This is the author's version of the manuscript of the paper that has been accepted for publication and includes any author-incorporated changes suggested through the peer review process. This version does not include however other publisher value added contributions such as formatting and pagination.

The final version will be available online at www.sciencedirect.com

First North American Search Based Software Engineering Symposium

An Empirical Investigation of a Genetic Algorithm for Developer's Assignment to Bugs

Md. Mainur Rahman[a, b], Muhammad Rezaul Karim[a, *], Guenther Ruhe[a], Vahid Garousi[c], Thomas Zimmermann[d]

[a]*Software Engineering Decision Support Laboratory, University of Calgary, Calgary, Alberta, Canada*
[b]*MNP LLP, Calgary, Alberta, Canada*
[c]*System and Software Quality Engineering Research Group, Atilim University, Ankara, Turkey*
[d]*Microsoft Research, One Microsoft Way, Redmond, WA 98052 USA*

Abstract

Software development teams consist of developers with varying expertise and levels of productivity. With reported productivity variation of up to 1:20, the quality of assignment of developers to tasks can have a huge impact on project performance. Developers are characterized according to a defined core set of technical competence areas. The objective is to find a feasible assignment, which minimizes the total time needed to fix all given bugs.
In this paper, the modeling of the developer's assignment to bugs is given. Subsequently, a genetic algorithm called GA@DAB (Genetic Algorithm for Developer's Assignment to Bugs) is proposed and empirically evaluated. The performance of GA@DAB was evaluated for 2040 bugs of 19 open-source milestone projects from the Eclipse platform. As part of that, a comparative analysis was done with a previously developed approach using K-Greedy search. Our results and analysis shows that GA@DAB performs statistically significantly better than K-greedy search in 17 out of 19 projects. Overall, the results support the argument of the applicability of customized genetic search techniques in the context of developer-to-bug assignments.

© 2012 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of Global Science and Technology Forum Pte Ltd

Keywords: Search-based software engineering; bug fixing; genetic algorithms; empirical evaluation; open source projects

---

* Corresponding author. Tel.: +1-403-808-7112.
E-mail address: mrkarim@ucalgary.ca

1. Introduction

"Who should fix this bug?" is a frequent decision problem to be solved in software development and maintenance. It is well known that skill level and productivity may significantly vary among developers. Each individual might have different level of expertise and productivity when performing different types of tasks. Literature suggests that productivity among workforce can vary in the range between 1 to 20 (Boehm et al., 2000; Sackman et al., 1968; Valett and McGarry, 1989). Consequently, instead of considering all developers equally, we will look at their competency and productivity and aim to assign bugs to developers in accordance to their strengths.

Looking at scenarios of having hundreds (if not thousands) of bugs at hand, a proper assignment of developers to bugs becomes a tremendous challenge. Many bugs are assigned to some developers and those developers pass them on to others, as they do not think they were the right people to fix those bugs, a phenomenon referred to as bug tossing (Jeong et al., 2009). Jeong et al. (2009) have studied the assignment and reassignment process for 450,000 bug reports from Eclipse (Eclipse Bugs, 2014) and Mozilla. It was found that 37%-44% of bugs have been re-assigned at least once to another developer, which indicates the inefficiency of ad hoc bug fixation tasks applied.

In order to find better assignments of developers to bugs, we propose the application of genetic optimization algorithms. These algorithms have proven efficient in numerous software engineering problems (Penta, 2011). For its application, the original assignment problem is modeled as an assignment problem with special constraints. The objective is to minimize time needed to fix a given set of bugs. Based on that, a customized genetic algorithm (GA) called GA@DAB (Genetic Algorithm for Developer's Assignment to Bugs) is proposed.

The main contributions of the paper are:

- Formulation of bug fixing activities as a special type of assignment problem that allows the application of genetic search optimization algorithms.
- Design of a customized genetic algorithm GA@DAB to solve the bug-developer assignment problem.
- Calibration of different GA@DAB parameters (cross-over rate, mutation rate, population size etc.).
- Evaluation of the resulting performance for a set of 19 open source Eclipse milestone projects.

The remainder of this paper is structured as follows. Related work is analyzed in Section 2. In Section 3, we give a formal problem description. The customized GA is presented in Section 4. Section 5 is devoted to a comprehensive empirical analysis of the solution methods. We report the limitations and threats to validity of results in Section 6. We finally conclude in Section 7 by providing an outlook for future research.

2. Related Work

2.1. Bug Assignment -Techniques, Processes and Practices

To improve bug triaging, that is, the process of deciding which bug reports get fixed and by whom, previous research proposed techniques to semi-automatically assign developers to bug reports (Anvik and Murphy, 2011; Kagdi and Poshyvanyk, 2009; Matter et al., 2009). Anvik and Murphy (2011) used machine learning techniques to build classifiers based on previous bug report assignments to suggest a small number of developers suitable to resolve a bug report.

Matter et al. (2009) used a vocabulary-based model of developers to assign them to bug reports. The vocabulary of developers was learned from their source code contributions and compared to the vocabulary of bug reports. Kagdi and Poshyvanyk (2009) applied a two-step strategy. First, they used information retrieval to identify affected source code elements. Second, expertise recommendation was applied to propose

developers who can address a textual change request. As an instance this approach was comprehensively evaluated with high accuracies of 80% for bug reports in KOffice (Kagdi et al., 2012). This strategy is similar to McDonald and Ackerman (2000) and Mockus and Herbsleb (2002) who both used the “Line 10 Rule” to determine the developer with the most recent expertise for the source file.

