central	TS	300	11/20/15
errbit/errbit	Ruby	202	11/24/15
Netflix/eureka	Java	36	08/10/15
Netflix/falcor	JS	21	02/19/15
mozilla-mobile/firefox-ios	C	40	08/13/15
jquery/jquery	JS	227	12/06/09
Katello/katello	Ruby	61	08/14/15
dib-lab/khmer	Python	56	05/20/15
getlantern/lantern	Go	48	08/20/15
apache/maven	Java	45	06/22/15
nasa/mct	Java	13	06/11/12
Perl/perl5	Perl	373	06/12/15
apache/phoenix	Java	30	02/4/14
ComputationalRadiationPhysics/picongpu	C++	12	08/01/13
Netflix/Priam	Java	27	06/02/15
gems-uff/sapos	Ruby	10	07/30/15
spree/spree	Ruby	638	11/20/15
sympy/sympy	Python	385	03/01/16
voldemort/voldemort	Java	55	07/29/15

Next, we calculated the Developer Ranking joint knowledge coverage of top-2 and top-3 recommendations. We then compared the Team Recommendation joint knowledge coverage for 2 and 3 developers with the Developer Ranking joint knowledge coverage for top-2 and top-3, respectively. With these two values, we calculated the joint knowledge coverage improvement of the Team Recommendation to that of selecting from the Developer Ranking.

As the Team Recommendation requires to set the number of developers and the weights of the medals, we calibrated the weights of the medals through a sensitivity test. It used the following steps: (1) we randomly selected 120 merge cases of our dataset from four real projects (Akka, Sympy, Katello, and Cassandra) – these merges were discarded from the evaluation to avoid bias; (2) we set the number of developers to $n = 2$ and 3; (3) we set different medals weights varying from 1 to 0 with a step value of 0.1. For instance, $W_g = 1, W_s = 0.9, W_b = 0.8$; $W_g = 1, W_s = 0.9, W_b = 0.7$; $W_g = 1, W_s = 0.9, W_b = 0.6$; ...; $W_g = 0.3, W_s = 0.2, W_b = 0.1$, where $W_g > W_s$ and $W_s > W_b$. In total, the test had 120 different combinations of weights for each merge case.

Considering the 120 merge cases and the 120 different combinations of weights, we checked the number of merges where we had improvement. Four combinations of weights had the highest value (159 of 240 merges considering $n = 2$ and 3): $W_g = 1, W_s = 0.8, W_b = 0.3$; $W_g = 0.9, W_s = 0.8, W_b = 0.4$; $W_g = 0.8, W_s = 0.7, W_b = 0.4$; $W_g = 0.7, W_s = 0.6, W_b = 0.3$. Since we needed to choose a combination of weights, we defined in our experiment the following: $W_g = 1, W_s = 0.8, W_b = 0.3$. However, as discussed before, users can parameterize weights according to their needs.

Directly comparing our dependent variable (i.e., joint knowledge coverage) by their difference or direct proportion may lead to inflated results (>100% improvement), therefore, we use a measure for normalized improvement in knowledge coverage.

Fig. 8 shows two scenarios where the joint knowledge coverage difference between TR (Team Recommendation) and DR (Developer Ranking) is 10%. In the first scenario (Fig. 8 (a)), TR is 100% greater than DR (20% vs. 10%). In the second scenario (Fig. 8 (b)), TR is just 12% greater than DR (90% vs. 80%). If we simply calculate the difference in knowledge coverage, it would indicate that both scenarios are equivalent. On the other hand, if we perform a proportional comparison, it would indicate a much higher increase in the first scenario (100% vs. 12%).

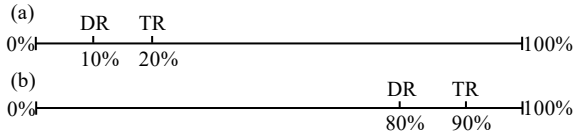


Fig. 8. Examples of improvement in a dependent variable

Intuitively, it is clear that creating an algorithm that improves an already high DR by 10% is much more difficult (and useful) than improving on a low DR by the same amount. For instance, the room for improving over DR in the first scenario is 90% (from 10% to 100%) and TR only reached 11% (10% ÷ 90%) of this potential. On the other hand, the room for improving over DR in the second scenario is only 20% (from 80% to 100%), but TR achieved 50% (10% ÷ 20%) of this gain.

