Optimized Assignment of Developers for Fixing Bugs

Md. Mainur Rahman  
Department of Computer Science  
University of Calgary  
1-403-284-3486  
mmrahma@ucalgary.ca

Guenther Ruhe  
Software Engineering Decision Support Laboratory  
University of Calgary  
1-403-220-7692  
ruhe@ucalgary.ca

Thomas Zimmermann  
Microsoft Research  
One Microsoft Way  
Redmond, WA 98052 USA  
1 425 703-8450  
tzimmer@microsoft.com

ABSTRACT

Decisions on “Who should fix this bug” have substantial impact on the duration of the process and its results. Expertise and related productivity level of developers might vary up to one order of magnitude. This is even more the case if we acknowledge that fixing a bug typically requires expertise in a number of components.

In this paper, optimized strategies for the assignment of the “right” developers for doing the “right” task are studied and the results are compared to manual (called ad hoc) assignment. The quality of assignment is measured by the match between requested (from bugs) and available (from developers) competence profile. Different variants of Greedy search with varying parameter of look-ahead time are studied.

The quality of the results has been evaluated for nine milestones of the open source Eclipse JDT project. The optimized strategies with largest look ahead time are demonstrated to be substantially better than the ad hoc solutions in terms of the quality of the assignment and the number of bugs which can be fixed within the given time interval.

Categories and Subject Descriptors

D.2 [Software]: Software Engineering.

General Terms

Project and People Management.

Keywords

Empirical evaluation, bug fixing, human resource management, open source data.

1. INTRODUCTION

Completing maximum number of jobs (a bug, feature request, task) by a limited number of developers in a quantifiable way has potential impact on the quality of the bug fixing process as well as on the number of bugs which can be fixed in a given time interval. Typically, the number of bugs by far exceeds the capacity available to fix them. In addition, often there is a shortage of skilled and experienced personnel. Looking at scenarios of having hundreds (if not thousands) of bugs in a repository, a proper assignment of developers to bugs becomes a tremendous challenge. Each developer available might have different level of experience when performing different types of tasks. Good resource allocation, i.e. assigning the most qualified developers to the necessary tasks, takes on to be even more importance [1].

Considering skill profile, capacity and experience record of the developers for allocating them to jobs is challenging for decision makers. Decision based on only intuition may lead to poor utilization of resources. Poor staffing might cause delay as well. The skill and productivity level of developers might vary up to one order of magnitude. This is even more the case if we consider different types of tasks or tools needed for the implementation of a specified amount of functionality.

Looking at some form of incremental development, we are studying optimized strategies for assigning developers to bugs that will enable the assignment of the best qualified developers to fix the maximum numbers of weighted (by size) bugs. We consider a pool of developers with a documented development history on tasks formerly done. In the same way, a set of tasks is considered, which are pre-selected for the development. The look-ahead time is flexible and can vary from weekly to a monthly time horizon. Correspondingly, the set of tasks of consideration will vary. The questions relating to this are:

(i) Can the pre-selected number of bugs be fixed with the personal available?

(ii) If the answer of (i) is “yes”, the follow-up question is: How to optimally assign developers to tasks for achieving the implementation of the proposed functionality?

(iii) If the answer of (i) is “no”, the follow-up question is: What needs to be done to adjust the existing plan of functionality?

The proposed strategies have been empirically evaluated for a set of nine data sets from open source Eclipse JDT project [10]. In Section 2, we report about related work. Problem description and
formalization is given in Section 3. The optimized solution strategies are the content of Section 4. Their evaluation for the Eclipse data sets is following in Section 5. Discussion of the results and their validity is presented in Section 6. The paper concludes in Section 7 with a summary and an outlook to future research.

2. RELATED WORK

Many researchers have worked on improving bug management, especially in the context of open source bug repositories [2]. Anvik et al. [3] addressed the problem of assigning bug fix tasks to software developers and used supervised machine learning algorithm successfully for Eclipse [10] and Firefox [11] projects. They analyzed a bug repository with machine learning algorithms to identify the kinds of bugs that each developer usually resolves. These developers have been assigned to new bugs pertaining to the same categories. This technique produces a classifier for new reports and suggests the developers suitable for resolving the reports.

