Autom Softw Eng (2017) 24:455–498
DOI 10.1007/s10515-016-0204-z

An effective change recommendation approach for supplementary bug fixes

Xin Xia[1] · **David Lo**[2]

Received: 26 July 2015 / Accepted: 4 August 2016 / Published online: 26 August 2016

Abstract Bug fixing is one of the most important activities during software development and maintenance. A substantial number of bugs are often fixed more than once due to incomplete initial fixes which need to be followed up by supplementary fixes. Automatically recommending relevant change locations for supplementary bug fixes can help developers to improve their productivity. It also help improve the reliability of systems by highlighting locations that a developer potentially needs to change to completely remove a bug. Unfortunately, a recent study by Park et al. shows that many change recommendation techniques do not work for supplementary bug fixes. In this paper, to advance the capabilities of existing change recommendation techniques, we propose an effective approach named SUPLOCATOR to recommend relevant locations (i.e., methods) that need to be changed for supplementary bug fixes. Based on various relationships among methods, classes, and packages in the source code (such as containment, inheritance, historical co-change, etc.), SUPLOCATOR extracts six change relationship graphs. Next, SUPLOCATOR performs random walk on each of the 6 graphs, and for each it outputs a ranked list of candidate change locations. Finally, SUPLOCATOR combines these six ranked lists by leveraging genetic algorithm. To investigate the benefits of SUPLOCATOR, we perform experiments on three projects, i.e., Eclipse JDT, Eclipse SWT, and Equinox p2. The experimental results show that on average SUPLOCATOR can achieve top-1, top-5, and top-10 accuracies, mean reciprocal rank (MRR), and mean average precision (MAP) of 0.51, 0.65, 0.67, 0.58 and

✉ Xin Xia
xxia@zju.edu.cn

David Lo
davidlo@smu.edu.sg

[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China

[2] School of Information Systems, Singapore Management University, Singapore, Singapore

0.32 for the three projects, which improve the best variants of the approach proposed by Park et al. by 1523.09, 639.70, 550.62, 919.41, and 1478.44 %, respectively. It also improves the approach proposed by Saul et al. in terms of top-1, top-5, and top-10 accuracies, MRR, and MAP by 71.81, 29.54, 18.30, 47.24, and 56.60 %, respectively. Statistical tests show that the improvements are statistically significant.

Keywords Change recommendation · Supplementary bug fixes · Random walk · Genetic algorithm

1 Introduction

Due to the complexity of software systems, bugs are inevitable. Bug fixing is an essential activity in the life cycle of software development and maintenance. In practice, a substantial number of bugs are fixed in more than one try, i.e., the initial fixes for these bugs were incomplete or incorrect, and developers have to perform additional fixes (Park et al. 2012, 2014). These additional fixes are often referred to as *supplementary bug fixes* (Park et al. 2012). A previous study shows that around 22–33 % of resolved bugs involve supplementary bug fixes (Park et al. 2012). Automatically recommending relevant change locations for the supplementary bug fixes can help developers reduce debugging time, and improve their productivity. Moreover, change recommendation can help increase system reliability by highlight locations that a developer potentially needs to change to remove a bug completely.

Over the last decade, a number of change recommendation approaches have been proposed (Ying et al. 2004; Zimmermann et al. 2005; Hassan and Holt 2004; Nguyen et al. 2010). Zimmermann et al. and Ying et al. leverage association rule mining techniques to predict change locations based on historical co-change patterns mined from version control systems (Ying et al. 2004; Zimmermann et al. 2005). Hassan and Holt use both co-change patterns and structural dependencies in source code to predict change locations (Hassan and Holt 2004). Nguyen et al. propose FixWizard which leverages code clone techniques to suggest change locations (Nguyen et al. 2010). The accuracies of these approaches were evaluated on software commit data, i.e., commits were grouped into sets of transactions, and for each transaction, a subset of change locations in the transaction is used to predict the remaining change locations in the transaction.

Recently, Park et al. investigate the effectiveness of existing change recommendation approaches for supplementary bug fixes (Park et al. 2014). Park et al. generalize many existing change recommendation approaches by constructing multiple change relationship graphs (CRGs) where nodes correspond to methods, classes, and packages, and edges correspond to structural dependencies, co-change relationships, and code similarities. It thus considers structural dependencies previously considered by Robillard (2005) and Hassan and Holt (2004, 2006), co-change patterns previously considered by Ying et al. (2004) and Zimmermann et al. (2005), and content similarity considered by Nguyen et al. (2010). Their approach then traverses one or multiple of these CRGs to predict buggy methods that still need to be fixed after an initial bug fix. Unfortunately, their study finds that the effectiveness of traditional change recom-

mendation approaches is very low, highlighting the inherently challenging problem of recommending change locations for supplementary bug fixes.

In this paper, to advance existing change recommendation works, we propose an approach named SUPLOCATOR to effectively recommend relevant change locations (i.e., methods) for supplementary bug fixes. Extending Park et al.'s approach, our approach also leverages multiple relationships that exist between methods, classes, and packages in source code. We analyze six kinds of relationships: method invocation (i.e., one method invokes another method), containment (i.e., one method is contained in a class or a package), inheritance (i.e., one class extends another class), historical co-change (i.e., one method is changed together with another method in a commit), content similarity (i.e., two methods share similar contents), and name similarity (i.e., two methods share similar names). Based on the six relationships types, we create six CRGs where nodes are methods, classes, and packages, and edges are relationships between the nodes. In the end we have six CRGs: method invocation, containment, inheritance, content similarity, and name similarity graphs. Then, to identify the relevant change locations for a supplementary bug fix, SUPLOCATOR performs a random walk (Page et al. 1999) on each of the six graphs starting from nodes that correspond to methods that were fixed in the initial bug fix(es). Random walk would assign weights to each of the nodes in a graph according to the proximities of the nodes to the previously fixed nodes, and it would output a ranked list of the nodes. In total, we have six ranked lists correspond to the six CRGs. Next, SUPLOCATOR combines these six ranked lists by assigning different weights to them by leveraging genetic algorithms (GAs; Goldberg and Holland 1988).

We evaluate our approach on three projects: Eclipse JDT, Eclipse SWT, and Equinox p2 which contains 53350, 64962, and 51424 methods, respectively. In total, we analyze 2543 bugs with supplementary fixes. We measure the performance of our approach in terms of top-1, top-5, and top-10 recommendation accuracies, mean reciprocal rank (MRR; Baeza-Yates et al. 1999), and mean average precision (MAP; Baeza-Yates et al. 1999). The experimental results show that SUPLOCATOR achieves average top-1, top-5, and top-10 accuracies, MRR, and MAP of 0.51, 0.65, 0.67, 0.58 and 0.32, respectively for the three projects. We compare the effectiveness of our approach against the approach proposed by Park et al. and another approach (i.e., FRAN) that performs random walks on a single graph to find similar methods that we use for change recommendation (Saul et al. 2007). The results show that our approach can outperform the best variants of Park et al. approach by 1523.09, 639.70, 550.62, 919.41, and 1478.44 %, respectively, and Saul et al.'s approach by 71.81, 29.54, 18.30, 47.24, and 56.60 %, respectively. Statistical test (i.e., Wilcoxon signed-rank test) results show that the improvements are significant.

The main contributions of this paper are:

(1) We propose a hybrid change recommendation approach SUPLOCATOR, which performs random walk on multiple CRGs, and leverages GA to combine the multiple ranked lists outputted by these graphs.
(2) We experiment on a broad range of datasets containing a total of 2543 bugs with supplementary fixes to demonstrate the effectiveness of SUPLOCATOR. We show that SUPLOCATOR outperforms the approaches proposed by Park et al. and Saul

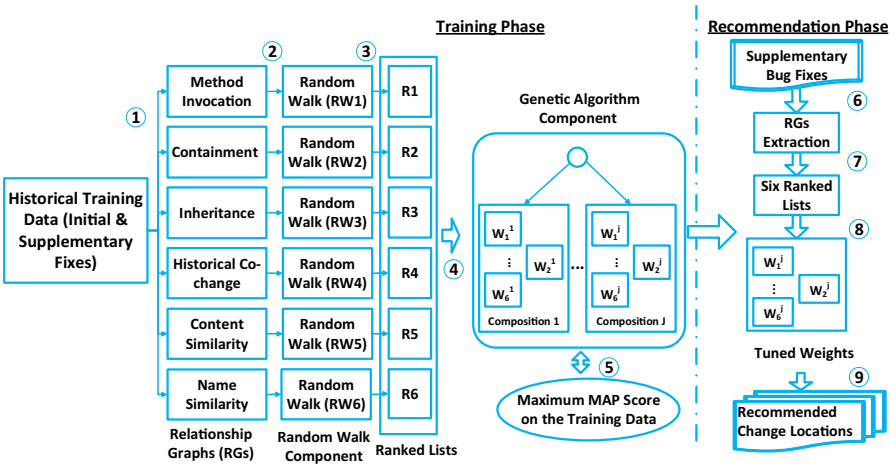

Fig. 1 Overall architecture of SUPLOCATOR

et al. by a substantial margin. And statistical test shows the improvements are significant.

The remainder of the paper is organized as follows. Section 2 describes an overview of SUPLOCATOR's architecture. Sections 3–5 elaborate on the three main components of SUPLOCATOR, i.e., CRG component, random walk component, and GA component, respectively. Section 6 presents the results of our comparative evaluation of SUPLOCATOR. Section 7 discuss the other settings of SUPLOCATOR, and threats to validity. Section 8 surveys the related work. Finally, Section 9 concludes the paper and mentions future work.

2 SUPLOCATOR architecture

Figure 1 presents the overall architecture of SUPLOCATOR which works on two phases: training phase and recommendation phase. In the training phase, we learn the values of some parameters (or weights) of SUPLOCATOR based on a training data. In the recommendation phase, we use SUPLOCATOR along with the learned weights to recommend change locations to bugs that require supplementary bug fixes.

2.1 Training phase

Our framework takes as input historical training data consisting of bug reports with known initial and supplementary bug fixes. For each of the initial bug fixes, we extract six CRGs of different types,[1] i.e., method invocation, containment, inheritance, historical co-change, content similarity, and name similarity graphs (Step 1).

[1] For more details of CRGs, please refer to Sect. 3.

Then, SUPLOCATOR inputs these graphs into six random walk components.[2] Each of the random walk components traverses one of the relationship graphs starting from the nodes corresponding to methods fixed in the initial bug fix (Step 2). The random walk process produces a ranked list of nodes in the CRG sorted based on their likelihood to be buggy (Step 3).

Next, in the GA component,[3] SUPLOCATOR searches for a good composition of the random walk components by assigning suitable weights to the components (Step 4). For each ranked list, we assign a weight to it, and in total we assign six weights to the six ranked list which correspond to the six CRGs. GA is a search heuristic that mimics the process of natural selection which models solutions in a search space as *chromosomes*. In our setting, a solution is a set of values for the weights of the random walk components. The algorithm picks a composition that maximizes the MAP score when it is used to recommend change locations to bugs in the training data (Step 5). MAP is a single-figure measure of quality which considers *all* correct results (in our case: change locations or methods).

2.2 Recommendation phase

After suitable values of the weights are learned, in the recommendation phase, SUPLOCATOR is then used to recommend change locations to new bugs that require supplementary bug fixes. For each of the bug, SUPLOCATOR first extracts the six CRGs (Step 6), and then performs random walk on these CRGs to get six ranked lists (Step 7). Next, these six ranked lists are then merged into one list by using the weights learned in the training phase (Step 8). Finally, it will output a unified ranked list of change locations (i.e., methods, Step 9).

3 Change relationship graph (CRG)

We use graph representations to capture different types of relationships among methods, classes, and packages in a source code. The nodes in such graphs are methods, classes, and packages, while the edges are the relationships. We consider six relationship types which correspond to the construction of six graphs (each capturing one relationship type); these relationships include: method invocation (i.e., one method invokes another method), containment (i.e., one method is contained in a class or a package), inheritance (i.e., one class extends another class), historical co-change (i.e., one method is changed together with another method in a commit), content similarity (i.e., two methods share similar contents), and name similarity (i.e., two methods share similar names).

Notice that methods fixed in an initial fix and those fixed in corresponding supplementary fixes contribute to the implementation of the same feature that is affected by the same bug. Thus, they are likely to be closely related to one another. By travers-

[2] For more details of the random walk component, please refer to Sect. 4.