We thus normalize the percentage of improvement in joint knowledge coverage by considering "the room for improvement" by using f_p [14]:

$$f_p = \begin{cases} \dfrac{TR_p - DR_p}{1 - DR_p}, if\ TR_p > DR_p \\ \dfrac{TR_p - DR_p}{DR_p}, otherwise \end{cases} \quad \text{(Eq. 1)}$$

Where TR_p represents the mean joint knowledge coverage obtained by the Team Recommendation for project p, and DR_p represents the mean joint knowledge coverage of the Developer Ranking for project p.

4.2 Results and Discussion

In our dataset (1,920 merges after discarding 120 merges used in the sensitivity test), we calculated the normalized improvement of joint knowledge coverage by the Team Recommendation over that of the Developer Ranking.

Table 8 lists the mean joint knowledge coverage for the top-2 developers in the Developer Rankings and for the optimized Team Recommendation (n = 2) for each project. We also list the normalized improvement of joint knowledge coverage (Eq. 1) by the Team Recommendation (color coded in the table). Team Recommendation led to improvements in coverage for all projects except the MCT

project. These improvements ranged from 2.2% (Falcor) to 52.4% (Phoenix), with a mean of 24.2% (median 21.2%). Projects with already high coverage in the Developer Rankings still improved with the Team Recommendations. The MCT project is an outlier as it had only one merge case with two developers in each branch. In this case, the developers in the top-2 Developer Ranking were the same as the Team Recommendation for two developers.

Table 8 also presents similar data for three-developer teams. The mean normalized improvement for three-developer teams is 49.5% (median 50.4%). In the Eureka project, the normalized improvement is 100% and in Phoenix it is 94.2% (from 91.3% to 99.5%), showing that the recommendation with three developers can achieve a very high or even full knowledge coverage.

TABLE 8
NORMALIZED IMPROVEMENT (NI) OF THE TEAM RECOMMENDATION (TR) OVER THE DEVELOPER RANKING (DR)

Project	Two-developer Teams			Three-developer Teams		
	DR (%)	TR (%)	NI (%)	DR (%)	TR (%)	NI (%)
Active Merchant	74.0	86.3	47.4	86.3	93.3	50.8
Akka	84.3	86.9	16.2	91.7	96.2	54.7
Amarok	68.0	71.5	10.8	77.8	81.9	18.6
Angular	56.5	61.3	11.0	73.1	75.3	8.3
Astropy	89.7	94.5	46.6	95.8	99.4	84.8
Cassandra	80.2	83.1	14.5	87.4	90.7	26.0
Comm-central	76.0	79.1	13.2	85.2	88.2	19.9
Errbit	73.1	78.3	19.6	85.4	88.7	22.2
Eureka	98.3	99.0	40.9	99.0	100.0	100.0
Falcor	93.2	93.4	2.2	96.0	98.8	69.5
Firefox for iOS	84.1	88.5	27.8	94.2	98.2	68.9
jQuery	79.1	89.0	47.1	87.8	97.7	80.8
Katello	74.0	78.0	15.7	84.2	88.4	26.8
Khmer	91.3	94.3	34.3	94.3	99.2	86.4
Lantern	89.5	92.5	29.1	91.3	97.4	70.7
Maven	89.2	92.4	29.1	95.8	96.2	10.7
MCT	74.1	74.1	0.0	89.7	89.7	0.0
Perl5	81.8	84.2	13.3	88.9	92.1	29.0
Phoenix	90.2	95.3	52.4	91.3	99.5	94.2
PIConGPU	88.1	92.9	40.4	95.6	98.6	68.9
Priam	93.6	94.7	18.1	96.3	99.1	76.0
Sapos	85.1	86.3	7.9	95.1	97.5	48.5
Spree	78.7	83.7	23.6	87.6	92.2	37.2
Sympy	80.3	84.5	21.2	89.1	92.9	34.5
Voldemort	81.2	85.7	24.0	91.5	95.8	50.4
Mean			24.2			49.5
Median			21.2			50.4

Fig. 9 uses box plots to compare the joint knowledge coverages from the Team Recommendation and the Developer Ranking for two- and three-developer teams. It shows that the Team Recommendation consistently does better.