Anvik et al. [3] considered only the contents (bug description, affected component) of the bug report without considering the estimated effort. Similar, Cubranic and Murphy tried to establish the connection between bug reports and program features by applying supervised machine learning using a naïve Bayes classifier to automatically assign reports to the developers for Eclipse project. This bug assignment mechanism of a “multi-class, single-label classification problem” where the developers are the classes (each developer is a separate class) and each bug report belongs to a single class, i.e., will be assigned to exactly one developer. Other approaches for bug assignment have been proposed by Canfora and Cerulo [13] and by Bettenburg et al. [14]. All these approaches solely focus on the contents of bug reports; in particular, they assume unlimited availability of developers, which is not the case. The work in this paper, presents a solution for this problem by combining bug assignment with release planning.

Another frequent problem in bug triage is the detection of duplicate bug reports. Wang et al. [5] proposed a heuristic based on classification to detect duplicate bugs for Eclipse and Firefox. In their approach they considered natural language information as well as execution information to deal with both the external and internal abnormal behavior of the bugs. Upon reporting of a bug, the method calculates the similarities between the newly reported bug and the existing reports and uses heuristic to find the similarities between the reports and determines the duplication. Earlier approaches on duplicate detection solely considered the content of bug reports, and not any runtime data [15,16,17].

Anvik [6] argued that the process of bug triage cannot be fully automated. It requires some sort of human interaction because of the required contextual knowledge to make decisions. Therefore, he suggested a semi-automated approach that will recommend a set of suitable developers to whom the bug report can be assigned. The recommendation would be based on the model of expertise of the developers, which would be designed using information about previous expertise of the developers. Weiss et al. [7] built a model to estimate the time it will take to fix a new bug. This automated model relieves managers who have a long queue of bug reports waiting to be estimated. The model also supports better allocation of resources and scheduling future stable releases, however only manually. In this paper, we present an approach that combines the approaches by Anvik et al. and Weiss et al. The technique in this paper helps to better allocate resources and schedule bugs based on both expertise and effort estimates.

It is well known [8] that productivity may vary significantly among developers. Each developer might have different productivity when performing different types of tasks. Acuna et al. [8] emphasized on identifying the relationship between software development roles and the capabilities required to perform them. The project type and the development organization can be used to identify the roles. The assignment can be done by using the ratio of necessary capabilities and the required capacities for a particular job.

Kapur et al. [1] considered the productivity and availability of the developers while assigning tasks to them for product releases. Ngo-The et al. [9] have introduced an average skill level with a normalized productivity factor of 1.0 for each type of task. This allows the manager choosing from a group of more or less skilled developers with a higher or lower productivity factor than 1, accordingly. They assumed that the project manager is able to evaluate the different degrees of productivity of the developers in consideration.

3. PROBLEM DESCRIPTION

We consider a process of fixing a given set BUGS of bugs, which are considered for fixing in a given time (release interval). An individual bug from this set is denoted by bug(n). We assume to have N bug in consideration, i.e. BUG = [bug(1) … bug(N)].

Our time interval is defined as [0,T], i.e., we have a planning horizon of T time units (days) in consideration for fixing the N bugs. But who should fix which bug and when? There is assumed a pool of developers DEVELOPERS consisting of D individual developers available during the time interval.

To evaluate the degree of fitness of assigning a developer dev(d) to bug(n), we model the requested and the available skill profiles, respectively. In the context of the organization and/or the project, there is a set of competencies, which have been observed to be relevant for fixing bugs. These competencies can be related to the usage of certain techniques or tools. Competency here means a combination of knowledge and experience in the area in consideration. We summarize all these competencies into a vector called having K components. For example, this can be a competence in programming Java and in using a testing tool such as JUnit.

For the assignment of developers to bugs, we define requested and available skills from the perspective of a bug(n) to be fixed and a developer dev(d), respectively. We also call this two profiles Request(n) and DevProfile(d), respectively. Both of these vectors have the same number K of components as the CompProfile vector.