One of the most recent results is described in (Bhattacharya et al., 2012). Therein, the authors employ a comprehensive set of machine learning tools and a probabilistic graph-based model (bug tossing graphs) that lead to highly accurate predictions. The authors validated their approach on Mozilla and Eclipse projects, covering 856,259 bug reports and 21 cumulative years of development. They also demonstrated that their techniques can achieve up to 86.09% prediction accuracy in bug assignment and reduce tossing path lengths.

Weiss et al. (2007) built a model to estimate the time it will take to fix a new bug. This model takes the experience from the previous bug fixes into account and predicts the time it would take to fix a new bug report. They used a nearest neighbor algorithm with a vector-based approach to measure text similarity and evaluated the approach on 567 bug reports from the JBoss (JBoss Community, 2014) data set. Other studies investigated the bug survival time, that is the time from the introduction of a bug until it is fixed (e.g., Canfora et al., 2011).

K-Greedy approach is a recently developed approach for developer assignment to bugs (Rahman et al., 2009; Rahman et al., 2010). Given a bug’s arrival (filing) time, the K-Greedy algorithm considers all the developers who are available at and after that time (until a time window called the *look-ahead time (LAT)*) (Rahman et al., 2010). Look-ahead time can be *K* days or hours ahead of a bug selection time. For the bug in consideration, this algorithm finds the distance between the required competencies for solving a bug and the available competencies of a developer. The developer with the lowest distance is assigned the bug. This algorithm is greedy in this sense. The performance of this approach was evaluated with nine milestones of the open source Eclipse JDT project and two industrial case study projects. In this work, the authors concluded that substantial savings in terms of time can be achieved, if manual staffing plans are replaced with optimized staffing policies.

2.2. Staffing in Software Development

Staffing is one of the key responsibilities of a project manager. Assigning developers to bug fixing activities is considered as a form of staffing. Sackman et al. (1968) studied professional programmers with an average of 7 years’ experience. The general finding that ‘‘There are order-of-magnitude differences among programmers’’ has been confirmed by other studies such as (Valett and McGarry, 1989) and (Boehm et al., 2000). Despite its importance, there is little support for this task, leading to unease and misunderstandings among the people involved (Acuna et al., 2006).

While the impact of staffing on project performance is known to be high, the current state of the research on human resource allocation techniques is limited (Otero et al., 2009). Trendowicz and Münch (2009) have concluded that the capabilities of developers and the tools and methods that are used have significant impact on the productivity of software development processes. Kapur et al. (2008) considered the productivity and availability of the individual developers while assigning tasks to them for product releases.

Penta et al. (2011) combined search-based optimization techniques with a queuing simulation model to determine staffing plans that minimize the completion time and reduce schedule fragmentation.. Park et al. (2014) also employed search-based optimization technique on the human resource allocation problem in software projects. They designed a genetic algorithm that take the practical issues of human resource allocation into account like generate short project plans, minimize multi-tasking time of developers, assign developers to relevant tasks, balance allocation of developers in terms of having different staff levels assigned together.

2.3. Parameter and Performance Tuning of Genetic Algorithms

There are many studies in operations research (for instance in (Smit and Eiben, 2010)) which empirically studies the performance and parameter tuning (e.g., crossover ratio) of GAs and other Evolutionary Optimization Techniques (EOT). For the area of software engineering, a comprehensive study done by Arcuri and Fraser (2011) is one of the latest contributions on this topic. Their study confirms that "tuning does have a critical impact on algorithmic performance, and over-fitting of parameter tuning is a direct threat to external validity of empirical analyses in SBSE." (Arcuri and Fraser, 2011).

3. Modeling and Problem Statement

3.1. Bugs

In this study, we consider the process of fixing a given set of bugs. The set of bugs called *BUGS* is considered static for this problem formulation. An individual bug from this set is denoted by *bug(i)*. We assume to have N bugs in consideration, i.e., *BUGS = {bug(1) ... bug(N)}*. Each bug requires a set of requested competencies as explained in the subsequent subsection.

3.2. Requested Competencies for Fixing Bugs

Competencies can be defined as measurable capabilities required in performing tasks in the context of some concrete work situations (Dzinkowski, 2000). Competency here means a combination of knowledge, skills and experience in the area under consideration. From analyzing similar bugs of former projects, these competencies are assumed to be relevant for fixing bugs. These competencies can be related to the usage of certain languages, techniques or tools.

In the context of a specific project, a number *K* of relevant competencies is determined upfront by the project manager. In the same way as for the set of bugs, we considered these competencies to be static. The extension to dynamic scenarios is discussed in Section 7 as future work.

3.3. Developers and their Productivity

For fixing bugs, we assume a set (pool) of developers called *DEV*. The competencies of developers are related to a combination of knowledge and experience as defined by the context of the project and/or organization.

We assume that the competency of a developer in some specific area is directly proportional to the productivity of the developer. Even though this is a simplification of the reality, it helps to initiate the more systematic planning and assignment process. The underlying assumption here also is that the productivity level is continuously updated over time.

Productivity here is expressed relative to an average productivity (defined to be equal to 1). For assumed average productivity of a developer, the effort value (in person days) of an activity equals the duration (in days). The productivity of developer *d* related to competence area *k* is denoted by *prod(d,k)* and is defined relative to average productivity.

The estimated level of productivity of a developer is determined based on existing work experience. To estimate productivity of developers, there are also many approaches known; for example, to mine the history and magnitude of past changes by developers from version control systems (Schröter et al., 2006).

The main operation to achieve shorter total time for fixing a given set of bugs is to look at the assignment of bugs to developers in accordance to the requested (by the bugs) and available (by the developers) competences.

3.4. Assignment of Developers to Bugs

In this paper we consider what is called *batch assignment*. Batch assignment mostly happens in large-scale projects. At the beginning of a (maintenance) release, the project manager decides which bugs are to be fixed and assigns them to developers.