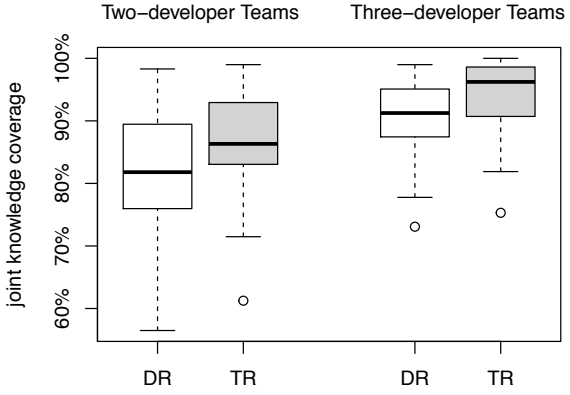


Fig. 9. Developer Ranking (DR) and Team Recommendation (TR) joint knowledge coverage distributions

To test for normality in the knowledge coverage data, we used Shapiro-Wilk test ($H_ø$ = population is normally distributed). We observed normality in the data for two-developer teams (Developer Ranking p-value = 0.438 and Team Recommendation p-value = 0.090) at 95% confidence level. We did not observe normality in the data for three-developer teams (Developer Ranking p-value = 0.052 and Team Recommendation p-value = 0.001) at 95% confidence level.

Thus, for the two-developer teams we used Fisher's F test (HØ = both populations have the same variance) to check homoscedasticity and found that the variance is the same (p-value = 0.750). Therefore, we applied the paired t-test and observed a statistically significant difference (p-value < 0.001) at 95% confidence level. We applied Wilcox paired test for three-developer teams. We again observed a statistically significant difference (p-value < 0.001) at 95% confidence level.

We thus conclude that Team Recommendation for two and three developers has higher joint knowledge coverage in comparison to Developer Ranking for top-2 and top-3 developers. Finally, we checked the paired Cohen's d effect size (used for normal population) for two-developer team and the Cliff's Delta effect size (used for non-normal population) for three-developer teams. We observed a large effect size (1.416) [34] in the former and a medium effect size (0.451) [35] in the latter. A lower effect size for three-developer teams was expected. As stated before, increasing an already high value is harder. However, the mean normalized improvement for three-developer teams is expressive (49.6%).

We also analyzed whether the project characteristics, such as number of commits, number of developers, number of merges, and number of branch merging, correlate with the results, but we could not find any relevant correlation.

In terms of execution time, TIPMerge presented different times for different merge cases. TIPMerge generated the Developer Ranking in few seconds for some cases, but took more than one hour for others. The execution time depends on the number of changed files; the larger the number of files the longer it is to check the dependencies. For example, in a usual merge case with 29 files changed in parallel and with dependencies, TIPMerge generated the Developer Ranking with 7 developers in 1 second and 709 milliseconds, running on a i7 4790k (4GHz) with 4 cores and 16 GB of memory (2 x 8 GB DDR3 1600). On the other hand, in a more extreme merge case with 237 files changed in parallel and with dependencies, TIPMerge needed 1 hour, 75 minutes, and 94 seconds to generate the Developer Ranking with 265 developers, running on the same computer. After obtaining the Developer Ranking, we applied our optimization algorithm in the latter case, and we obtained the Team Recommendation for two developers in 3 minutes and 1 second, and for three developers in 5 minutes and 6 seconds, in addition to the time needed to obtain the Developer Ranking. This shows that the overhead for obtaining the Team Recommendation is just a small fraction of the total time, which includes the computation of the Developer Ranking.