Finally, we assume that the individual components k = 1…K of both Request(n) and DevProfile(d) are denoted by Request(n,k) and DevProfile(d,k). The individual skill levels are formulated by numbers from interval [0,1] expressing the skill level in the respective component of the vector. Both vectors have been normalized to 1:

\[ \sum_{k=1..K} \text{Request}(n,k) = 1 \text{ for all } n = 1..N \]
In order to find optimized assignments of developers to bugs, we need to define a distance measure between the requested and the available skill profiles. For a particular bug(n) and a given developer dev(d), we define the distance dist(n,d) as follows:

\[ \text{Dist}(n,d) = \sum_{k: \text{Request}(n,k) \neq 0} \text{abs}((\text{Request}(n,k) - \text{DevProfile}(d,k))) \]

The function abs(x) denotes the absolute value of x. The summation is done over all skill profile components k. Based on distance, we define similarity as the inverse of distance.

\[ \text{Sim}(n,d) = 1 - \text{Dist}(n,d) \]

The higher the similarity value, the better is the match between the two profiles, e.g., the more appropriate the assignment of developer dev(d) to fix bug(n) would be.

For the formulation of the solution approach, we also consider the priority pri(o(n)) assigned to bug(n). Priorities are defined on a five point scale with pri(o(n)) = 1 being highest and pri(o(n)) = 5 being lowest priority. In addition, the size of a bug is expressed by the total number of K source lines of code (KSoC) impacted by fixing the bug. These can be new, revised or deleted lines of code.

The fitness between profiles goes beyond similarity and takes into account also the priority of the bug and its size.

\[ \text{Fitness}(n,d) = \text{Sim}(n,d) \times \text{size}(n) \times \text{priority}(n) \]

For each bug(n), open(n) denotes the opening date of the bug. This is the date when the bug has been assigned to the developer in the original ad hoc assignment set-up.

Similarly, all the developers have the profile for their estimated productivity with respect to perform the different types of activities related to fixing a specified bug. We refer to this profile as productivity of the developers. Different fields describe how much experience the developers have with changing files in each of these categories. The productivity is also normalized to 1 and the sum of all the productivities is 1. For example, a value of 0.509 for JDT means the developer is about 50.9% productive for fixes related to JDT.

Our problem statement is relying on an existing manual (called ad hoc) assignment of developers to bugs. For each milestone in consideration, a fixed number of bugs is given. The question is to fix all the bugs more effective and more efficient by assigning developers having a higher fitness with the requested bug competence profile Request(n).

In this paper we answer the following questions:

1. How does the quality of the assignment of developers to bugs change when comparing the baseline manual assignment with the results from Greedy-X? The quality here is defined as the fitness between requested and assigned competence profile.

2. How does the total fitness of the assignment of developers to bugs change when comparing the baseline manual assignment with the results from Greedy-X?

3. Our final analysis question is devoted to the total time savings gained from optimized allocation of developers to bugs.

4. SOLUTION APPROACH

The problem formulated in Section 3 is a special case of the resource-constraint project scheduling problem known to be NP-complete [19]. Our proposed heuristic solution method is based on Greedy search [18]. The idea is to look at each stage of the decision-making process at the locally best choice, hoping that this strategy end up in something which is not too far away from optimum. Greedy search has been proven successful in different contexts, and is proving even the guaranteed optimal solution for the class of minimum spanning tree problems.

Greedy search is based on some given order of the objects in consideration. In our case, this order is defined by the opening dates of the bugs, e.g., we consider the bugs in the same order as they were opened in case of the manual assignment. At each step where a new bug is assigned to an idle developer, the selection is done based upon the match of fitness between the requested competence profile and the ones available form the developers.

To overcome the limitations of making local decisions, which eventually later prevent from assigning more promising developers to fixing a bug, we have introduced additional look-ahead time for the search. That means, for a given point in time, we not only look at the idle developers at this time, but consider all developers becoming available within the given look-ahead time.

Our solution method relies on a number of assumptions:

Assumption 1: Each developer fixes exactly one bug at the time

Assumption 2: Each bug is fixed by exactly one developer

Assumption 3: Once assigned to a bug, each developer is working continuously on this bug.