[3] For more details of the GA component, please refer to Sect. 5.

ing a CRG, we can find closely related methods to the initial fixed methods. These methods located in the vicinity of the fixed methods in a CRG and these are likely candidates of methods fixed in the supplementary fixes. As an example, consider bug #210521 of Eclipse JDT, methods changed in its supplementary fixes are located one hop away from the initial methods in the inheritance and historical co-change graphs. As another example, for bug #251126 of Eclipse SWT, methods changed in its supplementary fixes are one hop away from the initial methods in the name similarity graph.

From the examples presented earlier, we can note that for some bugs, some CRGs are better able to capture closely related methods that match to those fixed in supplementary fixes. However, for other bugs, other CRGs are better. For example, for bug #210521 of Eclipse JDT, although the changed methods in its supplementary fixes can be located one hop away from the initial methods in the inheritance and historical co-change graphs, for the other CRGs, the changed methods cannot be located. As another example, for bug #251126 of Eclipse SWT, although the changed methods in its supplementary fixes can be located one hop away from the initial methods in the name similarity graph, and two hops away from the initial methods in the content similarity graph, for the other CRGs, these changed methods cannot be located. Since there is no CRG that is able to consistently outperform other graphs, we need to consider multiple CRGs. All of the six graphs are directed graphs, and we discuss how these graphs are created in the following paragraphs.

3.1 Method invocation, containment, and inheritance graphs

We use a partial program analysis (PPA) tool proposed by Dagenais and Hendren (2008) to extract structural dependency information from packages, classes, and methods in the source code files. Following Park et al. (2014), we use PPA to extract ASTs from changed files in revisions prior to the initial bug fix. Based on the ASTs, we extract the method invocation, containment, and inheritance relationships among methods, classes, and packages in the ASTs. Different from the work of Park et al. (2014), we create three independent graphs instead of one graph to represent the three relationships. We refer to them as the method invocation, containment, and inheritance graphs, respectively. For some changes which do not contain sufficient information for PPA to derive the type or package information, we set the number of nodes in the method invocation, containment, and inheritance graphs as 0, i.e., these three graphs do not contains any nodes and edges. Notice that even if some graphs do not contain any nodes or edges, SUPLOCATOR still can return a recommendation result.

To construct the method invocation graph, two methods A and B would have a edge if A calls B (the edge direction is from A to B), or B calls A (the edge direction is from B to A). To construct the containment graph, for each class and each method in it, we create a directed edge from the class to the method. Also, for each package and each class in it, we create a directed edge from the package to the class. To construct the inheritance graph, we create a directed edge from class $Class_1$ to class $Class_2$ if $Class_1$ is a subclass of $Class_2$. Also, we create a directed edge from method $Method_1$ to method $Method_2$ if $Method_1$ overrides $Method_2$.

3.2 Historical co-change graph

In the historical co-change graph, two method nodes A and B would have two directed edges if they were changed within the same commit (revision) prior to the initial bug fix (the directions are from A to B, and B to A). To create such a graph, we first extract all changes from a revision control system that were committed before the initial bug fixing commit was made. We then analyze the co-change relationships among methods in the source code and construct the historical co-change graph.

3.3 Content similarity graph

In the content similarity graph, two method nodes A and B would have two directed edges if their corresponding method bodies have similar contents (the directions are from A to B, and B to A). Following Park et al. (2014) study, we identify similar contents by using the output of CCFinderX (Kamiya et al. 2002) with the value of the minimum token size parameter set to 40. To create such a graph, we first extract the methods in the current snapshot of source code of the initial bug fix, and construct the content similarity edges among the methods.

Notice we use a clone detection technique instead of a textual similarity technique (e.g., semantic coupling proposed by Poshyvanyk et al. 2009) to construct the content similarity graph. If two methods share many common words, their textual similarity scores would be large. We find that methods in the same class/package tend to show higher textual similarity scores than methods in different classes/packages. Based on this observation, if we use textual similarity, it would be prone to recommend methods in the same class/package for supplementary bug fixes. However, we empirically find that most of the methods changed in the supplementary bug fixes are in the different classes/packages as the methods in the initial bug fixes. Thus, textual similarity cannot work well for our problem. Clone detection techniques focus on structural similarity rather than textual similarity. Structural similarity will assign high similarity scores to methods that share a similar structure, and we find that these methods are typically in different classes/packages. Thus, in this paper, we choose to use a clone detection technique instead of a textual similarity technique.

3.4 Name similarity graph

In the name similarity graph, following Park et al. (2014) study, two method nodes A and B would have two directed edges (the directions are from A to B, and B to A) if the following conditions hold: (1) the two methods are in the same package, (2) the classes that define the two methods have a name similarity score larger than 0.5, and (3) the names of the two methods have a name similarity score larger than 0.7. To measure the similarity of method and class names, Park et al. use the approach proposed by Xing and Stroulia (2005), i.e., the similarity between the names of two methods A and B is twice the number of consecutive character pairs (a.k.a. bigrams) that are common to both method names divided by the sum of the number of bigrams in the two method names. To create the name similarity graph, we first extract the

methods in the current snapshot of source code of the initial bug fix, and construct the name similarity edges among the methods.

4 Random walk component

After the extraction of the CRGs, given a set of nodes corresponding to methods changed in an initial bug fix, for each CRG, our approach ranks method nodes based on the likelihood of the corresponding methods to be changed in the supplementary bug fixes. Intuitively, the *closer the relationships* between a method to methods that are changed in the initial bug fix is, the more likely it is to be modified in the supplementary fixes.

Park et al.'s approach only recommends methods that are one hop away from the methods that are fixed in the initial bug fix. However, not all changed methods can be found one hop away from the methods that are fixed in the initial bug fix in the graphs. For example, for historical co-change graph, we find 59.36, 3.42, and 37.22 % changed methods in the supplementary fixes can be located 1 hop, 2 hops, and more than 3 hops away from the methods that are fixed in the initial bug fixes of Eclipse JDT. And for containment graph, we find 32.26, 20.55, and 47.19 % changed methods in the supplementary fixes can be located 1 hop, 2 hops, and more than 3 hops away from the initial fixed methods in Eclipse SWT.

Saul et al. have shown that closely-related methods can be identified by performing a random walk on a graph capturing structural dependencies of program elements in the vicinity of a method (Saul et al. 2007). In this work, we follow the intuition of Saul et al. by running random walk algorithms on each of the six CRGs to create six ranked lists each capturing closely related methods to those fixed in the initial fixes. To create this ranking, in a nutshell, for each CRG, our approach works as follows: (1) given a set of nodes that are fixed in the initial bug fix, we extract a subgraph that captures the vicinity of these nodes, (2) we then perform random walk to estimate the relationship strength of each node in this subgraph to the initial fixed methods. We then rank the method nodes based on their relationship strengths. We describe our subgraph extraction method and our random walk step in the following sub-sections.

4.1 Subgraph extraction

Before we describe the subgraph that we extract, we need to introduce some terminologies. Let us denote the nodes corresponding to methods that are changed in the initial bug fixes as V_{ini}. We define the set of nodes with outgoing edge(s) to at least one of the nodes in V_{ini} as parent nodes—denoted as V_{parent}. Next, we define the set of nodes with incoming edge(s) from at least one of the nodes in V_{ini} as child nodes—denoted as V_{child}. Furthermore, we define the set of nodes not in V_{ini} with incoming edge(s) from at least one node in V_{parent} as sibling nodes—denoted as $V_{sibling}$. Finally, we define the set of nodes not in V_{ini} with outgoing edge(s) to at least one node in V_{child} as spouse nodes—denoted as V_{spouse}. Figure 2 presents an illustration of these four

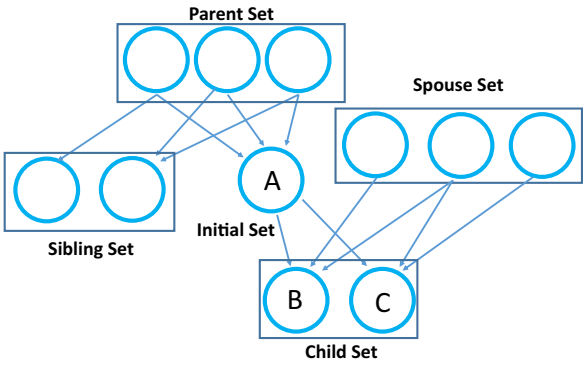

Fig. 2 An illustration of the parent set, child set, sibling set, and spouse set of an initial set of changed *nodes* in a change relationship graph

sets of nodes in a CRG.[4] Given V_{ini}, our approach first identifies its corresponding V_{parent}, V_{child}, $V_{sibling}$, and V_{spouse} sets. It then extracts a subgraph with nodes that appear in the union of these sets along with edges that connect them.

4.2 Random walk

We make use of a popular random walk algorithm named PageRank that was proposed by Page et al. (1999) and is used in Google to rank web pages. PageRank works in a number of iterations and eventually computes the probability of a random walker to traverse a node from an arbitrary node in a graph. The PageRank algorithm is originally proposed to study the importance of web pages in the Internet. In PageRank, one node corresponds to one web page, and one edge corresponds to the link in/out relationship between two web pages. In our paper, one node corresponds to a method, and one edge corresponds to one of the six types of change relationships as shown in the previous section. In such a way, the relationships among methods can be mapped to the relationships among webpages. Applying PageRank on our graphs will recover important nodes that are closely related to the initial methods (i.e., method that are modified in the initial bug fix) that are used to construct the graphs. These closely related methods are likely to be locations that are missed in the initial bug fix. In the initial iteration, all nodes in a graph are given an equal fixed probability. This probability is then updated in each iteration. At iteration k, the probability that PageRank assigns to a node n can be computed as follows:

$$P(n,\ k) = \frac{1-d}{ND} + r \sum_{u \in K(n)} \frac{P(u,\ k-1)}{|L(u)|}.$$

In the equation, d is the probability that a random walker continues to visit other nodes (a.k.a. the *damping factor*), ND is the number of nodes in the network, $K(n)$ is the set of nodes that link to n, and $L(u)$ is the set of nodes that u links to. The

[4] Similar terminologies were also used by Saul et al. in their similar-method recommendation work (Saul et al. 2007).

iteration continues until all the scores converge. Implementation-wise, we make use of the PageRank algorithm implemented in jung library.[5] We denote the PageRank probability of a node n as $PageRank(n)$.

In our random walk component, we adjust the probabilities outputted by PageRank by considering the degree of a node (West et al. 2001). The degree of a node n is the number of edges from and to the node n, and is denoted as $Degree(v)$. Our hypothesis is nodes with more degrees have more dependencies and thus are more likely to be buggy. The final ranking score of a node n, denoted as $Rank(n)$, is computed as:

$$Rank(n) = PageRank(n) \times Degree(n). \tag{1}$$

Since we have six types of CRGs which correspond to six random walk components, we denote the six random walk components as RW_1, $RW_2, \ldots, RW_6$.

5 Genetic algorithm component

The GA component merges the six ranked lists that are produced by the six random walk components (each working on a different CRG). Notice in our paper, we consider six types of CRGs, we need to combine the multiple lists from the output of the six types of CRGs. Some CRGs may produce better ranked lists than the others. Intuitively, ranked lists produced by those graphs need to be assigned higher weights. GAs (Goldberg and Holland 1988) can be used to combine the multiple ranking lists by assigning different weights to each of them. Other algorithms can potentially be used, e.g., learning to rank (Liu 2009). However, in this work, we choose GA since it is less susceptible to the issue with local optima, while learning to rank is often casted as an optimization problem (e.g., hill climbing Dai et al. 2013) which may often converge in a local optimum.

To merge the six ranked lists, our approach computes the composite rank of each method and ranks them based on their composite ranks. We define this composite rank in Definition 1.

Definition 1 (*composite rank*) Let us denote the rank of a method v produced by the six random walk components as $Rank_1(v)$, $Rank_2(v), \ldots, Rank_6(v)$. The composite rank $CRank(v)$ of method v is computed as:

$$CRank(v) = \sum_{i=1}^{6} \alpha_i \times Rank_i(v). \tag{2}$$

In the above equation, α_1–α_6 are the weights of the six random walk components. Each weight is a real number from zero to one.

To estimate the composite rank of a method well, our approach needs to estimate suitable weights α_1–α_6 in Eq. 2. We make use of a GA to search for a suitable

[5] http://jung.sourceforge.net/.

combination of weights based on a training data of completely fixed bugs and their supplementary fixes.