5 THREATS TO VALIDITY

As in any study, our study has limitations. First, our approach uses the committers' Git ID to identify developers.

It is possible that developers use multiple aliases. We manually verified the TIPMerge ranking with the merge developer to fix possible mistakes by considering their ID similarity. Although this suffices in most cases, we may have missed some cases when the aliases are lexically different.

Second, we checked for merges with at least two unique developers in each branch. In these cases, there is no workspace merge, which is a simpler scenario than branches with a large number of contributors. However, we are exposed to false negatives, as some merges of branches may not be listed (branch merges with only one person on a branch).

Finally, although we have commented about considering only changes in a given time frame (e.g., past release) to accommodate decay in expertise, we considered all the project history in our experiments. As the projects have different time frames, it can have some impact on the results.

6 Related Work

To the best of our knowledge, there is no work that addresses recommendation of teams to collaborative merge of branches. The more closely related works provide support for the identification of experts in software projects. Besides that, some proposals focus on supporting collaborative model merging.

6.1 Identification of Experts

Some approaches, such as Dominoes [24], [31], Emergent Expertise Locator [36], Expertise Browser [37], and Usage Expertise [38], aim at identifying experts in software projects. Some of these approaches (Dominoes and Emergent Expertise Locator) are based on the approach by Cataldo et al. [39], [40], who developed a technique to measure task dependencies among people. They use matrices to represent various dependency relationships. From this, they aim at answering who must coordinate with whom to get the work done. Dominoes [24], [31] allows different kinds of explorations over matrices, and can be used to identify experts for a given project or software artifact. Dominoes [24], [31] is capable of using GPU for processing operations, which enables the analysis of large-scale data. Emergent Expertise Locator [36] produces a ranked list of the most likely emergent team members with whom to communicate, given a set of files currently deemed to be of interest. Expertise Browser [37] identifies experts over regions of the code, such as modules or even subsystems by using the concept of: (1) Experience Atoms (EAs), which are basic units of experience in change management systems, and (2) the atomic change (delta) made to the source code or to the project's documentation. Finally, the concept of Usage Expertise [38] is introduced to recommend experts for files, where the developer accumulates expertise not only by editing methods, but also by calling (using) them.

All these approaches extract information from the Version Control Systems and Issue Tracking Systems. Similar to TIPMerge, some of these approaches analyze changes performed via commits. Other approaches check for different kinds of information, such as a method calls, opened and closed issues, etc. While all these approaches identify experts, they only take into consideration previous history, and do not discern changes in branches. As a result, equal weights are assigned to all files. However, in our situation we know that changes across branches and their dependencies might have a bigger impact on the merge decision than prior changes alone.

Other studies on identification of experts have focused on pull request assignment [17]–[23] and factors that influence the reviewer assignment to pull requests [41]. Yu et al. [19]–[21] proposed an approach that combines information retrieval with social network analysis to help project managers find a suitable reviewer for each pull request. Jiang et al. [22], [23] proposed CoreDevRec to recommend core members for contribution evaluation in GitHub. CoreDevRec [23] uses support vector machines to analyze different kinds of features, including file paths of modified code, relationships between contributors and core members, and activeness of core members. De Lima Júnior et al. [17], [18] proposed the use data mining to identify the most appropriate developers to analyze a pull request. They use a set of attributes and classification strategies to suggest developers to analyze pull requests. Also using data mining, Soares et al. [41] mined association rules from 22,523 pull requests belonging to 3 projects and identified factors that influence the reviewer assignment to pull requests, such as that some reviewers have more chances to analyze pull requests containing files that they have recently changed.

These works are closely related to the recommendation of developers for branch merging, as they aim to recommend developers to verify the actual contribution and possibly merge it with the rest of the project. Nevertheless, in general, pull requests contain commits of a single developer and are small [42]. Moreover, the author of the pull request usually syncs their forked branch in advance to ease reintegration, making the process more like a workspace merge. In the more general case of merging branches, the number of developers, the syncing interval, and the number of commits per branch is variable and can be high in some situations.