The set of all possible assignments of bugs to developers is denoted by *ASSGN*. A specific assignment called *assgn* is formulated as a vector. The ith component of this vector corresponds to *bug(i)*. The value of this component describes the developer who (according to this assignment) is handling this bug. Typically, a developer is assumed to fix several bugs. However, the assumption is that developers are not working on more than one bug simultaneously.

3.5. Estimation of Bug Fixing effort

Optimizing the assignment of developers to bug-fixing tasks needs the estimation of the required effort. *Effort(i)* denotes the total amount of effort required for fixing *bug(i)*. The amount of effort requesting the specific competence *k* is denoted by *effort(i,k).*

As for any effort estimation, estimation of bug fixing effort is known to be inherently difficult. Although manual estimation is always an option, techniques for automated or at least semi-automated effort estimation are considered more efficient. The principle of estimation by analogy (EBA) has been used in (Weiss et al., 2007) to search for similar, earlier bug reports and using their average time as a prediction for the new one.

3.6. Completion Time as the Objective Function

For a given assignment *assgn*, the *Competency-area Completion Time* denoted by *CCT(assgn,i,k)* is defined as the time needed for fixing the competency portion *k* of the total bug fixing effort if performed by the specific developer *d* specified by the ith component of *assgn*:

$$\text{CCT}(i,assgn,k) = effort\,(i,k)/prod(d{=}assgn(i),k) \quad (1)$$

The Completion Time *CT(i,assgn*) of fixing *bug(i)* by developer *d* taken from the ith component of *assign* is defined in equation (2) as the summation of all completion times *CCT(i,assgn,k)* for all *K* competencies:

$$CT(i,assgn) = \sum_{k=1\ldots K} CCT(i,assgn,k) \quad (2)$$

In the bug assignment problem considered in this paper, the objective function is to minimize the make-span for fixing all *N* given bugs. The *make-span* is determined by the developer’s time spend for fixing all the bugs assigned to her. For developer *d*, the total Developer Completion Time *DCT(d,assgn)* according to assignment *assgn* is defined by equation (3).

$$DCT(assgn,d) = \sum_{i:\ assign(i)=d} CT(i,assgn) \quad (3)$$

The total completion time *TCT(assgn)* for fixing all bugs according to *assgn* is given by Equation 4.

$$TCT(assgn) = Max\ \{DCT(assgn,d)\ among\ all\ developers\ d\ from\ DEV\} \quad (4)$$

Finally, the *Developer's Assignment to Bugs* problem *DAB* is to find an assignment which minimizes the maximum duration of *TCT(assgn)* among all assignments.

$$DAB:\ Min\ \{TCT(assgn):\ assgn\ from\ ASSIGN\} \quad (5)$$

4. Solution Method based on Genetic Search

4.1. Overview

In order to find the most suitable developers for assigning them to bugs, we have designed a customized GA called GA@DAB. By GA, problems are solved by a process that mimics natural evolution in looking for a best (fittest) solution (survivor). The detailed steps of the design and implementation of GA@DAB are discussed in the following sections.

4.2. Design of the GA

4.2.1. Chromosome Structure

A chromosome is a representation of an individual solution for a specific problem. For the problem under consideration, a chromosome represents the assignments of developers to bugs. Each gene in the chromosome corresponds to an assigned developer to a bug. Consequently, the size of a chromosome is defined by the number of bugs. Fig. 1 shows a chromosome representation with a simple problem.

4.2.2. Fitness Function

The fitness function provides a quantitative measure of the fitness of the chromosome. The fittest chromosomes are selected for reproduction and are mixed using applicable techniques with a hope to produce a new and better generation. The fitness of a chromosome representing a specific assignment *assgn* is defined as:

$$Fitness(assgn) = TCT(assgn) \quad (6)$$

4.2.3. GA Model and Selection Operator

In this work, we use generational GA. In generational GA, genetic operators are used to create new offspring from the members of an old population and the whole new population replaces the old population (Goldberg, 1989). There are several existing generic parent selection algorithms in GA: *roulette wheel selection, tournament selection* etc. (Goldberg, 1989).

In roulette wheel selection, each candidate assignment represents a pocket on the wheel. The size of the pocket is proportional to the fitness of the assignment (Goldberg, 1989; Haupt and Haupt, 1998). In our implementation, the raw fitness F_r (see Eq. 6) of each individual is converted to scaled fitness F_s, before using in the roulette wheel selection, where $F_s = 1/(1 + F_r)$. This scaling operation (Goldberg, 1989) was performed to overcome two limitations of the roulette wheel selection. First, it avoids repeated selection of the supersolution (solution with high raw fitness). Second, in the later part of evolution, when almost all the individuals have almost similar fitness, it avoids random selection of individuals. Random selection is avoided as a small difference in fitness has significant effect in the selection process.

Fig. 1. Chromosome representation for a assignment problem with 10 bugs and 20 available developers. Each chromosome has 10 genes, each of them representing a single bug. Each gene value represents a developer. In this chromosome, developer 10 and developer 5 have been assigned 3 and 2 bugs, respectively.

In tournament selection, to select an individual, a tournament is run among few individuals chosen at random. The number of individuals in a tournament is given by the tournament size parameter. Unless we compared the performance of both selection algorithms, we used roulette wheel selection in our experiments.

4.2.4. Crossover and Mutation Operator

In GA's, crossover is a genetic operator used to vary the structure of chromosomes from one generation to the other. A crossover operator aims to interchange the information and genes between chromosomes. Therefore, a crossover operator combines two or more parents to reproduce new children. We chose standard one point crossover operator (Goldberg, 1989). In this crossover, a single crossover point is selected on both parents' chromosome. The gene values after the chosen point are exchanged between the two parent chromosomes. The mutation operator that we use, with certain probability, alters value of each gene with a randomly chosen integer value. The new integer also represents a developer.