5.1 Search space and fitness functions

5.1.1 Search space

The search space of all possible combinations corresponds to the various assignments of values to the weights assigned to the six lists $\{\alpha_1, \alpha_2, \ldots, \alpha_6\}$. These weights are needed for the composite ranking problem, i.e., the creation of one unified list that merges the six. Each weight is a real number from zero to one. We refer to each composition of weights as a solution in the search space, denoted as $Sol = \{\alpha_1, \alpha_2, \ldots, \alpha_6\}$.

5.1.2 Fitness function

A fitness function measures the quality of a solution in the search space. In this paper, by default, we use the following fitness function we try to maximize:

$$Fitness = MAP(Sol). \tag{3}$$

In the above equation, $MAP(Par)$ is the MAP score achieved by using the weights in Sol on the training data. The details on how MAP score is computed is given in Sect. 6.2.

5.2 Detailed procedure

GA is a well-known search algorithm which models solutions in a search space as *chromosomes*. In our setting, a solution is a set of values for the weights. A chromosome contains a set of *genes* where a gene corresponds to a part of a solution (e.g., a value of a weight, in our setting). GA starts with a population of randomly constructed chromosomes, referred to as the initial *population*. It then evolves the population by generating subsequent *generations*, where each generation is another population of chromosomes. GA evolves the population by three operations: (1) selection operator, which selects *parent* chromosomes according to their fitness scores, (2) crossover operator, where the selected parents exchange their genes with a given probability, (3) mutation operator, where the genes of new chromosomes would be modified according to a given probability. More details about GA can be found in Sivanandam and Deepa (2007) and Goldberg and Holland (1988). We use the Roulette wheel selection procedure (Sivanandam and Deepa 2007; Goldberg and Holland 1988) as the selection operator. It assigns a high probability to a chromosome with a higher fitness score to be selected. For the crossover operator, we use the single point crossover operator. It processes pairs of chromosomes and for each pair, with a certain probability, it randomly picks a gene (i.e., a value of a weight) from a parent chromosome and swaps that gene and the subsequent ones with corresponding genes from the other parent

Algorithm 1 Estimation of weights in the GA component of SUPLOCATOR.

1: **EstimateWeights**($\{RW_1, RW_2, \ldots, RW_6\}$, $TRData$, $PopSize$, $MaxGen$)
2: **Input:**
3: $\{RW_1, RW_2, \ldots, RW_6\}$: Six random walk components
4: $TRData$: Training data
5: $PopSize$: Number of chromosomes in a population
6: $MaxGen$: Maximum number of generations
7: **Output:** $\alpha_1, \alpha_2, \ldots, \alpha_6$
8: **Method:**
9: Let P = Initial population with $PopSize$ members;
10: Evaluate P and record the best solution found so far (i.e., the solution with the maximum MAP score on $TRData$);
11: Let $curGen = 0$
12: **while** $curGen < MaxGen$ **do**
13: Let $P' = select(P)$;
14: $P' = crossover(P')$;
15: $P' = mutation(P')$;
16: Evaluate P' and record the best solution so far;
17: $curGen = curGen + 1$;
18: **end while**
19: **Output** $(\alpha_1, \alpha_2, \ldots, \alpha_6)$ which achieves the highest MAP score.

chromosome in the pair. For the mutation operator, we use random mutation. For each gene, with a certain probability, it randomly swaps the gene with another value in the range of zero to one.

Algorithm 1 presents the detailed steps to search for a suitable combination of weights. We first create an initial population (i.e., P) containing $PopSize$ chromosomes (i.e., solutions) that are chosen randomly, and we record the best solution (i.e., the solution with the maximum MAP score on $TRData$) among the solutions in P (lines 9 and 10). Remember that each solution in P is a set of weights $\{\alpha_1, \alpha_2, \ldots, \alpha_6\}$. Next, we evolve the population in $MaxGen$ iterations; for each iteration, we perform the selection, crossover, and mutation operations on the current population, and record the best solution found so far (lines 11–18). The algorithm returns the combination of $\alpha_1, \alpha_2, \ldots, \alpha_6$ which achieves the highest MAP score on $TRData$ (i.e., the best solution among solutions in the initial population and the populations generated in the $MaxGen$ generations).

6 Experiments and results

In this section, we evaluate the performance of SUPLOCATOR. The experimental environment is a Windows Server 2008, 64-bit, Intel Xeon 2.00 GHz server with 80 GB RAM.

6.1 Experiment setup

We use the datasets provided by Park et al. (2014) which contain a total of 2543 bugs each with one or more supplementary fixes. Table 1 present the statistics of the collected data. The columns correspond to the project name (**Project**), the time period

Table 1 Statistics of collected data

Projects	Time Period	# Bugs with Sup.	# Methods
Eclipse JDT	1 November 2001–18 December 2007	873	53,350
Eclipse SWT	11 October 2001–21 December 2008	1215	64,962
Equinox p2	25 October 2001–23 December 2009	455	51,424

of the collected bugs (**Time Period**), the number of bugs with supplementary fixes (**# Bugs with Sup.**), and the number of methods in the source code (**# Methods**). For each of these bugs, we extract the methods changed in the supplementary fixes as ground truth. Our task is to use the information (i.e., CRGs) in the initial fix to predict the modified methods in the supplementary fixes.

To simulate the usage of our approach in practice, we use the longitudinal data setup. We first sort bugs described in Table 1 in chronological order of their reported time. Next, we use the first 10 % of the bugs as a training set to build the SUPLOCATOR model. We use the remaining 90 % of the bugs as the test set to evaluate the effectiveness of SUPLOCATOR. This setup simulates the real usage of our tool in reality, since it is not possible to use future data to predict the past.

We compare SUPLOCATOR with a set of change recommendation approaches proposed by Park et al. (2014) and FRAN proposed by Saul et al. (2007). Park et al. generalize a number of previous works on change recommendation, e.g., Robillard (2005), Hassan and Holt (2004), Ying et al. (2004), Zimmermann et al. (2005) and Nguyen et al. (2010), and propose to use a number of different CRGs capturing different relationships between methods, classes, and packages to recommend change locations. They consider four types of CRGs capturing four different relationships, i.e., structural (method invocation, containment, and inheritance), co-change, content similarity, and name similarity. Their approach then identifies nodes in the CRGs that correspond to changes made in an initial bug fix and recommends methods whose nodes are one-hop away from the initial fix nodes (in one or more of the graphs) to be recommended as candidate change locations for the corresponding supplementary bug fix. Park et al. propose various variants of their approach: four variants only use one of the CRGs and we denote them as $Park^{Structure}$, $Park^{Change}$, $Park^{Content}$, and $Park^{Name}$; one variant makes use of selected edges from all the four CRGs and we denote it as $Park^{Sel}$; two other variants leverage developer specific patterns and package patterns and we denote them as $Park^{Dev}$ and $Park^{Pack}$, respectively. Saul et al. propose FRAN, which also performs random walk to find similar methods to input methods (Saul et al. 2007). We adapt their approach to recommend change locations. Differently from our approach, FRAN only performs random walk on one graph (i.e., structural dependency graph). Furthermore, they employ hypertext induced topic selection (HITS) algorithm (Kleinberg 1999) rather than PageRank. We obtained the implementation of Park et al.'s approaches from the authors and we reimplemented FRAN by making use of a publicly available implementation of HITS.[6]

[6] http://jung.sourceforge.net/.

In SUPLOCATOR, we use a simple GA (Sivanandam and Deepa 2007; Goldberg and Holland 1988) implemented in jgap (Meffert et al. 2011) to construct the GA component. Since GA exhibits some randomness (Sivanandam and Deepa 2007; Goldberg and Holland 1988; Xia et al. 2014a), we run the GA 10 times and report the average performance score. This follows the recommendation made by Arcuri and Briand (2011), Liu et al. (2010), Canfora et al. (2013), and Xia et al. (2014a) to evaluate random algorithms in software engineering context. The parameters of GA that we use in this study are as follows:

(1) *Population size* we set a moderate population size (PopSize) of 500 chromosomes.
(2) *Number of generations* we set the maximum number of generations (MaxGen) to 200.
(3) *Crossover operator* we use a single point crossover operator with crossover probability (CrossProb) of 0.35.
(4) *Mutation operator* we use a random mutation operator with mutation probability (MutProb) of 0.08.

6.2 Evaluation metrics

To evaluate SUPLOCATOR, we use top-k prediction accuracy, MRR (Baeza-Yates et al. 1999), and MAP (Baeza-Yates et al. 1999), which are commonly used in evaluating recommendation systems in the software engineering literature (Zhou et al. 2012; Rao and Kak 2011; Xia et al. 2014c; Tamrawi et al. 2011).

6.2.1 Top-k prediction accuracy

Top-k prediction accuracy is the percentage of bugs with supplementary patches whose ground truth methods are ranked in the top-k positions in the returned ranked lists of change locations (i.e., methods). Given a bug b, if at least one of its top-k recommended methods is actually changed in its supplementary bug fix, we consider the recommendation to be successful, and set the value $success(b, top-k)$ to 1; else we consider the recommendation to be unsuccessful, and set the value $success(b, top-k)$ to 0. Given a set of bugs requiring supplementary fixes, denoted as $SupBugs$, its top-k prediction accuracy $Top@k$ is computed as:

$$Top@k = \frac{\sum_{b \in SupBugs} success(b, top\text{-}k)}{|SupBugs|}. \quad (4)$$

The higher the top-k accuracy score is, the better a code-reviewer recommendation technique performs. In this paper, we set k = 1, 5, and 10.

6.2.2 Mean reciprocal rank (MRR)

MRR is a popular metric used to evaluate an information retrieval technique (Baeza-Yates et al. 1999). Given a query (in our case: a bug b), its reciprocal rank is the multiplicative inverse of the rank of the first correct document (in our case: change

location or method) in a rank list produced by a ranking technique (in our case: a change recommendation technique). MRR is the average of the reciprocal ranks of all bugs in a set of bugs. The MRR of a set of bugs requiring supplementary fixes $SupBugs$ is computed as:

$$MRR(R) = \frac{1}{|SupBugs|} \sum_{b \in SupBugs} \frac{1}{rank(b)}. \quad (5)$$

In the above equation, $rank(b)$ refers to the position of the first correctly recommended method in the ranked list returned by a change recommendation technique for bug b.

6.2.3 Mean average precision (MAP)

MAP is a single-figure measure of quality, and it has been shown to have especially good discrimination and stability to evaluate ranking techniques (Baeza-Yates et al. 1999). Different from top-k accuracy and MRR that only consider the first correct result, MAP considers *all* correct results. For a single query (in our case: a bug b), its *average precision* is defined as the mean of the precision values obtained for different sets of top-k documents (in our case: change locations or methods) that were retrieved before each relevant document is retrieved, which is computed as:

$$AvgP(b) = \frac{\sum_{j=1}^{M} P(j) \times Rel(j)}{Number\ of\ relevant\ methods}. \quad (6)$$

In the above equation, M is the number of candidate methods in a ranked list, $Rel(j)$ indicates whether the method at position j is relevant or not (in our case: a ground truth method or not), and $P(j)$ is the precision at the given cut-off position j and is computed as:

$$P(j) = \frac{Number\ of\ relevant\ methods\ in\ top\ j\ positions}{j}.$$

Then the MAP for a set of bugs requiring supplementary fixes $SupBugs$ is the mean of the average precision scores for all bugs in $SupBugs$:

$$MAP = \frac{\sum_{b \in SupBugs} AvgP(b)}{|SupBugs|}. \quad (7)$$

In change recommendation, a supplementary bug fix may require a number of methods to be changed. We use MAP to measure the average performance of SUPLOCATOR to recommend all of the relevant methods. The higher the MAP value, the better the change recommendation performance is.

6.2.4 Completeness@K

Suppose we have m bugs with supplementary fixes. For the ith bug b_i, let the set of its actual change locations be C_i. We recommend the top-k change locations T_i for b_i according to our approach. Then, the completeness of our approach at the top-k recommendations, denoted as $Completeness@K$ is:

$$Completeness@K = \frac{1}{m}\sum_{i=1}^{i=m}\frac{|C_i \bigcap P_i|}{C_i}. \quad (8)$$

In this paper, we choose K equals to 5, 10, and 20, and compute the $Completeness@K$ values for SUPLOCATOR and the baseline approaches. We use Completeness@K as a supplementary evaluation metric in Sects. 7.4–7.6.

6.3 Research questions

This paper addresses the following research questions:

RQ1: How effective is SUPLOCATOR in recommending change locations? How much improvement can it achieve over the state-of-the-art approaches?
RQ2: What is the performance of the six random walk components? And what is the benefit of our composite GA model?
RQ3: What is the effect of varying the number of bugs in the training set on the effectiveness of SUPLOCATOR?
RQ4: What is the effect of varying the fitness function of the GA component on the effectiveness of SUPLOCATOR?