6.2 Collaborative Merge Support

Some proposals focus on supporting collaborative model merging [4], [7] and collaborative real-time editor [5], [8]. Koegel et al. [4] propose an approach to collaborative merging to facilitate discussion on conflicts and collaborative conflict resolution. The approach is based on the application and integration of Rationale Management into Model Merging [4]. Brosch et al. [7] propose to use a model checker to detect semantic merge conflicts in the context of model versioning. This technique is used to check the semantic consistency of an evolving UML sequence diagram with respect to state machine diagrams that remain unchanged [7]. Lautamäki et al. [8] and Nieminen [5] present a process for real-time collaborative merging as well as a web-based tool supporting the process, CoRED. A collaborative real-time editor must somehow manage concurrent edits by different users by applying each users' changes into the shared document as well as possible [5].

These studies aim at enabling all involved participants to work in a collaborative fashion. However, they do not address how people should be chosen to participate in the collaborative merge session. Studies based on model checkers try to create strategies to facilitate the collaborative modeling and works based on collaborative real-time editors try to create an environment to help people resolve the conflicts collaboratively. Nevertheless, these approaches still need support for choosing the right people to collaborate, being complementary to TIPMerge.

7 Conclusion

In this research, we propose TIPMerge, a novel approach that analyzes changes in branches, file dependencies, and the past history to build a ranked list of developers who are supposed to be the most appropriate to integrate a pair of branches. Using this ranked list, TIPMerge is able to recommend the most suitable developers in terms of *joint knowledge coverage to a collaborative merge session*. TIPMerge can help software teams in complex merge cases, where changes in key files can lead to direct and indirect conflicts.

We quantitatively evaluated TIPMerge over 25 real-world projects. The Team Recommendation for two- or three-developer teams led to a normalized improvement of 21.2% (median) and 50.4% (median) for collaborative merge in comparison to choosing the top-2 or top-3 developers in the ranked list, respectively.

As a future work, we intend to qualitatively evaluate the TIPMerge Team Recommendation in contrast to the team composed by the top-n developers in the Developer Ranking by interviewing the involved developers and asking which team is the most appropriate for some merge case. Besides that, we would like to identify attributes that indicate when merges should be performed by a single developer or team. TIPMerge could also consider other information to recommend developers, such as developer skills and social relationships. For instance, the skills on specific design patterns or programming language features could be considered during assignment. Besides that, as TIPMerge already has the option of not considering one or more developers for a recommendation, we suggest that TIPMerge could have an option to force someone to be part of the recommendation for a collaborative merge, helping on-boarding new developers.

Acknowledgment

This work is partially supported by CAPES (10614-14-1), CNPq (305569/2014-7 and 306137/2017-8), FAPERJ (E-26/201.523/2014), and NSF CCF- 1560526 and IIS-1559657.

References

[1] C. Bird and T. Zimmermann, "Assessing the Value of Branches with What-if Analysis," in *ACM SIGSOFT Int'l Symp. Foundations of Software Eng (FSE)*, New York, NY, USA, 2012, p. 45:1-45:11.

[2] C. Bird, T. Zimmermann, and A. Teterev, "A theory of branches as goals and virtual teams," in *4th International Workshop on Cooperative and Human Aspects of Software Engineering*, New York, NY, USA, 2011, pp. 53–56.

[3] C. Costa, J. Figueiredo, L. Murta, and A. Sarma, "TIPMerge: Recommending Experts for Integrating Changes Across Branches," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2016, pp. 523–534.

[4] M. Koegel, H. Naughton, J. Helming, and M. Herrmannsdoerfer, "Collaborative model merging," in *ACM international conference companion on Object oriented programming systems languages and applications companion*, New York, NY, USA, 2010, pp. 27–34.

[5] A. Nieminen, "Real-time collaborative resolving of merge conflicts," in *2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2012, pp. 540–543.