Assumption 4: The productivity of a developer is inverse proportional to the effort required to fix a bug (where the time is known).

Assumption 5: The duration for fixing a bug is defined by the ration between closing and opening date as defined from the manual assignment divided by the ration in expertise between the new versus manual assignment.

The status of validity of the assumptions is discussed later in Section 6. A pseudo-code description of Greedy-X, the application of the greedy search with a look-ahead time X, is presented below.

**Procedure Greedy-X**

For re-assigning developers to bugs, we assume the following:

- N bugs, each of them having an assigned developer d*(n)
- Developer dev^*(n) is working on bug(n) in the interval [open(n), close(n)]
- Bugs are arranged in increasing order with respect to their opening date.
- For all developers dev(d) (d = 1...D), we have their competence profile DevProfile(d)
- For all bugs bug(n) (n=1...N), we have their (requested) bug profiles Request(n)
- IDLE(t) := {d from DEVELOPER: d idle at some point in time in [t,t+X]}

For each of the bugs, look into the developers set to find the idle developers at point in time open(t). If an individual developer is not currently available but would be with the specified (X) look-ahead time, mark this as special. For the bug in consideration find the distance from the requested profile for all the available developers as well as special ones according to equation (3). Assign the bug to the developer with the lowest distance. The assigned developer will start working on the bug according to the opening date of the bug if the developer is in idle state. Otherwise (look-ahead/special), fixing the bug would start from the date when the assigned developer would be available.

For determining the end date for fixing the bug using Greedy-X assignment, we need to find the expertise level of the baseline and Greedy-X along with the improvement. The baseline expertise level is formulated as

\[
\text{Expertise}_X(n) = \sum_{k: \text{Request}(n,k) \neq 0} \text{DevProfile}(d(n),k).
\]

Similarly, the expertise for the assignment of a developer to requested competencies of a bug as suggested by Greedy-X is defined as

\[
\text{Expertise}_X(n) = \sum_{n=1..N} \text{Expertise}_X(n) - \sum_{n=1..N} \text{Expertise}_{\text{Adhoc}}(n)/\sum_{n=1..N} \text{Expertise}_{\text{Adhoc}}(n).
\]

With the above two definitions, we are able to express in (8) the relative improvement (in %) for expertise gained from the Greedy-X assignments when compared to ad hoc assignment:

\[
\text{Expertise}_X(n) = \sum_{n=1..N} \left( \text{Expertise}_X(n) - \text{Expertise}_{\text{Adhoc}}(n) \right)/\sum_{n=1..N} \text{Expertise}_{\text{Adhoc}}(n)
\]

The end date for fixing the bug is determined by deducting the product of total duration of the bug and equation (8) from the total duration of the bug. The assigned developer is assumed to work on that particular bug for the time period between start and end date of fixing the bug.

BEGIN

IDLE(t) = DEVELOPER for all t

For all n = 1...N do

BEGIN

Determine d^*(n) = arg max{ Fitness(n,d) with d from IDLE(t)}

Assign dev(d^*(n)) to bug(n)

IDLE(t) + IDLE(t) – {dev(d^*(n))} for t from [open(n),close*(n)]

end

END

5. CASE STUDY FOR ECLIPSE JDT

5.1 Description of the data sets

In this paper we are considering JDT. The JDT project provides the tool plug-ins that implements a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. Each of the bugs in the repository has the profile description. For fixing the bugs the developers need to modify certain number of lines in different components for a particular product.

The JDT project allows Eclipse to be a development environment for itself [12]. The JDT project is broken down into components. Each component operates like a project unto its own, with its own set of committers, bug categories and mailing lists. Table 1 gives a description of the components looked at for this study.