6.4 Results

6.4.1 RQ1: How effective is SUPLOCATOR in recommending change locations? How much improvement can it achieve over the state-of-the-art approaches?

Motivation the more accurate SUPLOCATOR is, the more benefit SUPLOCATOR would give to its users. Thus, in this research question, we evaluate the effectiveness of SUPLOCATOR and compare it with the state-of-the-art approaches.

Approach to answer RQ1, we compare SUPLOCATOR with the approaches proposed by Park et al. (2014) (i.e., $Park^{Structure}$, $Park^{Change}$, $Park^{Content}$, $Park^{Name}$, $Park^{Sel}$, $Park^{Pack}$, and $Park^{Dev}$), and FRAN proposed by Saul et al. (2007). We evaluate them by using the longitudinal data setup, and record the top-k prediction accuracies (k = 1, 5, 10, and 20), MRR, and MAP.

To check if the differences in the performance of SUPLOCATOR and the baseline approaches are statistically significant, for the each dataset, we apply the Wilcoxon signed-rank test (Wilcoxon 1945) at 95 % significance level on the paired data which corresponds to the top-k accuracies, MRR, and MAP scores of two competing approaches, respectively. Since we have three projects, for each evaluation metrics

Table 2 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those for the approaches proposed by Park et al. and FRAN for bugs in Eclipse JDT project

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.61**	**0.74**	**0.78**	**0.67**	**0.39**
$Park^{Structure}$	0.03	0.07	0.09	0.05	0.02
$Park^{Change}$	0.03	0.06	0.09	0.04	0.02
$Park^{Content}$	0.01	0.01	0.01	0.01	0.00
$Park^{Name}$	0.03	0.09	0.11	0.06	0.02
$Park^{Sel}$	0.01	0.03	0.05	0.03	0.01
$Park^{Dev}$	0.00	0.01	0.01	0.01	0.00
$Park^{Pack}$	0.00	0.02	0.03	0.01	0.00
FRAN	0.35	0.56	0.64	0.45	0.25

The best performance is highlighted in bold

Table 3 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those for the baseline approaches for bugs in Eclipse SWT project

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.57**	**0.70**	**0.73**	**0.63**	**0.33**
$Park^{Structure}$	0.03	0.06	0.07	0.04	0.01
$Park^{Change}$	0.04	0.11	0.17	0.08	0.03
$Park^{Content}$	0.01	0.02	0.03	0.01	0.00
$Park^{Name}$	0.05	0.12	0.15	0.08	0.03
$Park^{Sel}$	0.01	0.03	0.07	0.03	0.01
$Park^{Dev}$	0.01	0.01	0.04	0.02	0.01
$Park^{Pack}$	0.01	0.02	0.06	0.02	0.01
FRAN	0.34	0.54	0.60	0.43	0.21

The best performance is highlighted in bold

Table 4 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those for the baseline approaches for bugs in Equinox p2 project

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.38**	**0.49**	**0.52**	**0.43**	**0.23**
$Park^{Structure}$	0.04	0.09	0.12	0.06	0.02
$Park^{Change}$	0.01	0.03	0.06	0.03	0.01
$Park^{Content}$	0.00	0.00	0.00	0.00	0.00
$Park^{Name}$	0.01	0.05	0.06	0.03	0.01
$Park^{Sel}$	0.01	0.04	0.07	0.03	0.01
$Park^{Dev}$	0.00	0.02	0.04	0.02	0.01
$Park^{Pack}$	0.00	0.01	0.02	0.01	0.00
FRAN	0.21	0.40	0.46	0.30	0.14

The best performance is highlighted in bold

(i.e., top-k accuracies, MRR, and MAP), we also use Bonferroni correction (Abdi 2007) to counteract the results of multiple comparisons.

Results Tables 2, 3, and 4 present the top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those of the approaches proposed by Park

Table 5 P-values of SUPLOCATOR compared with the baseline approaches for bugs in Eclipse JDT project (after Bonferroni correction)

Sup. versus baseline	Top-1	Top-5	Top-10	MRR	MAP
Sup. versus $Park^{Structure}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Change}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Content}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Name}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Sel}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Dev}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Pack}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
FRAN	$2.1e^{-9}$	$7.2e^{-6}$	$5.4e^{-10}$	$3.6e^{-10}$	$6.3e^{-8}$

Table 6 P-values of SUPLOCATOR compared with the baseline approaches for bugs in Eclipse SWT project (after Bonferroni correction)

Sup. versus baseline	Top-1	Top-5	Top-10	MRR	MAP
Sup. versus $Park^{Structure}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Change}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Content}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Name}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Sel}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Dev}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Pack}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
FRAN	$2.7e^{-10}$	$4.2e^{-8}$	$5.4e^{-6}$	$6.6e^{-7}$	$8.1e^{-7}$

et al. and FRAN proposed by Saul et al. for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2 projects, respectively. We notice that SUPLOCATOR outperforms the approaches proposed by Park et al. by a substantial margin.

On average across the three projects, SUPLOCATOR achieves top-1, top-5, and top-10 prediction accuracies, MRR, and MAP of 0.52, 0.65, 0.67, 0.58, and 0.32, which outperform the best approaches (i.e., $Park^{Name}$) proposed Park et al. by 1523.09, 639.70, 550.62, 919.41, and 1478.44 %, and FRAN by 71.81, 29.54, 18.30, 47.24, and 56.60 %, respectively.

Tables 5, 6, and 7 present the p-values of SUPLOCATOR compared with the baseline approaches for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2 projects by applying Wilcoxon signed-rank test with Bonferroni correction, respectively. Statistical testing shows that the improvements of SUPLOCATOR over the those of Park et al.'s approaches and FRAN are significant at 95 % confidence level across the three projects (i.e., the p-values are less than 0.05).

We notice the performance of the approaches proposed by Park et al. are extremely low; on average across the three projects, the top-1, top-5, top-10, and top-20 prediction

Table 7 P-values of SUPLOCATOR compared with the baseline approaches for bugs in Equinox p2 project (after Bonferroni correction)

Sup. versus baseline	Top-1	Top-5	Top-10	MRR	MAP
Sup. versus $Park^{Structure}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Change}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Content}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Name}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Sel}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Dev}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
Sup. versus $Park^{Pack}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$	$6.6e^{-16}$
FRAN	$4.2e^{-12}$	$8.7e^{-10}$	$4.8e^{-8}$	$5.4e^{-7}$	$6.9e^{-8}$

accuracies, MRR, and MAP of the best approach ($Park^{Name}$) are 0.03, 0.09, 0.10, 0.12, 0.06, and 0.02, respectively. Our findings are consistent with the conclusion in Park et al.'s study which describes the authors' skepticism on the ability of existing change recommendation techniques for supplementary bug fixes.

Notice Park et al.'s approaches only recommend methods that are one hop away from the methods that are fixed in the initial bug fix. As shown in Sect. 4.2, a number of methods changed in the supplementary bug fixes are more than one hop away from the methods that are fixed in the initial bug fix. Random walk technique can help to identify methods which are more than one hop away from the methods that are fixed in the initial bug fix. Also, Park et al.'s approaches and FRAN only consider one type of CRG, while SUPLOCATOR considers six types of CRGs and combine them to achieve a better performance. These differences make our SUPLOCATOR achieves substantial improvements over the baseline approaches.

SUPLOCATOR *outperforms the approaches proposed by Park et al. and FRAN proposed by Saul et al. by substantial margins, and the improvements are statistically significant.*

6.4.2 RQ2: What is the performance of the six random walk components? And what is the benefit of our composite GA model?

Motivation SUPLOCATOR has six random walk components which processes the six different CRGs. In this RQ, we would like to investigate the performance of each of them. We want to see whether the combination of the six random walk components can achieve better result than the individual random walk components.

Approach to address RQ2, we compare SUPLOCATOR with its six random walk components built on the six different CRGs. We denote the six random walk components as RW^{Method}, $RW^{Containment}$, $RW^{Inheritance}$, RW^{Change}, $RW^{Content}$, and RW^{Name}. We evaluate them by using the longitudinal data setup, and record the top-k prediction accuracies (k = 1, 5, and 10), MRR, and MAP.

Table 8 Spearman's rho and correlation level

Spearman's rho	Correlation levels
0.0–0.1	None
0.1–0.3	Small
0.3–0.5	Moderate
0.5–0.7	High
0.7–0.9	Very high
0.9–1.0	Perfect

Table 9 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those of its six random walk components (denoted as RW^{Method}, $RW^{Containment}$, $RW^{Inheritance}$, $RW^{Historical}$, $RW^{Content}$, and RW^{Name}) for bug reports in Eclipse JDT project
The best performance is highlighted in bold

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.61**	**0.74**	**0.78**	**0.67**	**0.39**
RW^{Method}	0.04	0.17	0.28	0.12	0.06
$RW^{Containment}$	0.00	0.56	0.67	0.19	0.13
$RW^{Inheritance}$	0.58	0.71	0.74	0.64	0.37
RW^{Change}	0.22	0.37	0.46	0.30	0.16
$RW^{Content}$	0.50	0.67	0.69	0.58	0.33
RW^{Name}	0.18	0.41	0.54	0.29	0.17

Table 10 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those of its six random walk components for bug reports in Eclipse SWT project

The best performance is highlighted in bold

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.57**	**0.70**	**0.73**	**0.63**	**0.33**
RW^{Method}	0.02	0.07	0.17	0.06	0.03
$RW^{Containment}$	0.00	0.55	0.66	0.18	0.11
$RW^{Inheritance}$	0.50	0.67	0.69	0.57	0.30
RW^{Change}	0.16	0.32	0.42	0.24	0.10
$RW^{Content}$	0.32	0.50	0.56	0.40	0.21
RW^{Name}	0.16	0.44	0.56	0.28	0.15

We use Spearman's rho to measure the correlation level of our SUPLOCATOR to its six components. The Spearmans rho ranges from −1 to 1, where −1 and 1 separately mean that a perfect negative and positive monotonic correlation, and 0 means the variables are independent of each other. Table 8 describes the meanings of various correlation coefficient values and the corresponding correlation levels (Hopkins 1997). To do so, for the each dataset, we compute the Spearmans rho on the paired data which corresponds to the top-k accuracies, MRR, and MAP scores of two competing approaches, respectively. Since we have three projects, for each evaluation metrics (i.e., top-k accuracies, MRR, and MAP), we also use Bonferroni correction to counteract the results of multiple comparisons.

Results Tables 9, 10, and 11 present the top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those of the six random walk components

Table 11 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those of its six random walk components for bug reports in Equinox p2 project

The best performance is highlighted in bold

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.38**	**0.49**	**0.52**	**0.43**	**0.23**
RW^{Method}	0.04	0.14	0.21	0.10	0.04
$RW^{Containment}$	0.00	0.32	0.40	0.11	0.07
$RW^{Inheritance}$	0.33	0.45	0.47	0.38	0.20
RW^{Change}	0.18	0.34	0.40	0.26	0.12
$RW^{Content}$	0.33	0.45	0.47	0.38	0.20
RW^{Name}	0.15	0.35	0.40	0.23	0.12

for bug reports in Eclipse JDT, Eclipse SWT, and Equinox p2 projects, respectively. Among the 6 random walk components, $RW^{Inheritance}$ achieves the best performance, i.e., top-1, top-5, and top-10 prediction accuracies, MRR, and MAP of 0.47, 0.61, 0.63, 0.53, and 0.29 on average across the three projects, respectively. Still our SUPLOCATOR achieve a better performance, it improves the inheritance component by 10.47, 5.99, 6.54, 8.80, and 10.85 % in terms of top-1, top-5, top-10, and top-20 prediction accuracies, MRR, and MAP.

To investigate why $RW^{Inheritance}$ achieves the best performance among the six components, we find that in our datasets, if method *A* is changed in the initial patch, the methods that *A* overrides would have high chances to be changed in the supplementary fixes. Moreover, recall that the inheritance graph consists of different types of edges (i.e., edges between packages, classes, and method nodes), and these multiple types of edges provide more information for our random walk component to more effectively estimate buggy methods requiring additional fixes.