[6] C. Costa, J. J. C. Figueiredo, G. Ghiotto, and L. Murta, "Characterizing the Problem of Developers' Assignment for Merging Branches," *Int. J. Softw. Eng. Knowl. Eng. IJSEKE*, vol. 24, no. 10, pp. 1489–1508, 2014.

[7] P. Brosch *et al.*, "Towards semantics-aware merge support in optimistic model versioning," in *Models in Software Engineering*, Berlin, Heidelberg, 2012, pp. 246–256.

[8] J. Lautamäki, A. Nieminen, J. Koskinen, T. Aho, T. Mikkonen, and M. Englund, "CoRED: browser-based Collaborative Real-time Editor for Java web applications," in *ACM 2012 conference on Computer Supported Cooperative Work*, New York, NY, USA, 2012, pp. 1307–1316.

[9] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–35, 2011.

[10] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *4th International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, 2007, pp. 2–2.

[11] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix This Bug?," in *International Conference on Software Engineering (ICSE)*, New York, NY, USA, 2006, pp. 361–370.

[12] Y. C. Cavalcanti, P. A. da Mota Silveira Neto, I. do C. Machado, T. F. Vale, E. S. de Almeida, and S. R. de L. Meira, "Challenges and opportunities for software change request repositories: a systematic mapping study," *J. Softw. Evol. Process*, vol. 26, no. 7, pp. 620–653, 2014.

[13] H. Kagdi and D. Poshyvanyk, "Who can help me with this change request?," in *IEEE 17th International Conference on Program Comprehension, 2009. ICPC '09*, 2009, pp. 273–277.

[14] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 2012, pp. 451–460.

[15] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *IEEE International Working Conference on Mining Software Repositories*, Vancouver, BC, 2009, pp. 131–140.

[16] T. Zhang and B. Lee, "How to Recommend Appropriate Developers for Bug Fixing?," in *IEEE 36th Annual Computer Software and Applications Conference*, Izmir, 2012, pp. 170–175.

[17] M. L. de Lima Júnior, D. M. Soares, A. Plastino, and L. Murta, "Automatic assignment of integrators to pull requests: The importance of selecting appropriate attributes," *J. Syst. Softw.*, vol. 144, pp. 181–196, Oct. 2018.

[18] M. L. de Lima Júnior, D. M. Soares, A. Plastino, and L. Murta, "Developers Assignment for Analyzing Pull Requests," in *30th Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2015, pp. 1567–1572.

[19] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?," *Inf. Softw. Technol.*, vol. 74, pp. 204–218, Jun. 2016.

[20] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Reviewer Recommender of Pull-Requests in GitHub," in *IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, 2015, pp. 609–612.

[21] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Who Should Review this Pull-Request: Reviewer Recommendation to Expedite Crowd Collaboration," in *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, Jeju, 2014, vol. 1, pp. 335–342.

[22] J. Jiang, Y. Yang, J. He, X. Blanc, and L. Zhang, "Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development," *Inf. Softw. Technol.*, vol. 84, pp. 48–62, Apr. 2017.

[23] J. Jiang, J.-H. He, and X.-Y. Chen, "CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation," *J. Comput. Sci. Technol.*, vol. 30, no. 5, pp. 998–1016, Sep. 2015.

[24] J. R. da Silva, E. Clua, L. Murta, and A. Sarma, "Niche vs. breadth: Calculating expertise over time through a fine-grained analysis," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Montréal, Québec, Canada, 2015, pp. 409–418.

[25] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Washington, DC, USA, 2004, pp. 563–572.

[26] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.

[27] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1994, pp. 487–499.

[28] G. A. Oliva and M. A. Gerosa, "On the Interplay between Structural and Logical Dependencies in Open-Source Software," in *2011 25th Brazilian Symposium on Software Engineering (SBES)*, Sao Paulo, Brazil, 2011, pp. 144–153.

[29] L. P. Hattori and M. Lanza, "On the nature of commits," in *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, L'Aquila, Italy, 2008, pp. 63–71.

[30] F. Glover, "Tabu search—part I," *ORSA J. Comput.*, vol. 1, no. 3, pp. 190–206, 1989.