4.2.5. Constraint handling

In our implementation, we assume that the number of developers assigned to a particular bug cannot exceed a certain threshold value. Because of the nature of crossover or mutation operations, newly created individuals might violate this constraint. After any genetic operation, if this constraint is violated for a new individual, repair operation is performed. First, we identify the developers for which this constraint is violated. Then for each identified developer, a gene value representing this developer is converted to a gene value representing another developer. This conversion is repeated until the number of bugs assigned to that developer is lower than or equal to the threshold.

4.2.6. Termination Criterion

The GA was run for maximum of 500 generations.

5. Empirical Analysis

5.1. Data Sets

In this paper, we have studied 2,040 bugs from 19 milestones of two projects of the Eclipse platform. The Eclipse JDT (Java Development Tools) project provides the tool plug-ins that implements a Java IDE supporting the development of any Java application, including Eclipse plug-ins (Eclipse JDT, 2014). Each of the bugs in the repository of this project has a profile description. For fixing the bugs, developers need to modify certain number of lines of code in different components of a particular product. In order to calculate the estimated effort in this study, we selected only bugs that were opened and closed within a period of 6 months.

The Eclipse platform is a universal tool platform that provides the core frameworks and services upon which all Eclipse plug-in extensions are developed (Eclipse Platform, 2014). Version 3.0 of the Platform project had in total 10 milestones (Eclipse Bugs, 2014). We selected all 10 milestones of the Platform project. With the same justification used for the JDT project, we identified the 78 most-active developers who had

been assigned to fix between 79%-97% of all the bugs in the actual manual assignments for the Platform project.

To estimate productivity of developers, we mined their past code changes from the version control repository. We mapped these files to competencies. Like before, we used the problem domain for the JDT dataset and the file location for the Platform dataset. For each competency, we counted how many file changes a developer had made and subsequently applied Analytic Hierarchy Process (AHP) (Saaty, 1980) to estimate the relative productivity of the 20 developers of the JDT project and 78 developers for the Platform project. All the pre-processed data is available online (SEDS, 2014) for replication purposes.

5.2. Research Questions

Motivated by existing deficits outlines in Section 2, four research questions were formulated related to the proposed genetic search approach, GA@DAB:

- RQ 1: How different combinations of crossover and mutation rates impact performance of GA@DAB?
- RQ 2: How does population size impact the performance?
- RQ 3: How does the choice of a selection algorithm affects the performance?
- RQ 4: How does GA@DAB-results compare to K-Greedy search?

In this paper, our primary goal is to measure the performance of the GA@DAB by answering the RQ 4. Before we can answer RQ 4, first, we need to tune various parameters of GA. The first three research questions were designed keeping the parameter tuning in mind.

5.2.1. RQ1: Impact of Crossover and Mutation Rate

In order to calibrate crossover and mutation rates, we considered three population sizes (100, 250, 500) and varied the value of both rates between 5% and 90% with increments of 5% and applied GA@DAB for all possible 1083 (=3×19×19) combinations. To account for the GA's randomness effect, we ran the GA@DAB 50 times for each configuration and then calculated the average of all the outputs. Next, we analyze the impact of crossover and mutation using two representative projects: *PlatformMilestoneM2* and *JDTMilestone3.1.1.*

Fig. 2 shows the average completion time across 50 runs for the PlatformMilestoneM2 project with three different population sizes, varying crossover rates and fixed mutation rate of 1%. It is evident from the figure that, with all population sizes, for a particular crossover setting, there is a high variance in the developers completion time across all runs. For this reason, we cannot conclude which crossover setting is better for a particular population size. Mutation rate, on the other hand, showed bit different effect that we usually notice while applying GA on other problems.

Fig. 3 shows that low mutation rates ended up with higher developer completion time (worse results) than high mutation rates, specially for the high crossover rates (70% or more). Particularly, when we increased the mutation rate to 0.5 or 0.9, crossover did not show any effect on the completion time. However, when we randomly picked 15 bugs for this project milestone, instead of all 51 bugs, we noticed that completion time increased as we increased the mutation rate. This was true for any crossover settings. Assignment of 78 developers to 15 bugs converts the problem into a harder assignment problem. Usually high mutation destructs the discovered good solutions and has negative impact on the performance. We also noticed that crossover operator showed similar behavior even after changing the number of bugs.

For the JDTMilestone3.1.1 project, we had to assign 20 developers among 78 bugs. This is also a hard assignment problem. In this case, we did not observe any significant differences with varying crossover rates. This is because of the high variances shown across different runs. For this JDT project milestone, as the mutation rate was increased, completion time also increased (see Fig. 4). This was true for any crossover

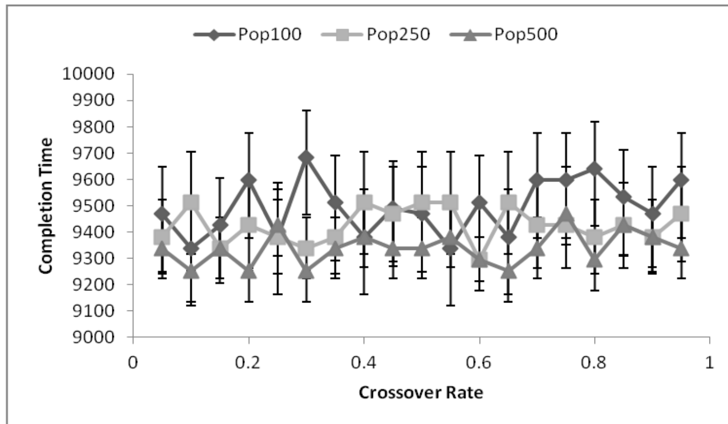

Fig. 2. Average developer completion time (hours) for PlatformMilestoneM2 with varying crossover rates and fixed mutation rate of 1%. The crossover effect is shown for three different population sizes of 100, 250 and 500. The error bars represent 95% confidence interval of errors.