Notice in Tables 9, 10, and 11, the performance of RW^{Method} is very low compared with the other components. We manually check the results, and we notice that the method invocation graph typically contains many more nodes and edges among the 6 graphs, and many of these nodes are irrelevant ones. This makes it more difficult to rank the correct nodes (i.e., methods that are fixed in the supplementary bug fixes) higher. Also, we notice that for $RW^{Containment}$, its top-1 accuracy is 0 while its top-5 accuracy is much higher in these three tables. We also manually check the results, and we find that in the ranking list outputted by $RW^{Containment}$, the methods with the highest in-degrees and out-degrees are always listed first, however, these methods are not changed in the supplementary bug fixes. Thus, the top-1 accuracies for $RW^{Containment}$ are 0 in all of the above tables. Also, for many cases, $RW^{Containment}$ can recommend the right methods in the top-5 positions.

Tables 12, 13, and 14 present the Spearman's rho and p-values of SUPLOCATOR compared with those of its six random walk components for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2 projects, respectively. We notice that the values of Spearman's rho are less than 0.3, thus the correlation levels of SUPLOCATOR rankings compared with those of its six components are small or none.

SUPLOCATOR *achieves a better performance than its six random walk components. And in most cases, the improvements are statistically significant.*

Table 12 Spearman's rho and p-values of SUPLOCATOR compared with its six random walk components for bugs in Eclipse JDT project

Sup. versus baseline	Top-1		Top-5		Top-10		MRR		MAP	
	Rho	p-value	Rho	p-value	Rho	p-value	Rho	p-value	Rho	p-value
Sup. versus RW^{Method}	0.04	$6.6e^{-16}$	0.05	$2.1e^{-15}$	0.02	$7.2e^{-10}$	0.04	$6.3e^{-10}$	0.02	$1.8e^{-9}$
Sup. versus $RW^{Containment}$	0.00	$6.6e^{-16}$	0.10	$3.6e^{-12}$	0.11	$2.4e^{-9}$	0.05	$3.6e^{-9}$	0.04	$3.9e^{-7}$
Sup. versus $RW^{Inheritance}$	0.15	0.0028	0.16	0.042	0.22	0.030	0.17	0.009	0.25	0.006
Sup. versus RW^{Change}	0.08	$2.4e^{-10}$	0.04	$1.8e^{-16}$	0.08	$6.3e^{-10}$	0.08	$2.7e^{-8}$	0.04	$2.4e^{-5}$
Sup. versus $RW^{Content}$	0.12	0.0012	0.12	0.0027	0.10	0.0018	0.11	0.0024	0.12	0.0013
Sup. versus RW^{Name}	0.06	$3.3e^{-10}$	0.08	$1.2e^{-9}$	0.04	$1.5e^{-11}$	0.04	$5.4e^{-10}$	0.02	$5.7e^{-9}$

Table 13 Spearman's rho and p-values of SUPLOCATOR compared with its six random walk components for bugs in Eclipse SWT project

Sup. versus baseline	Top-1		Top-5		Top-10		MRR		MAP	
	Rho	p-value	Rho	p-value	Rho	p-value	Rho	p-value	Rho	p-value
Sup. versus RW^{Method}	0.02	$2.4e^{-15}$	0.01	$6.6e^{-15}$	0.01	$5.7e^{-10}$	0.03	$7.8e^{-10}$	0.01	$6.6e^{-16}$
Sup. versus $RW^{Containment}$	0.00	$6.6e^{-16}$	0.11	$4.2e^{-6}$	0.18	$2.1e^{-9}$	0.05	$2.4e^{-9}$	0.03	$4.8e^{-13}$
Sup. versus $RW^{Inheritance}$	0.12	0.0012	0.19	0.003	0.24	0.0027	0.18	0.0012	0.27	0.0045
Sup. versus RW^{Change}	0.05	$3.6e^{-9}$	0.04	$7.5e^{-10}$	0.09	$3.6e^{-8}$	0.06	$2.4e^{-7}$	0.03	$2.7e^{-10}$
Sup. versus $RW^{Content}$	0.08	$3.9e^{-14}$	0.09	$3.3e^{-6}$	0.12	$1.2e^{-6}$	0.12	$4.5e^{-8}$	0.09	$2.4e^{-9}$
Sup. versus RW^{Name}	0.04	$4.5e^{-9}$	0.11	$1.8e^{-6}$	0.12	$1.8e^{-9}$	0.08	$5.1e^{-9}$	0.07	$5.4e^{-6}$

6.4.3 RQ3: What is the effect of varying the number of bugs in the training set on the effectiveness of **SUPLOCATOR***?*

Motivation SUPLOCATOR takes as input a training set of bugs with their supplementary fixes to tune the six parameters of its GA component. By default, this training set includes the first 10 % of all bugs that we investigate in this work. In this research question, we perform a sensitivity analysis by investigating the impact of varying the size of the training set on the effectiveness of SUPLOCATOR.

Table 14 Spearman's rho and p-values of SUPLOCATOR compared with its six random walk components for bugs in Equinox p2 project

Sup. versus baseline	Top-1		Top-5		Top-10		MRR		MAP	
	Rho	p-value	Rho	p-value	Rho	p-value	Rho	p-value	Rho	p-value
Sup. versus RW^{Method}	0.02	$6.6e^{-16}$	0.05	$2.7e^{-15}$	0.04	$8.4e^{-10}$	0.02	$9.3e^{-14}$	0.01	$3.6e^{-11}$
Sup. versus $RW^{Containment}$	0.00	$6.6e^{-16}$	0.11	$5.7e^{-13}$	0.12	$2.7e^{-9}$	0.03	$3.6e^{-14}$	0.02	$7.2e^{-14}$
Sup. versus $RW^{Inheritance}$	0.15	0.0018	0.18	0.036	0.18	0.036	0.14	0.0066	0.26	0.0004
Sup. versus RW^{Change}	0.06	$2.7e^{-6}$	0.12	$2.4e^{-15}$	0.15	$2.1e^{-11}$	0.08	$2.4e^{-12}$	0.06	$2.7e^{-12}$
Sup. versus $RW^{Content}$	0.16	0.0021	0.20	0.033	0.19	0.033	0.15	0.0063	0.26	0.0012
Sup. versus RW^{Name}	0.05	$4.2e^{-8}$	0.14	$1.8e^{-6}$	0.15	$1.8e^{-9}$	0.06	$5.7e^{-12}$	0.06	$6.6e^{-13}$

Approach to answer this research question, we vary the the size of the training set by including the first 5–30 % of the bugs with supplementary fixes, and record the top-k prediction accuracies (k = 1, 5, 10, and 20), MRR, and MAP. Suppose that there are M bugs with supplementary fixes in our dataset, when we set the number of bugs in the training set to $n\%$, we would use the remaining $(1 - n\%) \times M$ bugs as the test set.

Results Figure 3 presents the top-1, top-5, and top-10 accuracies, MRR, and MAP of SUPLOCATOR with different percentages of training bugs from Eclipse JDT, Eclipse SWT, and Equinox p2 datasets, respectively. We notice that as the number of bugs in the training set increases, the performance of SUPLOCATOR is slightly increased. For example, the MRR and MAP scores for the Eclipse JDT dataset are increased from 0.65 to 0.70, and from 0.38 to 0.43 when the percentage of bugs in the training set is increased from 5 to 30 %. With more bugs in the training set, we find that the model building time is increased. For example, the model building time for SUPLOCATOR trained using 30 % of all bugs in Eclipse SWT project is five times more than that for SUPLOCATOR trained using 10 % of all bugs (i.e., 1836 vs. 365 s).

The effectiveness of SUPLOCATOR *is slightly increased as we increase the number of bugs in the training set. However, with more bugs in the training set, the time it takes to build a model is increased. Developers need to judiciously consider training time and effectiveness in deciding the number of bugs used to train* SUPLOCATOR.

6.4.4 RQ4: What is the effect of varying the fitness function of the GA component on the effectiveness of **SUPLOCATOR**?

Motivation in the GA component of SUPLOCATOR, fitness score measures the quality of a solution in a search space. By default, we set the fitness function as the MAP,

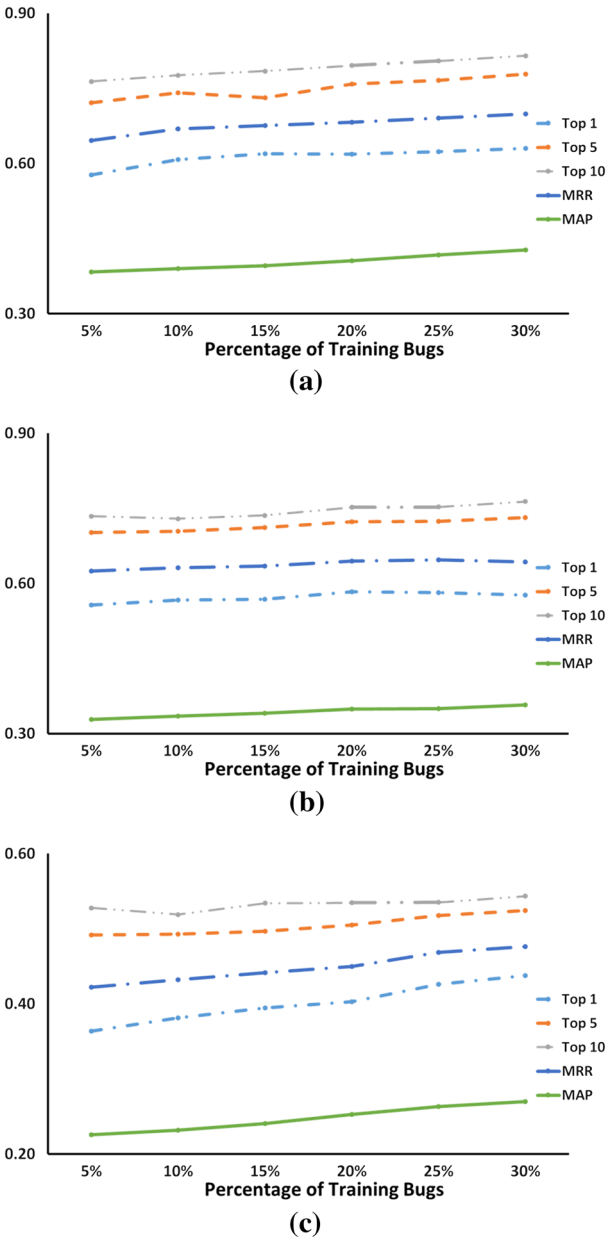

Fig. 3 Top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different percentages of training bugs in Eclipse JDT, Eclipse SWT, and Equinox p2

i.e., our target of the model building phase is to find a composition of weights which maximizes the MAP score on the training data. In this research question, we investigate the impact of using different fitness functions on the effectiveness of SUPLOCATOR.

Table 15 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR using different fitness functions for Eclipse JDT, Eclipse SWT, and Equinox p2 datasets

Projects	Approaches	Top-1	Top-5	Top-10	MRR	MAP
JDT	SUP^{TOP}	0.54	0.73	0.77	0.63	0.37
	SUP^{MRR}	0.59	0.72	0.78	0.65	0.39
	SUP^{MAP}	0.61	0.74	0.78	0.67	0.39
SWT	SUP^{TOP}	0.40	0.67	0.72	0.52	0.30
	SUP^{MRR}	0.55	0.70	0.73	0.62	0.33
	SUP^{MAP}	0.57	0.70	0.73	0.63	0.33
Equ.	SUP^{TOP}	0.29	0.46	0.51	0.36	0.21
	SUP^{MRR}	0.39	0.50	0.52	0.44	0.23
	SUP^{MAP}	0.38	0.49	0.52	0.43	0.23

Approach to answer this research question, we choose three different fitness functions such as top-20 prediction accuracy, MRR, and MAP, and record the top-k prediction accuracies (k = 1, 5, and 10), MRR, and MAP. We denote SUPLOCATOR with top-20 prediction accuracy, MRR, and MAP as SUP^{TOP}, SUP^{MRR}, and SUP^{MAP}, respectively.

Results Table 15 presents the top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR using different fitness functions for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2 projects. We notice among the three fitness functions, SUP^{TOP} does not perform as well as SUP^{MRR}, and SUP^{MAP}. SUP^{MRR} and SUP^{MAP} perform almost as well, i.e., the effectiveness of SUP^{MAP} is slightly better than that of SUP^{MRR} for Eclipse JDT and Eclipse SWT datasets, and the performance of SUP^{MRR} is slightly better than that of SUP^{MAP} for Equinox p2 dataset.