### Table 1. JDT components and their description

<table>
<thead>
<tr>
<th>Component name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APT</td>
<td>APT provides a means for generating files and compiling new java classes based on annotations found in the source code. <a href="http://www.eclipse.org/jdt/apt/index.html">http://www.eclipse.org/jdt/apt/index.html</a></td>
</tr>
<tr>
<td>Core</td>
<td>JDT Core is the Java infrastructure of the Java IDE. <a href="http://www.eclipse.org/jdt/core/index.php">http://www.eclipse.org/jdt/core/index.php</a></td>
</tr>
<tr>
<td>Doc</td>
<td>Java Development Toolkit Documentation and Help Content.</td>
</tr>
</tbody>
</table>

In addition, the components of Table 2 are relevant for performing the bug fixing process.

### Table 2. Additional components relevant for bug fixing

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>Eclipse Platform Compare framework.</td>
</tr>
</tbody>
</table>
A bug report in the Eclipse repository contains several pieces of information.

- **Bug#** - id for the bug.
- **Product** - product for which the problem is occurring.
- **Component** - Java development tools used to build the product.
- **Status field** - general state of a bug. One state out of [UNCONFIRMED, NEW, ASSIGNED, REOPENED, RESOLVED, VERIFIED, and CLOSED].
- **Resolution field** – One of the fields [FIXED, INVALID, WONTFIX, DUPLICATE, WORKFORME, REMIND, LATER, and NOT_ECLIPSE].
- **Severity** - indicates the impact of a bug on the system. Table 3 defines the classes of severity used in the case study.
- **Priority(n)** - importance at which bug(n) should be treated. The available priorities range from P1 (most important) to P5 (least important).
- **Platform** - hardware platform against which the bug was reported.
- **Operating** - operating system against which the bug was reported.
- **Assigned to** - person in charge of resolving the bug.
- **Target milestone** - milestone the developers planning to fix the bug.
- **Description** - textual description of the bug.
- **Opened** - date when the bug report was opened.
- **size(n)** - # of added + deleted + changed lines (as obtained through diff) in all files necessary to fix the bug.

For the case study, we consider a pool of D = 20 developers. Their competence profiles DevProfile(d) d = 1…20 are summarized in the Appendix. The competence profiles include K= 12 areas which are related to JDT and different JDT components. Relatively, the highest competence is on JDT. Other developers have specialized skills, e.g., developers 6 and 14 have substantial expertise in debugging.

### Table 3. Types of severity and their description

<table>
<thead>
<tr>
<th>Severity type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocker</td>
<td>Blocks development and/or testing work.</td>
</tr>
<tr>
<td>Critical</td>
<td>Causes crashes, loss of data, severe memory leak.</td>
</tr>
<tr>
<td>Major</td>
<td>Causes major loss of function.</td>
</tr>
<tr>
<td>Minor</td>
<td>Results minor loss of function, or other problem where easy workaround is present.</td>
</tr>
<tr>
<td>Trivial</td>
<td>Creates cosmetic problem like misspelled words or misaligned text.</td>
</tr>
<tr>
<td>Enhancement</td>
<td>Request for enhancement.</td>
</tr>
</tbody>
</table>

The profiles have been determined from retrospective analysis of former bug fixing activities. All the changed files were studied and it was checked what API the code is using (based on the import statements = "problem domain" [21]). Then, the API was mapped to one of the ten categories, e.g., org.eclipse.jdt.* becomes JDT, org.eclipse.jdt.jface.* becomes JFace, etc.

Since we looked only at bugs from the JDT project, most developers have high values for Java related categories. Only few developers are responsible for the Compare feature in JDT (actually only one developer).

In this paper we are considering nine milestones of the Eclipse release 3.1 for the JDT product. We are taking most active 20 developers at that period. We are using the mail ids for the developers. Table 4 gives the total number of bugs in different milestone for the 3.1 release.

Different components for a bug indicate how much effort is required for that particular component in the code of the fixed file. We refer these as bug requested profile. All the profiles are normalized to 1 and sum of all the profiles for a bug would be 1. For example, a value of 0.45 for the JDT competence area (k = 1) means that about 45% of the code of the fixed files had to do something with JDT.