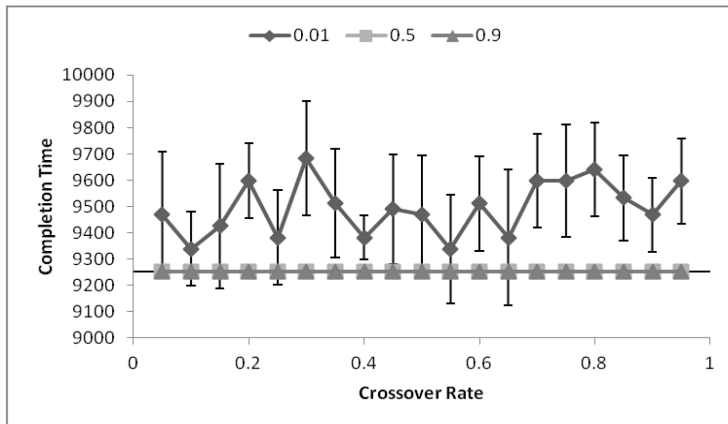

Fig. 3. Average developer completion time (hours) for the project PlatformMilestoneM2 with varying crossover rate and fixed population size of 100. The number of bugs considered is 51. The crossover effect is shown for three different mutation rates: 0.01 (1%), 0.5 (50%), 0.9 (90%). The error bars represent 95% confidence interval of errors. If the developer completion time for a particular crossover rate is same across all 50 runs (zero error), error bar is not shown.

settings. Even though we showed the plots for the population size of 100, we observed the same mutation behavior for the other population sizes.

From our analysis, we could not conclude which crossover rate is the best for a particular population size, as with each crossover setting, GA showed high variances. However, for mutation rate, our analysis suggests the use of a low mutation rate for the hard assignment problems, as commonly used in GA experiments. In our next experiments, unless otherwise stated, we use fairly standard crossover rate of 90% and mutation rate of 1%.

5.2.2. RQ2: Impact of Population Size

For this research question, we investigated the impact of three population sizes (100, 250 and 500) on (i) the GA's fitness function value (developer completion time) and on (ii) execution time. From Fig. 5, we

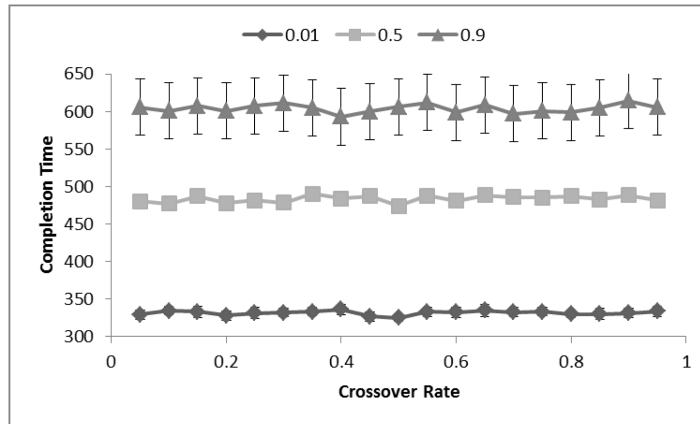

Fig. 4. Average developer completion time (hours) for JDTMilestone3.1.1 with varying crossover rates and fixed population size of 100. The crossover effect is shown for three different mutation rates: 0.01 (1%), 0.5 (50%), 0.9 (90%). The error bars represent 95% confidence interval of errors. If the developer completion time for a particular crossover rate is same across all 50 runs (zero error), error bar is not shown.

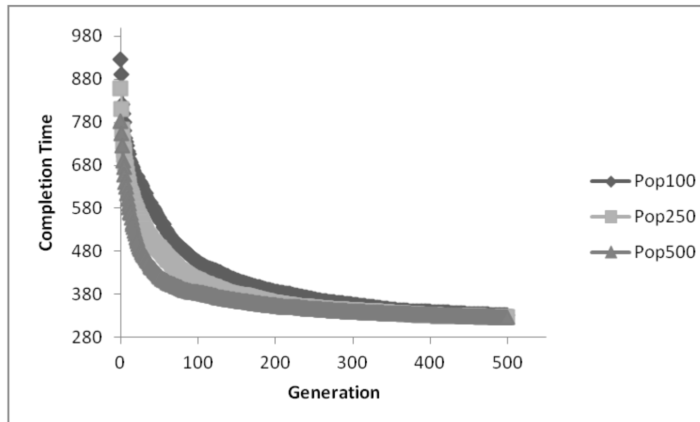

Fig. 5. Average developer completion time (hours) for JDTMilestone3.1.1 project for varying population sizes with fixed crossover rate of 90% and fixed mutation rate of 1%. To avoid clutter, error bars are not shown for all population sizes.

observe that, for the JDTMilestone3.1.1 project, in the early generations, larger population sizes (beyond 100) achieved better results than smaller population sizes, in terms of developer completion time. In this case, a fairly standard crossover (90%) and mutation rate (1%) were used. However, in the latter half of the generations, with any population size, GA@DAB achieved almost same level of developer completion time. In addition to that, with smaller population size of 100, GA@DAB also took very small amount of execution time.