The variants of SUPLOCATOR *with MAP and MRR used as fitness functions (i.e.,* SUP^{MRR} *and* SUP^{MAP}*) are almost equally effective and are better than the variant with top-20 accuracy as fitness function (i.e.,* SUP^{TOP}*). In practice, we recommend developer to use* SUP^{MRR} *and* SUP^{MAP} *instead of* SUP^{TOP}.

7 Discussion

7.1 Component weights

Table 16 presents the average weights for the six random walks components for Eclipse JDT, Eclipse SWT, and Equinox p2 datasets, respectively. We notice for different projects, the weights are different. Across the projects, we notice that the weights for $RW^{Inheritance}$ is the highest. This is consistent with our findings to answer RQ2: we find $RW^{Inheritance}$ achieves the best performance among the six random walk components.

Table 16 Average weights for the six CRGs

Weights	Eclipse JDT	EclipseSWT	Equinox p2
RW^{Method}	0.02	0.13	0.60
$RW^{Containment}$	0.01	0.01	0.01
$RW^{Inheritance}$	1.00	1.00	0.99
RW^{Change}	0.20	0.56	0.04
$RW^{Content}$	0.14	0.01	0.97
RW^{Name}	0.02	0.04	0.02

Table 17 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those for the approaches proposed by Park et al. and FRAN for bugs in Eclipse JDT project after removing the tangled changes

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.62**	**0.74**	**0.79**	**0.66**	**0.40**
$Park^{Structure}$	0.04	0.08	0.10	0.06	0.04
$Park^{Change}$	0.05	0.07	0.10	0.06	0.04
$Park^{Content}$	0.04	0.03	0.02	0.01	0.00
$Park^{Name}$	0.04	0.09	0.12	0.08	0.02
$Park^{Sel}$	0.03	0.02	0.06	0.02	0.04
$Park^{Dev}$	0.00	0.05	0.04	0.02	0.00
$Park^{Pack}$	0.00	0.03	0.04	0.01	0.00
FRAN	0.37	0.57	0.65	0.47	0.27

The best performance is highlighted in bold

7.2 Impact on the tangled changes

Notice our datasets may contain changes for other purposes such as functionality enhancement, presentation/output changes, refactoring, re-structuring, etc. Such changes are commonly referred to as tangled changes (Herzig and Zeller 2013), and these changes may affect the performance of SupLocator. Here, we use the algorithm proposed by Herzig and Zeller (2013) to remove the tangled changes in our datasets. After the clean-up, we have 752, 1023, and 398 changes left in Eclipse JDT, Eclipse SWT, and Equinox p2, respectively.

Tables 17, 18, and 19 present the top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR compared with those of the approaches proposed by Park et al. and FRAN proposed by Saul et al. for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2 projects after removing the tangled changes, respectively. We notice the performance of our SUPLOCATOR and the baseline approaches are slightly increased. For example, for bugs in Eclipse JDT, the MRR and MAP of SUPLOCATOR are increased from 0.67 to 0.68, and 0.39 to 0.40, respectively. We use Wilcoxon rank-sum test (1945) with Bonferroni correction to test whether the performance of SUPLOCATOR in the cleaned datasets is significantly different with its performance in the original datasets. Notice we use Wilcoxon rank-sum test instead of Wilcoxon signed-rank test since the cleaned datasets contains less changes than the original datasets. And the statistical test shows that the performance difference is not statistically significant at the confidence

Table 18 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those for the baseline approaches for bugs in Eclipse SWT project after removing the tangled changes

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.57**	**0.72**	**0.74**	**0.64**	**0.33**
$Park^{Structure}$	0.05	0.06	0.09	0.06	0.03
$Park^{Change}$	0.05	0.15	0.20	0.11	0.04
$Park^{Content}$	0.03	0.05	0.05	0.02	0.00
$Park^{Name}$	0.06	0.14	0.18	0.10	0.04
$Park^{Sel}$	0.02	0.05	0.09	0.04	0.02
$Park^{Dev}$	0.04	0.06	0.08	0.03	0.02
$Park^{Pack}$	0.04	0.07	0.06	0.02	0.01
FRAN	0.37	0.58	0.64	0.45	0.24

The best performance is highlighted in bold

Table 19 Top-k accuracies (k = 1, 5, and 10), MRR and MAP for SUPLOCATOR compared with those for the baseline approaches for bugs in Equinox p2 project after removing the tangled changes

Approaches	Top-1	Top-5	Top-10	MRR	MAP
SUPLOCATOR	**0.39**	**0.50**	**0.52**	**0.44**	**0.23**
$Park^{Structure}$	0.06	0.11	0.15	0.08	0.05
$Park^{Change}$	0.02	0.04	0.09	0.04	0.01
$Park^{Content}$	0.00	0.00	0.00	0.00	0.00
$Park^{Name}$	0.04	0.06	0.09	0.05	0.02
$Park^{Sel}$	0.03	0.05	0.09	0.05	0.02
$Park^{Dev}$	0.00	0.04	0.06	0.04	0.05
$Park^{Pack}$	0.00	0.02	0.05	0.02	0.00
FRAN	0.23	0.42	0.48	0.31	0.15

The best performance is highlighted in bold

level of 95 % across the three projects. Thus, removing tangled changes has small and statistically insignificant impact on the performance of SUPLOCATOR.

Also, from Tables 17, 18, and 19, SUPLOCATOR still outperform the baseline approaches by a good margin. Wilcoxon signed-rank test with Bonferroni correction shows that the improvements of SUPLOCATOR over the baseline approaches are statistically significant at the confidence level of 95 % across the three projects.

Notice that in Table 19, the top-1, 5, and 10 accuracies, MRR, and MAP for $Park^{Content}$ are all zero. We manually check the results, and we find after removing the tangled changes, the nodes in the content similarity graph are isolated from one another. Since $Park^{Content}$ would recommend the nearest neighbors of the starting node, if nodes in the graph are isolated from one another, then there are no nearest neighbors. Thus, $Park^{Content}$ cannot recommend any change locations with this content similarity graph which causes all the values of the evaluation metrics to be zeroes.

7.3 Impact on the token size in content similarity graph

To create content similarity graphs, by default, we set the minimum token size parameter $token$ to 40. Here, we investigate the performance of SUPLOCATOR with different

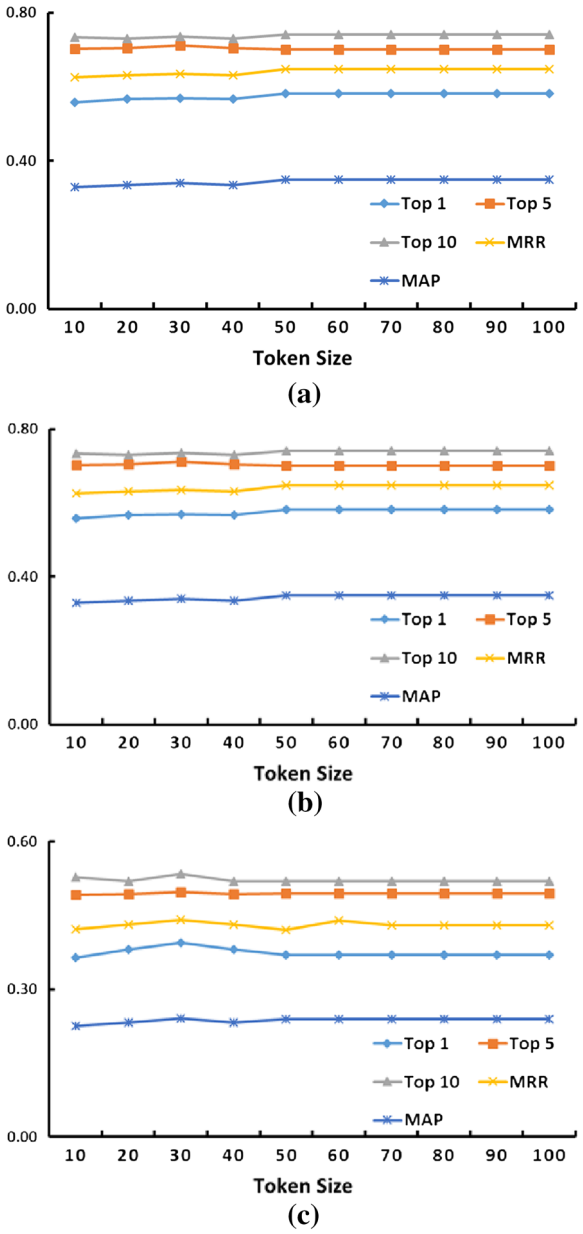

Fig. 4 Top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different *token* values for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2

token values. We vary the value of *token* from 10 to 100, and report the top-k accuracies (k = 1, 5, and 10), MRR, and MAP scores. Figure 4 presents the top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different *token* values for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2. We notice that for different values of *token*, the performance of SUPLOCATOR is stable. For example, for Eclipse JDT,

Table 20 Completeness@K (K = 5, 10, and 20) for SUPLOCATOR compared with those for the baseline approaches for bugs in Eclipse JDT project

Approaches	Completeness@5	Completeness@10	Completeness@20
SUPLOCATOR	0.47	0.52	0.56
$Park^{Structure}$	0.08	0.09	0.10
$Park^{Change}$	0.07	0.10	0.13
$Park^{Content}$	0.01	0.01	0.01
$Park^{Name}$	0.07	0.10	0.02
$Park^{Sel}$	0.02	0.05	0.07
$Park^{Dev}$	0.01	0.02	0.05
$Park^{Pack}$	0.02	0.03	0.05
FRAN	0.32	0.38	0.45
RW^{Method}	0.07	0.13	0.21
$RW^{Containment}$	0.34	0.43	0.50
$RW^{Inheritance}$	0.45	0.49	0.51
RW^{Change}	0.18	0.24	0.32
$RW^{Content}$	0.40	0.42	0.43
RW^{Name}	0.24	0.31	0.39

the MAP of SUPLOCATOR varies from 0.38 to 0.39. Thus, in practice, the value of *token* has small impact on the performance of SUPLOCATOR.

7.4 Completeness of SUPLOCATOR's recommendation

Here, we also investigate the completeness of SUPLOCATOR's recommendation. SUPLOCATOR returns a ranked list of change locations, and we would like to investigate the proportion of ground truth of change locations which are correctly suggested in the top-k results. Tables 20, 21, and 22 present the $Completeness@K$ scores for SUPLOCATOR compared with the baseline approaches for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2 projects. On average across the three projects, SUPLOCATOR achieves $Completeness@5$, $Completeness@10$, and $Completeness@20$ scores of 0.37, 0.41, and 0.44, respectively. SUPLOCATOR outperforms all the baseline approaches. Among the 14 baseline approaches, we notice $RW^{Inheritance}$ shows the best performance, and on average across the three projects, SUPLOCATOR improves the $Completeness@5$, $Completeness@10$, and $Completeness@20$ scores of $RW^{Inheritance}$ by 8.50, 10.55, and 14.38 %, respectively. Statistical testing shows that the improvements of SUPLOCATOR over the baseline approaches are significant at the confidence level of 95 % across the three projects.

7.5 Precision-related versus completeness-related fitness functions

In our previous section, we use precision-related fitness functions such as MAP, MRR, and top-20 accuracy in our SUPLOCATOR. Notice that our approach is meant to be a

Table 21 Completeness@K (K = 5, 10, and 20) for SUPLOCATOR compared with those for the baseline approaches for bugs in Eclipse SWT project

Approaches	Completeness@5	Completeness@10	Completeness@20
SUPLOCATOR	0.37	0.41	0.44
$Park^{Structure}$	0.04	0.05	0.06
$Park^{Change}$	0.08	0.09	0.10
$Park^{Content}$	0.02	0.02	0.03
$Park^{Name}$	0.06	0.10	0.14
$Park^{Sel}$	0.05	0.09	0.12
$Park^{Dev}$	0.01	0.02	0.05
$Park^{Pack}$	0.04	0.07	0.09
FRAN	0.25	0.29	0.33
RW^{Method}	0.02	0.05	0.10
$RW^{Containment}$	0.27	0.35	0.38
$RW^{Inheritance}$	0.34	0.37	0.38
RW^{Change}	0.12	0.17	0.23
$RW^{Content}$	0.24	0.27	0.29
RW^{Name}	0.24	0.27	0.29

Table 22 Completeness@K (K = 5, 10, and 20) for SUPLOCATOR compared with those for the baseline approaches for bugs in Equinox p2 project

Approaches	Completeness@5	Completeness@10	Completeness@20
SUPLOCATOR	0.28	0.31	0.34
$Park^{Structure}$	0.06	0.07	0.08
$Park^{Change}$	0.03	0.05	0.06
$Park^{Content}$	0.00	0.00	0.00
$Park^{Name}$	0.02	0.02	0.03
$Park^{Sel}$	0.02	0.03	0.04
$Park^{Dev}$	0.02	0.03	0.05
$Park^{Pack}$	0.03	0.04	0.06
FRAN	0.19	0.24	0.29
RW^{Method}	0.05	0.08	0.15
$RW^{Containment}$	0.18	0.23	0.26
$RW^{Inheritance}$	0.24	0.26	0.27
RW^{Change}	0.16	0.21	0.25
$RW^{Content}$	0.24	0.26	0.27
RW^{Name}	0.18	0.19	0.22

recommendation tool. The goal is to find as many relevant locations as possible in the top-k recommended locations. For such setting, precision-related measures (such as top-k accuracy, MRR, and MAP) are more important than completeness-related measures (such as Completeness@k). Thus, to further improve the precision-related measures, we use precision-related fitness functions in our SUPLOCATOR.