[31] J. R. da Silva Junior, E. Clua, L. Murta, and A. Sarma, "Exploratory Data Analysis of Software Repositories via GPU Processing," in *The International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Vancouver, Canada, 2014, pp. 495–500.

[32] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. V. D. Hoek, "On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub," *IEEE Trans. Softw. Eng.*, pp. 1–1, 2018.

[33] G. L. Pappa and A. A. Freitas, "Automatically evolving rule induction algorithms," in *Machine Learning: ECML 2006*, Springer, 2006, pp. 341–352.

[34] J. Cohen, "A power primer," *Psychol. Bull.*, vol. 112, no. 1, pp. 155–159, 1992.

[35] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys," in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

[36] S. Minto and G. C. Murphy, "Recommending Emergent Teams," in *Fourth International Workshop on Mining Software Repositories (MSR)*, Washington, DC, USA, 2007, p. 5–.

[37] A. Mockus and J. D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," in *24th International Conference on Software Engineering (ICSE)*, New York, NY, USA, 2002, pp. 503–512.

[38] D. Schuler and T. Zimmermann, "Mining Usage Expertise from Version Archives," in *International working conference on Mining software repositories (MSR)*, New York, NY, USA, 2008, pp. 121–124.

[39] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2008, pp. 2–11.

[40] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the Design of collaboration and awareness tools," in *20th anniversary conference on Computer supported cooperative work (CSCW)*, 2006, pp. 353–362.

[41] D. M. Soares, M. L. de Lima Júnior, A. Plastino, and L. Murta, "What factors influence the reviewer assignment to pull requests?," *Inf. Softw. Technol.*, vol. 98, pp. 32–43, Jun. 2018.

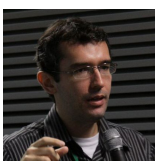
[42] G. Gousios, M. Pinzger, and A. van Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014, pp. 345–355.

Catarina Costa is a Professor at Universidade Federal do Acre (UFAC). She holds a Ph.D. (2017) degree in Computer Science from Universidade Federal Fluminense (UFF), a M.S. (2010) degree also in Computer Science from Universidade Federal de Pernambuco (UFPE), and a B.S. (2007) degree in Informatics from UFAC. Her research area is software engineering, and her current research interests include configuration management, software evolution, and global software development.

Jair Figueiredo is a Software Engineer at Federal Institute of Acre (IFAC). He holds a M.S. (2011) degree in Computer Science from Universidade Federal de Pernambuco (UFPE) and a B.S. (2007) degree in Informatics from Universidade Federal do Acre (UFAC). His research area is software engineering and performance evaluation and his current research interests include configuration management and data engineering.

João Felipe Pimentel is a PhD student at Universidade Federal Fluminense (UFF). He graduated with Honors from UFF with a degree in Computer Science (2014). He was an undergraduate research student (CNPq), acting on automatic refactoring of source code. Currently, he is working with provenance from scripts and interactive notebooks.

Anita Sarma is an Associate Professor in the School of Electrical Engineering and Computer Science, at Oregon State University. She holds a Ph.D. degree in Computer Science from the University of California, Irvine. Her research interests lie primarily in the intersection of software engineering and computer-supported cooperative work, focusing on understanding and supporting coordination as an interplay of people and technology. She has over 100 papers in journals and conferences. Her work has been recognized by an NSF CAREER award as well as several best paper awards.



Leonardo Gresta Paulino Murta is an Associate Professor at the Computing Institute of Universidade Federal Fluminense (UFF). He holds a Ph.D. (2006) and a M.S. (2002) degree in Systems Engineering and Computer Science from COPPE/UFRJ, and a B.S. (1999) degree in Informatics from IM/UFRJ. He has published over 150 papers in journals and conferences and received an ACM SIGSOFT Distinguished Paper Award at ASE 2006 and three best paper awards at SBES, in 2009, 2014, and 2016. His current research interests include configuration management, software evolution, software architecture, and provenance.