Fig. 6 shows the execution time for the JDTMilestone3.1.1 project for varying crossover rates. It is evident from this figure that the lower the population size, the lower the execution time. From our further analysis, we observed the same trend for difficult milestones of both JDT and Platform projects, regardless of any crossover and mutation settings.

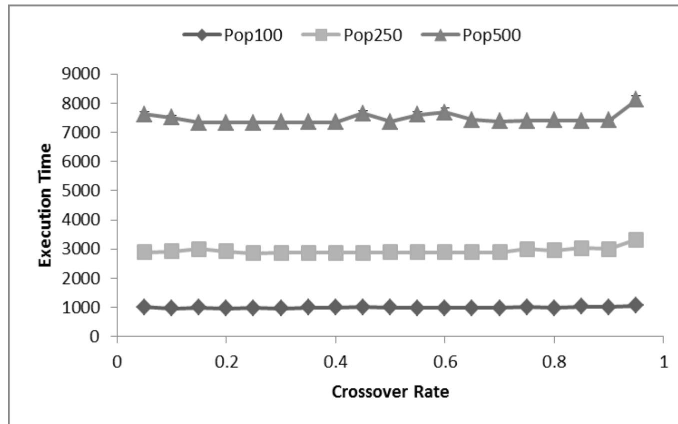

Fig. 6. Average GA execution time (ms) for JDTMilestone3.1.1 project with different population sizes. The results are shown for varying crossover rates and fixed mutation rate of 1%. As the GA execution time for each crossover rate is same across all 50 runs (zero error), error bars are not shown.

5.2.3. RQ3: Impact of Selection Algorithm

For this research question, we investigated two selection algorithms: roulette wheel selection and tournament selection. For the latter algorithm, we considered tournament sizes of 2 and 5. From our experiments, we noticed that GA@DAB had almost same level of performances, in terms of developer completion time, with the both selection algorithms. Fig. 7 shows the results for the PlatformMilestoneM2 project. It is evident that GA@DAB shows high variances across different runs for any crossover rates, no matter which selection algorithm is used. In addition to that, increasing the tournament size even did not improve the performance of GA@DAB.

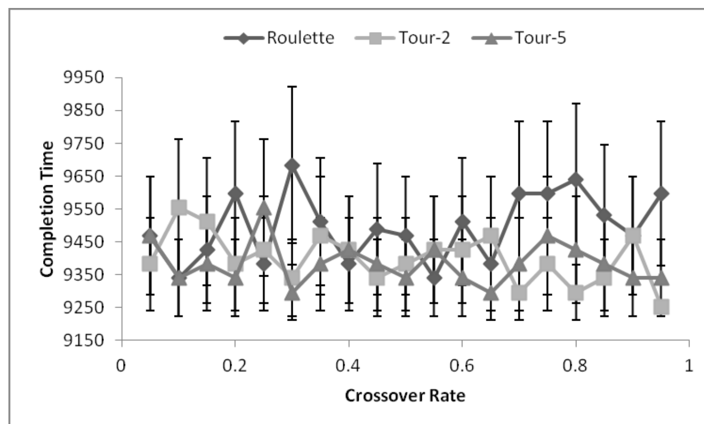

Fig. 7. The impact of selection algorithms on the PlatformMilestoneM2 project. Developer completion time (hours) are shown for varying crossover rates, fixed mutation rate of 1% and population size of 100. The error bars represent 95% confidence interval of errors.

5.2.4. RQ4: Comparison of GA@DAB and K-Greedy

For RQ4, we studied the quality of results gained from application of GA@DAB in comparison to the application of previously-developed K-Greedy search approach (Rahman et al., 2009; Rahman et al., 2010). For the Eclipse JDT and Platform projects, we considered all the 20 and 78 developers, respectively, as being available for all the available bugs. We varied the LAT in terms of percentage of total milestone duration (in hours), to ensure we get the best results from the K-Greedy approach. We varied the LAT up to 80% of total duration.

For GA@DAB, for all the milestones, we used the parameter configurations suggested in Section 5.2.1, 5.2.2 and 5.2.3. The GA@DAB was executed 50 times for all the milestones and the average result was taken. In each run, the population size considered was 100, with crossover and mutation rates of 90% and 1% respectively. We used roulette wheel selection as the selection mechanism.

To compare GA@DAB and K-Greedy approach, first, using *Mann-Whitney U* test (Mann and Whitney, 1947), we determine the statistical significance of the differences in developer completion time of the two approaches. Whenever the p-value of the Mann-Whitney U test is less than 0.01, we conclude that there is a significant difference in performances. However, Mann-Whitney U test result is not enough to decide which approach is better for a particular project. This test can indicate whether two approaches are different but it cannot quantify the performance difference. *Vargha-Delaney A* measure (Vargha and Delaney, 2000) can be used to complement the *Mann-Whitney U* test while comparing two approaches. For our comparison purpose, the *A* measure provided us with the probability that one approach will achieve lower developer completion time than another approach. In our experiments, when the *A* measure value is above 0.5, GA@DAB outperforms the K-Greedy approach. When the *A* measure is 0.5, the two approaches are equal. Otherwise, GA@DAB performs worse than K-Greedy.

Table 1 shows the comparison of results between GA@DAB and the K-Greedy approach for the JDT (1 to 9) and Platform (10 to 19) projects. We observe that, in 17 of the 19 cases, the assignments suggested by GA @DAB are better than those of K-Greedy, in terms of developer completion time. In all these cases, A-measure value is 1.00, which indicates that with high probability GA@DAB outperforms K-Greedy. For the two milestones, *JDTMilestoneM4* and *JDTMilestoneM5*, we did not observe any statistically significant difference between the two approaches. K-greedy approach considers bugs one after another and tries to select a developer that seems best in a particular context considering the look ahead time. When we have a batch of bugs, K-greedy approach might fail. In the first few selections, it might not select developers leading towards globally optimal solution for the whole batch.