Here, we also investigate the performance of SUPLOCATOR by using the completeness-related objective functions such as Completeness@20, and we denote SUPLOCATOR with Completeness@20 as fitness function by SUP^{COMP}.

Moreover, since MAP and Completeness@20 are likely to be conflictive, we also leverage multi-objective GA to consider these two fitness functions together, and we denote it as SUP^{MULTI}. In SUP^{MULTI}, we use a multi-objective GA (i.e., vNSGA-II (Zhang and Li 2007) to optimize the values of some parameters (e.g., the weights). Since we have two fitness functions to be maximized, SUPLOCATOR will find a set of Pareto optimal solutions (Deb 2001) defined in Definition 3.

Definition 2 (**dominance and Pareto optimal solutions**) A set of parameters Par_i *dominates* another set of parameters Par_j if and only if the values of the two objective functions (i.e., MAP and Completeness@20) satisfy:

$$\begin{aligned} &\text{MAP}\left(Par_i\right) > \text{MAP}\left(Par_j\right) \textit{ and } \text{Comp@20}\left(Par_i\right) \geq \text{Comp@20}\left(Par_j\right) \\ &\qquad\qquad\qquad\qquad \textit{or} \\ &\text{MAP}\left(Par_i\right) \geq \text{MAP}\left(Par_j\right) \textit{ and } \text{Comp@20}\left(Par_i\right) > \text{Comp@20}\left(Par_j\right). \end{aligned}$$

A set of parameters Par_i is Pareto optimal if and only if no other set of parameters dominates it in the feasible region, i.e, no other set of parameters Par_j exists that would improve the MAP, without worsening Completeness@20 scores, and vice versa.

There would be a number of solutions which are Pareto optimal. We further reduce these solutions by selecting the following subset:

$$ParSet_{selected} = \underset{Par_i \in Pareto}{argmax}\ MAP\left(Par_i\right) \times Comp@20\left(Par_i\right). \tag{9}$$

In other words, we select solutions from the Pareto optimal set which has the highest product of MAP and Completeness@20 scores. We randomly pick a solution from this set as the near optimal solution.

Table 23 presents the top-k accuracies (k=1, 5, and 10), MRR, MAP, Completeness@k (k=5, 10, and 20) for SUPLOCATOR using different fitness functions for Eclipse JDT, Eclipse SWT, and Equinox p2 datasets. We notice that among the three approaches, i.e., SUP^{MAP}, SUP^{COMP}, and SUP^{MULTI}, our SUP^{MAP} still achieves the best performance, although the difference among the three approaches are small.

Notice SUP^{MAP} aims to maximize the precision-related measures, and it still achieves the acceptable completeness@k scores. Among the three projects, SUP^{MAP} achieves similar Completeness@k scores as the other two approaches. And SUP^{MAP} slightly improves over the other two approaches in terms of precision-related measures such as top-k accuracies, MRR, and MAP. Thus, in practice, we recommend developers to use MAP as the fitness function.

Table 23 Top-k accuracies (k = 1, 5, and 10), MRR, MAP, Completeness@k (C@k) (k = 5, 10, and 20) for SUPLOCATOR using different fitness functions for Eclipse JDT, Eclipse SWT, and Equinox p2 datasets

Pro.	Approaches	Top-1	Top-5	Top-10	MRR	MAP	C@5	C@10	C@20
JDT	SUP^{MAP}	0.61	0.74	0.78	0.67	0.39	0.47	0.52	0.56
	SUP^{COMP}	0.59	0.74	0.78	0.66	0.37	0.47	0.51	0.56
	SUP^{MULTI}	0.60	0.73	0.78	0.66	0.39	0.47	0.52	0.56
SWT	SUP^{MAP}	0.57	0.70	0.73	0.73	0.33	0.37	0.41	0.44
	SUP^{COMP}	0.51	0.68	0.73	0.59	0.32	0.35	0.40	0.45
	SUP^{MULTI}	0.55	0.70	0.72	0.62	0.33	0.36	0.40	0.42
Equ.	SUP^{MAP}	0.38	0.49	0.52	0.43	0.23	0.28	0.31	0.34
	SUP^{COMP}	0.38	0.48	0.50	0.43	0.23	0.28	0.31	0.34
	SUP^{MULTI}	0.36	0.49	0.51	0.42	0.23	0.28	0.31	0.34

In terms of precision, our DelPredictor does not perform as well as Correa and Sureka's approach, however in terms of recall, DelPredictor shows *much better* performance compared to the baseline approaches. Neither our approach nor Correa et al.'s approach is ready for full automation yet (i.e., automatic deletion of questions without human intervention) since the precisions of these approaches are still relatively low.

7.6 Genetic algorithms versus other optimization approaches

In SupLocator, we use GA to combine multiple lists outputted by the six types of CRGs. There are other techniques to combine multiple lists, for example, learning to rank (Liu 2009), averaging voting (Zhou 2012), or maximum voting (Zhou 2012).

7.6.1 Learning to rank

We aim to rank the methods changed in the supplementary bug fixes above the methods not changed in the supplementary bug fixes. Given a bug B, we denote a method changed in the supplementary bug fixes as v^+, and a method not changed in the supplementary bug fixes as v^-. Then for any pairs $\langle v^+, v^- \rangle$ in B, we aim to minimize the following loss function:

$$Loss = \sum_{\langle v^+, v^- \rangle} \left\| CRank\left(v^+\right) \leq CRank\left(v^-\right) \right\|. \tag{10}$$

In the above equation, $CRank(v^+)$ and $CRank(v^-)$ refer to the composite rank score of the method v^+ and v^-. The definition of composite rank score can be found in Definition 1. The value of $\|CRank(v^+) \leq CRank(v^-)\|$ is 0 if $CRank(v^+) \leq CRank(v^-)$ else it is -1. In such a way, we cast the ranking problem as an optimization problem. A number of learning to rank techniques have been proposed (c.f., Freund et al. 2003; Burges et al. 2005; Joachims 2002; Tsai et al. 2007), and in this paper,

we use one of the state-of-the-art learning to rank techniques that comes with a publicly available implementation (i.e., RankSVM Joachims 2002) to tune the weights. RankSVM uses SVM to learn a hyperplane that separates v^+ from v^-.

7.6.2 Average voting and maximum voting

We denote the rank of a method v produced by the six random walk components as $Rank_1(v)$, $Rank_2(v), \ldots, Rank_6(v)$. Then, for average voting, the average rank score of v [denoted as $Average(v)$] is computed as:

$$Average(v) = \frac{\sum_{i=1}^{6} Rank_i(v)}{6}. \tag{11}$$

For maximum voting, the maximum rank score of v [denoted as $Max(v)$] is computed as:

$$Max(v) = Maximum\left(Rank_1(v), Rank_2(v), \ldots, Rank_6(v)\right). \tag{12}$$

7.6.3 Results

Table 24 presents the top-k accuracies (k = 1, 5, and 10), MRR, MAP, and Completeness@k (k = 5, 10, and 20) for SUPLOCATOR compared with RankSVM, average voting, and maximum voting for Eclipse JDT, Eclipse SWT, and Equinox p2 datasets. We notice still our SUPLOCATOR achieves the best performance. And Wilcoxon signed-rank test with Bonferroni correction shows that the improvements of SUPLOCATOR over RankSVM, average voting, and maximum voting are statistically significant at the confidence level of 95 % across the three projects.

Table 24 Top-k accuracies (k = 1, 5, and 10), MRR, MAP, Completeness@k (C@k, k = 5, 10, and 20) for SUPLOCATOR compared with RankSVM, averaging voting (Average), and maximum voting (Maximum) for Eclipse JDT, Eclipse SWT, and Equinox p2 datasets

Pro.	Approaches	Top-1	Top-5	Top-10	MRR	MAP	C@5	C@10	C@20
JDT	SUP.	0.61	0.74	0.78	0.67	0.39	0.47	0.52	0.56
	RankSVM	0.50	0.62	0.71	0.62	0.34	0.41	0.46	0.50
	Average	0.32	0.45	0.58	0.51	0.29	0.38	0.44	0.48
	Maximum	0.34	0.47	0.60	0.49	0.32	0.40	0.47	0.49
SWT	SUP.	0.57	0.70	0.73	0.73	0.33	0.37	0.41	0.44
	RankSVM	0.48	0.61	0.66	0.64	0.28	0.33	0.38	0.42
	Average	0.44	0.56	0.62	0.60	0.25	0.30	0.34	0.40
	Maximum	0.45	0.57	0.64	0.62	0.26	0.33	0.37	0.42
Equ.	SUP.	0.38	0.49	0.52	0.43	0.23	0.28	0.31	0.34
	RankSVM	0.32	0.43	0.47	0.37	0.19	0.24	0.28	0.31
	Average	0.29	0.40	0.45	0.34	0.16	0.22	0.26	0.29
	Maximum	0.30	0.42	0.47	0.34	0.17	0.23	0.28	0.31

7.7 Threats to validity

Threats to internal validity relates to errors in our code and experiment bias. We have double-checked our code, still there could be errors that we did not notice. Also, in this paper, we use a longitudinal data setup to simulate the actual usage of SUPLOCATOR. In practice, we can only use bugs reported before to build a model, and we can not use future bugs to build the model.

Another threat to internal validity relates to the GA configuration and the randomness involved. In this paper, we use a single point crossover operator with crossover probability of 0.35, and a random mutation operator with mutation probability of 0.08. We use this setting since we find it can achieve reasonable results for our SUPLOCATOR. It is unclear whether these configurations perform equally well for other datasets and whether other more optimal settings exist. Another threat is due to the randomness of GA. To deal with the randomness issue, we run SUPLOCATOR 10 times following the recommendations made by Arcuri and Briand (2011), Liu et al. (2010), Canfora et al. (2013), and Xia et al. (2014a).

Threats to external validity relate to the generalizability of our results. We have analyzed 2543 bugs with supplementary fixes from three open source projects. In the future, we plan to reduce this threat further by analyzing even more bugs from additional projects.

Threats to construct validity refer to the suitability of our evaluation metrics. We use top-k prediction accuracies, MRR, and MAP which are also used by past studies to evaluate the effectiveness of various automated software engineering techniques (Thongtanunam et al. 2015; Zhou et al. 2012; Rao and Kak 2011; Xia et al. 2014c; Tamrawi et al. 2011; Saha et al. 2013). Thus, the threat to validity is mitigated.

Another threat to construct validity corresponds to the selection of fitness function. In this paper, we mainly consider the precision-related fitness functions such as MAP, MRR, and top-k accuracy. To reduce the threat due to the selection of fitness functions, in the discussion section, we also consider the completeness-related fitness functions, and we also use a multi-objective GA to consider both precision-related and completeness-related fitness functions. The experimental results show that still SUPLOCATOR with MAP as the fitness function achieves the best performance.

8 Related work

8.1 Studies on change recommendation

Park et al. is the first to study change recommendation problem for supplementary bug fixes (Park et al. 2014). They propose the usage of multiple CRGs to generalize a number of previous change recommendation approaches (Ying et al. 2004; Zimmermann et al. 2005; Hassan and Holt 2004; Nguyen et al. 2010). However, they find most of the change recommendation approaches can not work well for supplementary bug fixes. Our study extends their work by:

(1) We propose a more accurate change recommendation approach by leveraging a random walk technique. Different from Park et al.'s approach, in our approach,

the recommended nodes do not need to be one hop away from the initial nodes (which represent methods that are fixed in the initial bug fix) in one of the graphs.