### Table 4. Characterization of the nine milestone projects

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Due date</th>
<th>Duration (days)</th>
<th>Total bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 M1</td>
<td>August 12, 2004</td>
<td>1036</td>
<td>78</td>
</tr>
<tr>
<td>3.1 M2</td>
<td>September 24, 2004</td>
<td>515</td>
<td>144</td>
</tr>
<tr>
<td>3.1 M3</td>
<td>November 5, 2004</td>
<td>545</td>
<td>178</td>
</tr>
<tr>
<td>3.1 M4</td>
<td>December 17, 2004</td>
<td>1161</td>
<td>284</td>
</tr>
<tr>
<td>3.1 M5</td>
<td>February 19th, 2005</td>
<td>990</td>
<td>363</td>
</tr>
<tr>
<td>3.1 M6</td>
<td>April 1st, 2005</td>
<td>545</td>
<td>230</td>
</tr>
<tr>
<td>3.1</td>
<td>June 26, 2005</td>
<td>670</td>
<td>19</td>
</tr>
<tr>
<td>3.1.1</td>
<td>September 27, 2005</td>
<td>160</td>
<td>106</td>
</tr>
<tr>
<td>3.1.2</td>
<td>January 9, 2006</td>
<td>204</td>
<td>39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTK</td>
<td>Language neutral API: the Language Toolkit (LTK).</td>
</tr>
<tr>
<td>JFace</td>
<td>JFace is a UI toolkit with classes for handling many common UI programming tasks. <a href="http://wiki.eclipse.org/index.php/JFace">http://wiki.eclipse.org/index.php/JFace</a></td>
</tr>
<tr>
<td>Others</td>
<td>Some other components like JDI, SVN, Ant.</td>
</tr>
</tbody>
</table>
5.2 Analysis of Results

The empirical results compare the baseline (manual) assignment with the results from the different greedy search algorithms, having varying look-ahead time. More precisely, we analyze three questions stated in section 3.

Answer to the Question 1:

The answer to this question is a significant improvement in the quality of the assignment. The situation is illustrated for Milestone 3.1-M1. Figures 1 to 3 describe the relative changes in the level of competence when comparing the baseline solutions with the solutions obtained from Greedy-1, Greedy-5, and Greedy-30, respectively. We see a clear improvement pattern, and the improvement becomes the more significant, the bigger the look-ahead time is. We note that the impact of the improve assignment becomes the more important the bigger the bug is in terms of its size.

Similar tendencies have been observed for the other milestones. The results are summarized in Figure 4. For all nine milestones, we observe an almost monotonous improvement form increasing the look-ahead time for the respective greedy search. The average improvement for the Greedy-30 application is 16%.

Answer to the Question 2:

Total fitness here is defined as the summation of all the individual fitness values (5). Total fitness depends on the assigned gained from the chosen assignment approach. The total fitness gained from Greedy-X and ad hoc is represented in (9) and (10), respectively.

\[
\begin{align*}
\text{Total Fitness}_X & = \sum_{n=1}^{N} \text{Fitness}(n,d_X(n)) \\
\text{Total Fitness}_{\text{Adhoc}} & = \sum_{n=1}^{N} \text{Fitness}(n,d_{\text{Adhoc}}(n))
\end{align*}
\]
Based on the different degrees of fitness, we are now able to define our relative improvement in the degree of total fitness:

\[
\text{Fitness Improvement}(X) = \frac{\text{Fitness}_X - \text{Fitness}_{\text{Adhoc}}}{\text{Fitness}_{\text{Adhoc}}}
\]

The detailed answer to question 2 is given again for the Milestone 3.1-M1. In Figure 5 we analyze the level of improvement expressed in the actual fitness value for varying look-ahead times. The relative improvement view (11) is given in Figure 6, where the results for all milestone projects have been summarized. We observed the similar type of repetitive improvement in fitness as well for all the nine milestones. The more the look-ahead time, the more the improvement is.

Figure 4. Relative improvement (in %) for total expertise of the Greedy-X assignments when compared to ad hoc assignment.

Figure 5. Milestone 3.1 M1: Fitness function value received from varying look-ahead time X and application of Greedy-X

Figure 6. Improvement (in %) for total fitness applying Greedy-X when compared to ad hoc assignment
Question 3
The results of the analysis are given in Table 5. For all the nine milestones, we have summarized the accumulated time savings gained from the different solutions of different greedy search. For milestone 3.1 M1, with 5 days look-ahead we can save 2237 hours. We observe a substantial time savings which would allow fixing additional bugs.