Table 1. Comparison of GA@DAB and K-greedy in terms of developer completion time (hours). The number before '±' represents the mean across 50 runs, while the number after '±' represent the standard deviation. The lower the mean for a project, the better. '*'indicates that there is statistically significant difference in the results for the two approaches, while '-' indicates that there is no statistically significant difference. The number after the '*' represents the *Vargha-Delaney A* measure value.

Project Milestone	*GA@DAB*	*K-Greedy*	*Stat-test*
JDTMilestone3.1	57.7 ± 0.28	80	*(1.00)
JDTMilestone3.1.1	331.75 ± 23.58	774	*(1.00)
JDTMilestone3.1.2	135.54 ± 0.88	390	*(1.00)
JDTMilestoneM1	246.22 ± 1.53	451	*(1.00)
JDTMilestoneM2	270.37 ± 8.88	569	*(1.00)
JDTMilestoneM3	511.29 ± 57.11	971	*(1.00)
JDTMilestoneM4	785.869 ± 102.97	805	-

Project Milestone	*GA@DAB*	*K-Greedy*	*Stat-test*
JDTMilestoneM5	835.81 ± 117.21	792	-
JDTMilestoneM6	433.2 ± 31.07	905	*(1.00)
PlatformMilestone3.0	56143.48 ± 4193.48	110754	*(1.00)
PlatformMilestone3.1	71549.71 ± 3696.93	145371	*(1.00)
PlatformMilestoneM2	9338.84 ± 422.38	33323	*(1.00)
PlatformMilestoneM3	125589.18 ± 36656.64	412607	*(1.00)
PlatformMilestoneM4	205221.5 ± 14567.06	1333393	*(1.00)
PlatformMilestoneM5	137642.63 ± 3337	466317	*(1.00)
PlatformMilestoneM6	52483.61 ± 1477.78	140718	*(1.00)
PlatformMilestoneM7	180774.12 ± 37619.08	331201	*(1.00)
PlatformMilestoneM8	131976.7 ± 34347.83	508473	*(1.00)
PlatformMilestoneM9	155926.24 ± 17419.0	404522	*(1.00)

6. Applicability, Limitations and Threats to Validity

6.1. Applicability and Limitations

While the results of the study are promising and support the effectiveness and efficiency of GA@DAB, the approach relies on several assumptions, which can be considered as limitations. Future work should focus on relaxing the following assumptions.

1. We assume that the sets of bug reports, developers, and competencies are fixed.
2. We assume that competency of developers in specific areas are directly proportional to productivity of developers.
3. Developers are assumed to be working on an assigned bug report until it is fixed.
4. The approach largely depends on availability and accuracy of effort estimates for individual bug reports.
5. The approach also depends on availability of productivity estimates for developers. Although this is difficult to estimate, the application of AHP can provide some confidence in the estimates (in case the number of developers is small to medium).
6. We currently ignore factors such as the complexity of bugs. Models considering the complexity, types, and nature of bug reports could lead to better results.

We have identified the following threats to the validity of this study:

Construct validity. To evaluate the effectiveness and efficiency of the proposed GA@DAB approach, we have used data mined from software repositories with standard techniques. This raises potential threats with respect to how accurately the development process was captured, e.g., does the calculated effort correspond to the actual effort, etc.

External validity. Drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of our study generalize beyond the specific environment in which it was conducted, in this case the JDT and Platform projects of Eclipse. While the specific crossover and mutation rates will likely be different, the approach that we have used to calibrate these parameters generalizes to other projects.

As with any research, replication studies are essential for strengthening external validity. To facilitate replication of this research, we made the dataset and prototype implementation of GA@DAB publicly available at (SEDS, 2014).

7. Conclusion

This research is considered part of the overall effort to qualify decision-making in software project management. By conducting extensive analysis and comparing assignments achieved with various techniques, substantial benefits in terms of time savings can be achieved that can allow project managers to allocate more bugs in the iterations.

Our methods are relying on some assumptions, which can be partially relaxed in future work. For example, the assumptions that each bug is fixed by exactly one developer can be relaxed and instead be assigned to a small group of developers. Additionally, for assigning the bugs we have considered some factors while other factors might be considered. Time windows of availability and dynamically changing sets of bugs are extensions for which the same genetic optimization approach could be applied. Finally, by re-running the optimization in shorter cycles (e.g., weekly) and with updated sets of bugs, the solution method can be extended to more dynamic settings with permanently updated sets of bugs. We have considered this process as a single objective optimization problem. However, it can be considered as a multiple-objective optimization problem with aspects of cost as additional criterion.

References

Acuna, S., Juristo, N., Moreno, A., 2006. Emphasizing human capabilities in software development. IEEE Software, vol. 23, no. 2, pp. 94-101.

Anvik, J., Murphy, G., 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Transactions on Software Engineering and Methodology, vol. 20, no. 3, article 10.

Arcuri, A., Fraser, G., 2011. On parameter tuning in search based software engineering. In: Proc. of the Third International Symposium on Search Based Software Engineering, pp. 33-47.

Basili, V., Shull, F., Lanubile, F., 1999. Building knowledge through families of experiments. IEEE Transaction on Software Engineering,vol. 25, no. 4, pp. 456-473.

Bhattacharya, P., Neamtiu, I., Shelton, C., 2012. Automated, highly-accurate, bug assignment using machine learning and tossing graphs.Journal of Systems and Software, vol. 85, no. 10, pp. 2275-2292.

Boehm, B., Abts, C., Chulani, S., 2000. Software development cost estimation approaches—A survey. Annals of Software Engineering, vol. 10, pp. 177-205.

Bugzilla [Online], 2014. URL http://www.bugzilla.org/ (accessed 30 December 2014).

Canfora, G., Ceccarelli, M., Cerulo, L., Penta,M., 2011. How long does a bug survive? an empirical study. In: Proc. of the 18th Working Conference on Reverse Engineering. pp. 191-200.

Dzinkowski, R., 2000. The measurement and management of intellectual capital: an introduction. Management Accounting, vol. 78, no. 2, pp. 32–36.

Eclipse Bugs [Online], 2014. URL https://bugs.eclipse.org/bugs/ (accessed 30 December 2014).

Eclipse JDT [Online], 2014. URL http://projects.eclipse.org/projects/eclipse.jdt/ (accessed 30 December 2014).

Eclipse Platform [Online], 2014. URL http://projects.eclipse.org/projects/eclipse.platform/ (accessed 30 December 2014).

Garousi, V., 2010. A Genetic Algorithm-Based Stress Test Requirements Generator Tool and Its Empirical Evaluation. IEEE Transaction on Software Engineering, vol. 36, no. 6, pp. 778-797.

Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization and Machine Learning, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Haupt, R.L., Haupt, S.E., 1998. Practical Genetic Algorithms. John Wiley & Sons, Inc., New York, NY, USA.

JBoss Community [Online], 2014. URL http://www.jboss.org/ (accessed 30 December 2014).

Jeong, G., Kim, S., Zimmermann, T., 2009. Improving Bug Triage with Bug Tossing Graphs. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09. ACM, New York, NY, USA, pp. 111-120.

Kagdi, H., Gethers, M., Poshyvanyk, D., Hammad, M., 2012. Assigning change requests to software developers. Journal of Software: Evolution and Process, vol. 24, no. 1, pp. 3-33.

Kagdi, H., Poshyvanyk, D., 2009. Who can help me with this change request? In: Proc. of the 17th International Conference on Program Comprehension. pp. 273-277.

Kapur, P., Ngo-The, A.Ruhe, G., Smith, A., 2008. Optimized staffing for product releases and its application at Chartwell Technology. Journal of Software Maintenance and Evolution: Research and Practice , vol. 20, no. 5, pp. 365 – 386.

Mann, H., Whitney, D., 1947. On a test of whether one of two random variables is stochastically larger than the other. Annals of Mathematical Statistics, vol. 18, no. 1, pp. 50 –60.

Matter, D., Kuhn, A., Nierstrasz, O., 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In: Proc. of the 6th IEEE International Working Conference on Mining Software Repositories. pp. 131-140.

McCartney, C., Lien, F., 2007. Development of a Genetic Algorithm-Based Solution Algorithm for 1-D Convection-Diffusion Analyses. In: Proc. of the 15th Conference of the Computational Fluid Dynamics Society of Canada.

McDonald, D., Ackerman, M., 2000. Expertise recommender: a flexible recommendation system and architecture. In: Proc. of the 2000 ACM Conference on Computer Supported Collaborative Work. pp. 231–240.

Mockus, A., Herbsleb, J., 2002. Expertise browser: a quantitative approach to identifying expertise. In: Proc. of the 24th International Conference on Software Engineering, pp. 503-512.

Otero, L., Centeno, G.,Ruiz-Torres, A., Otero, C., 2009. A systematic approach for resource allocation in software projects. Computers & Industrial Engineering, vol. 56, no. 4, pp. 1333-1339.

Penta, M.,Harman,M.,Antoniol, G., 2011. The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. Journal of Software - Practice and Experience, vol. 41, no. 5, pp. 495-519.

Park, J., Seo, D., Hong, G., Shin, D., Hwa, J., Bae, D. (2014). Practical Human Resource Allocation in Software Projects Using Genetic Algorithm. In: Proc. of the Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE 2014), Vancouver, Canada , pp. 688–694.

Rahman, M., Ruhe, G., Zimmermann, T., 2009. Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects. In: Proc. of the 3rd International Symposium on Empirical Software Engineering and Management, pp. 439-442.

Rahman, M., Sohan, S., Maurer, F., Ruhe, G., 2010. Evaluation of optimized staffing for feature development and bug fixing. In: Proc. of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Management.

Saaty, T., 1980. The analytic hierarchy process: planning, priority setting, resources allocation. McGraw-Hill.

Sackman, H., Erikson, W., Grant, E., 1968. Exploratory experimental studies comparing online and offline programming performance. Communications of the ACM., vol. 11, no. 1, pp. 3 – 11.

Schröter, A., Zimmermann, T., Zeller, A., 2006. Predicting component failures at design time. In: Proc. of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. pp. 18-27.

SEDS [Online], 2014. URL https://sites.google.com/site/mrkarim/data-sets (accessed 30 December 2014).

Smit, S., Eiben, A., 2010. Parameter tuning of evolutionary algorithms: Generalist vs. specialist. Lecture Notes in Computer Science, vol. 6024, pp. 542-551.

Trendowicz, A., Münch, J., 2009. Factors Influencing Software Development Productivity—State-of-the-Art and Industrial Experiences. Advances in Computers, vol. 77, pp. 185-241.

Valett, J., McGarry, F., 1989. A summary of software measurement experiences in the software engineering laboratory. Journal of Systems and Software, vol. 9, no. 2, pp. 137-148.

Vargha, A., Delaney, H., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics, vol. 25, no. 2, pp. 101 – 132.

Weiss, C., Premraj, R., Zimmermann, T., Zeller, A. , 2007. How long will it take to fix this bug? In: Proc. of the Fourth International Workshop on Mining Software Repositories.