<table>
<thead>
<tr>
<th>Milestone</th>
<th>0</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 M1</td>
<td>2050</td>
<td>2046</td>
<td>2237</td>
<td>2266</td>
<td>2281</td>
<td>2273</td>
<td>2294</td>
<td>2295</td>
</tr>
<tr>
<td>3.1 M2</td>
<td>1360</td>
<td>1384</td>
<td>1481</td>
<td>1615</td>
<td>1597</td>
<td>2019</td>
<td>2400</td>
<td>2358</td>
</tr>
<tr>
<td>3.1 M3</td>
<td>3053</td>
<td>3097</td>
<td>3107</td>
<td>3468</td>
<td>3349</td>
<td>3606</td>
<td>3802</td>
<td>3851</td>
</tr>
<tr>
<td>3.1 M4</td>
<td>5840</td>
<td>5916</td>
<td>5938</td>
<td>6282</td>
<td>6377</td>
<td>6685</td>
<td>7089</td>
<td>7034</td>
</tr>
<tr>
<td>3.1 M5</td>
<td>15363</td>
<td>16208</td>
<td>15793</td>
<td>15644</td>
<td>15828</td>
<td>16964</td>
<td>16583</td>
<td>16860</td>
</tr>
<tr>
<td>3.1 M6</td>
<td>1382</td>
<td>1383</td>
<td>1505</td>
<td>1571</td>
<td>1605</td>
<td>2103</td>
<td>2622</td>
<td>3153</td>
</tr>
<tr>
<td>3.1 M7</td>
<td>1176</td>
<td>1176</td>
<td>1185</td>
<td>1185</td>
<td>1220</td>
<td>1222</td>
<td>1222</td>
<td>1222</td>
</tr>
<tr>
<td>3.1 M8</td>
<td>621</td>
<td>645</td>
<td>487</td>
<td>693</td>
<td>1313</td>
<td>2073</td>
<td>2331</td>
<td>2476</td>
</tr>
<tr>
<td>3.1 M9</td>
<td>1838</td>
<td>1839</td>
<td>1843</td>
<td>1878</td>
<td>1880</td>
<td>1923</td>
<td>1928</td>
<td>1929</td>
</tr>
</tbody>
</table>

6. VALIDITY
Even though the results are looking promising, there are several threats to validity of results.
(i) Running nine milestones in the JDT open source environment does not give any justification to claim for external validity. More comprehensive analysis is necessary, covering both other open source data sets as well as exploiting proprietary data sets. This is one of the areas of future research.
(ii) The assumption of normalized vectors of the competence profiles implies that the total sum of all developer’s competences is the same. This is considered to be an approximation of the truth, ignoring individual differences in the total amount of competence. However, the decision about evaluating developers based on their total amount of competence on a rational scale is hard to be done and needs careful data collection and analysis.
(iii) Another uncertainty factor refers to the estimation of effort for bug fixing. Our simplified model takes lines of code as the sole indicator variable, combined with the indirect relationship to productivity when compared to the baseline assignment.
(iv) The whole experiment was done to perform a comparison between the manually generated baseline solution and assignments generated from greedy search. This implied sticking to the precedence relation as defined by between the opening dates of the bugs. However, this is not a limitation given from the real-world, as changing the sequence of bug fixing might have an even bigger impact on the final results.
(v) Another basic assumption of the study is that the number of defects and the defects themselves are known in advance. This assumption was valid in order to compare the manual with the greedy optimized assignments.
(vi) Determining the competence profiles of developers is important as well as extremely difficult. Ignoring the different competence priorities would be even more dangerous. What we suggest is to make a first step into this direction, knowing that evaluating the competence profile just quantitatively from what has been done in the past is again a simplification of reality.

Even though there are a number of acknowledged facts to limit validity of the results, we consider that the main value of the study is having proposed a method for which it was initially proven that manual decision-making is hardly to be recommended in the complex situation encountered for deciding “Who should fix that bug”. In that sense, we consider the contribution of the paper another instance of arguing towards “Limiting the danger of intuitive decision-making” as proposed by Pfleeger [20].

7. SUMMARY AND OUTLOOK
Continuing existing research related to the questions “Who should fix this bug” and “How long does it take to fix this bug”, we have applied rigorous techniques to optimize the assignment of developers to bugs. This was done by determining their profile of expertise from retrospective analysis of former bug fixing activities, and trying to match the requested and available profiles as good as possible.

The results are promising, at least in comparison with the baseline ad hoc assignment. Better assignments could be demonstrated for nine open source Eclipse projects. However, further research is still needed. Firstly, numerous other heuristics are available (as surveyed in [19]) which potentially are candidates to achieve further improvements, better than Greedy-X. A more comprehensive comparative analysis would allow to better judge the degree of optimality achieved by the different heuristics.
Secondly, our method is relying on some assumption which can be tried to be relaxed. As a trade-off, the problem gets even more complex, for example when allowing a developer to switch from fixing one bug to another (small) one. Thirdly, the availability of all the necessary data is critical for the method. Further investigation is needed how the reliability of the profile data can be improved. Finally, the investigations are planned to be extended to apply the optimized assignment strategies to other data sets, open source ones and proprietary ones.

Acknowledgement

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC Discovery Grant # 250343-07).

8. REFERENCES


9. APPENDIX

Developer’s expertise profile.

<table>
<thead>
<tr>
<th>Developer ID</th>
<th>JDT</th>
<th>JFace</th>
<th>Core</th>
<th>SWT</th>
<th>UI</th>
<th>Debug</th>
<th>LTK</th>
<th>Text</th>
<th>JDI</th>
<th>Search</th>
<th>Compare</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.34</td>
<td>0.14</td>
<td>0.13</td>
<td>0.09</td>
<td>0.11</td>
<td>0.17</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.61</td>
<td>0.09</td>
<td>0.11</td>
<td>0.04</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.08</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>0.88</td>
<td>0.01</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0.70</td>
<td>0.07</td>
<td>0.07</td>
<td>0.05</td>
<td>0.03</td>
<td>0.00</td>
<td>0.05</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0.82</td>
<td>0.01</td>
<td>0.17</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0.33</td>
<td>0.12</td>
<td>0.13</td>
<td>0.07</td>
<td>0.10</td>
<td>0.25</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>0.38</td>
<td>0.25</td>
<td>0.07</td>
<td>0.15</td>
<td>0.12</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>8</td>
<td>0.39</td>
<td>0.30</td>
<td>0.08</td>
<td>0.12</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>0.55</td>
<td>0.10</td>
<td>0.13</td>
<td>0.08</td>
<td>0.05</td>
<td>0.00</td>
<td>0.07</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>0.94</td>
<td>0.01</td>
<td>0.04</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>11</td>
<td>0.67</td>
<td>0.07</td>
<td>0.08</td>
<td>0.06</td>
<td>0.04</td>
<td>0.01</td>
<td>0.06</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.92</td>
<td>0.00</td>
<td>0.07</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>13</td>
<td>0.42</td>
<td>0.15</td>
<td>0.10</td>
<td>0.11</td>
<td>0.09</td>
<td>0.11</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>14</td>
<td>0.28</td>
<td>0.14</td>
<td>0.13</td>
<td>0.09</td>
<td>0.09</td>
<td>0.25</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>15</td>
<td>0.82</td>
<td>0.01</td>
<td>0.12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.03</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>16</td>
<td>0.40</td>
<td>0.19</td>
<td>0.12</td>
<td>0.18</td>
<td>0.09</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>17</td>
<td>0.88</td>
<td>0.02</td>
<td>0.09</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>18</td>
<td>0.85</td>
<td>0.02</td>
<td>0.10</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>19</td>
<td>0.51</td>
<td>0.07</td>
<td>0.11</td>
<td>0.05</td>
<td>0.06</td>
<td>0.17</td>
<td>0.00</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>20</td>
<td>0.56</td>
<td>0.13</td>
<td>0.10</td>
<td>0.10</td>
<td>0.08</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>