(2) We propose a new composite approach that integrates the analysis of multiple CRGs leveraging a search-based algorithm. Park et al.'s approach perform a simple boosting step that uses edges in multiple CRGs that connect two adjacent nodes to predict additional buggy methods. The edge combinations that perform better in a training data are selected and applied on the test data. Our approach, on the other hand, performs a different heuristics; it merges different recommendations that are produced by the random walk components by learning a set of weights leveraging a GA.

There have been a number of other studies on change recommendations (Ying et al. 2004; Zimmermann et al. 2005; Hassan and Holt 2004, 2006; Herzig and Zeller 2011; Nguyen et al. 2010). Zimmermann et al. and Ying et al. leverage association rule mining techniques to predict change locations based on historical co-change patterns mined from version control systems (Ying et al. 2004; Zimmermann et al. 2005). The average precision and recall scores of Zimmermann et al.'s approach are 0.33 and 0.29, while the scores for Ying et al.'s approach are around 0.4 and 0.2. Hassan and Holt propose different kinds of change recommendation heuristics, and their approach uses a combination of historical co-change patterns and structural dependencies in a program to predict change locations (Hassan and Holt 2004, 2006). In a later work, Malik and Hassan propose various change propagation heuristics such as history heuristic, containment heuristic, call use depends heuristic, and code ownership heuristic, to improve the performance of change location prediction (Malik and Hassan 2008). Nguyen et al. propose FixWizard which leverages code clones to suggest code peers that should be changed together (Nguyen et al. 2010). Herzig and Zeller propose GENEVA to predict change locations by utilizing temporal rules mined from a project's version history which capture key features of the change process followed by the project (Herzig and Zeller 2011).

Our work is related to the above studies, however we focus on the application of change recommendation for supplementary bug fixes. The above-mentioned approaches were evaluated using general software commit data, i.e., commits are grouped into a set of transactions, a model is built on a subset of commits in a transaction, and this model is used to predict the entities that are changed in the remaining commits of the transaction. A prior study by Park et al. show that applying change recommendation for supplementary bug fixes is a much more difficult problem which tests the limit of existing change recommendation approaches (Park et al. 2014).

8.2 Studies on change management

Robillard and Murphy propose concern graph which abstracts the implementation details of a concern by storing the key structure implementing a concern (Robillard and Murphy 2002), and it can be automatically extracted from the source code or an intermediate representation of a program. Concern graph consider seven types of structural dependency relationships among the methods, fields, and classes. For

example, if method A contains a call that can be bind to method B, A and B has the call relationship. Different from Robillard and Murphy's study, in this paper, we not only consider structural dependency relationships in a program, but also other kinds of dependency relationship such as name similarity and content similarity.

Kawrykow and Robillard propose an approach named DiffCat to detect non-essential changes in version histories (Kawrykow and Robillard 2011). DiffCat rollbacks rename refactorings, and matches differences against a number of templates to identify non-essential changes. Our study is different from theirs since the target of these two studies are different; in this paper, we focus on recommending methods that require supplementary fixes given a set of methods fixed by an initial bug fix. Furthermore, rolling back rename refactorings and identifying differences that fit DiffCat templates cannot help to identify additional buggy methods given a set of initial changes.

Tao et al. perform a large-scale empirical study on the impact of code changes during the software development process (Tao et al. 2012). They find that it is an important but yet a difficult task to determine a changes risk, and there is a crucial need to assess a changes quality (e.g., its completeness and consistency). Their study also highlights that we still lack good tool supports. Our study complements theirs by proposing a tool to recommend methods for supplementary bug fixes.

Herzig and Zeller perform an empirical study on the impact of tangled changes, and they find that up to 15 % of all bug fixes are tangled changes (Herzig and Zeller 2013). Based on the observation, they propose a tool to untangle the tangled changes. In this paper, we use their proposed tool to untangle the tangled changes, to clean the datasets. Our experiments show that there is no statistically significant difference in the performance of SUPLOCATOR before and after the clean-up.

8.3 Reopened bug report prediction

There have been a number of studies on reopened bug report prediction. Reopened bug reports refer to the bugs which are incompletely fixed, and thus supplementary fixes are required. Shihab et al. propose the problem of reopened bug report prediction and develop an approach that is based on machine learning algorithms to predict reopened bugs in three open source projects (Shihab et al. 2013). Zimmermann et al. study reopened bugs in Microsoft Windows operating system to investigate the different reasons for bug report reopenings (Zimmermann et al. 2012). Xia et al. propose a composite approach REOPENPREDICTOR which combines textual and meta features extracted from bug reports to achieve a better performance as compared to Shihab et al.'s approach (Xia et al. 2014b). An et al. perform an empirical study on the reopened bugs with supplementary fixes, and their prediction model can achieve a precision between 72.2 and 97 %, and a recall between 47.7 and 65.3 % (An et al. 2014). Our work nicely complements the above studies: the above studies predict bug reports that will be reopened and thus requiring supplementary fixes, while our study predicts locations that developers need to change when they create supplementary fixes.

8.4 Search-based software engineering

There have been a number of studies on search-based software engineering (Harman and Jones 2001; Harman et al. 2012; Panichella et al. 2013; Le Goues et al. 2012). Harman and Jones propose the concept of search-based software engineering and they demonstrate how to reformulate a SE problem as a search-based problem (Harman and Jones 2001; Harman et al. 2012). Panichella et al. propose a search-based GA which tunes latent Dirichlet allocation parameters (Panichella et al. 2013). Le Goues et al. propose *GenProg*, which leverages genetic programming to automatically repair defects in software projects (Le Goues et al. 2012). Wang et al. propose a search-based approach named EvaClone which finds suitable configurations by leveraging GA for clone detection (Wang et al. 2013). Lohar et al. propose a search-based approach which identifies the best configuration for a trace retrieval technique that recovers traceability links between software artifacts (e.g., requirement to code, requirement to design, etc.) (Lohar et al. 2013). Panichella et al. propose CODEP which combines the results from multiple classifiers to improve the performance of cross-project defect prediction (Panichella et al. 2014).

In this work, we also use a search-based technique to learn a composition of multiple random walk components. Different from the above mentioned studies, we address a different problem namely change recommendation for supplementary bug fixes.

9 Conclusion and future work

In this paper, we propose an effective change recommendation approach SUPLOCATOR for supplementary bug fixes. SUPLOCATOR first extracts six CRGs from the source code and the version history of a target software system, and performs random walk on the six CRGs. Then, SUPLOCATOR leverages GA to combine the multiple results outputted by the random walk components. To investigate the benefits of SUPLOCATOR, we perform experiments on Eclipse JDT, Eclipse SWT, and Equinox p2. The experimental results show that on average SUPLOCATOR can achieve top-1, top-5, and top-10 accuracies, MRR, and MAP of 0.51, 0.65, 0.67, 0.58 and 0.32 for the three projects, which improve the best approaches proposed by Park et al. by a substantial margin.

In the future, we plan to evaluate SUPLOCATOR with datasets from more software projects, and develop a better technique which could improve the recommendation accuracy further. We also plan to integrate the concern graph proposed by Robillard and Murphy (2002) to further improve the performance of SUPLOCATOR.

Acknowledgements We would like to thank Jihun Park for providing us the datasets and source code used in their study (Park et al. 2014). This research was supported by NSFC Program (Nos. 61602403 and 61402406) and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2015BAH17F01.

Appendix

Impact of the GA configurations

In this paper, by default, we set the population size (PopSize) as 500 chromosomes, the maximum number of generations (MaxGen) as 200, the crossover probability

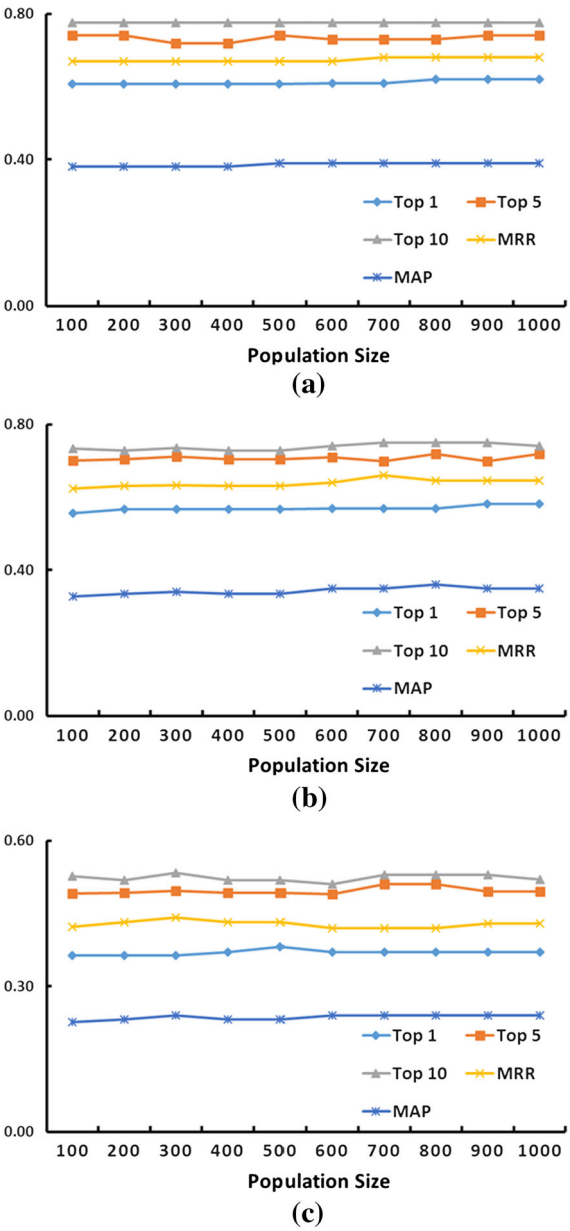

Fig. 5 Top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different population size for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2

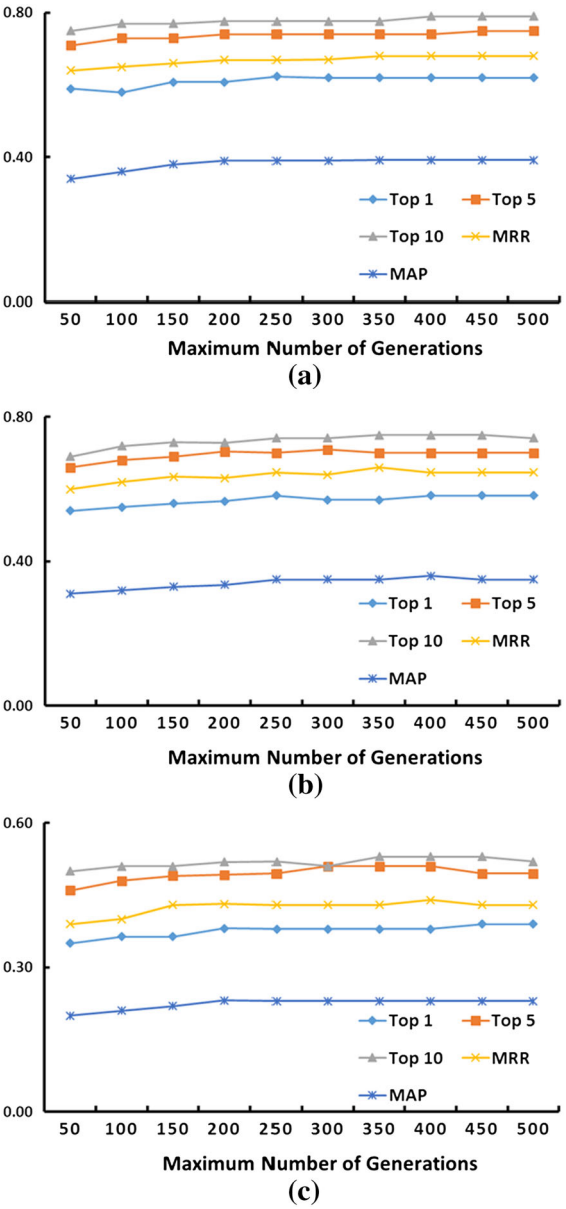

Fig. 6 Top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different maximum number of generations for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2

(CrossProb) as 0.35, and the mutation probability (MutProb) as 0.08. Here, we investigate the performance of SUPLOCATOR with other GA configurations (i.e., other values of the four parameters). For each parameter p, we keep the default values of the other three parameters, and we investigate the performance of SUPLOCATOR with various values of p. We vary the value of PopSize from 100 to 1000, MaxGen from 50 to 500, CrossProb from 0.10 to 0.50, and MutProb from 0.05 to 0.14.

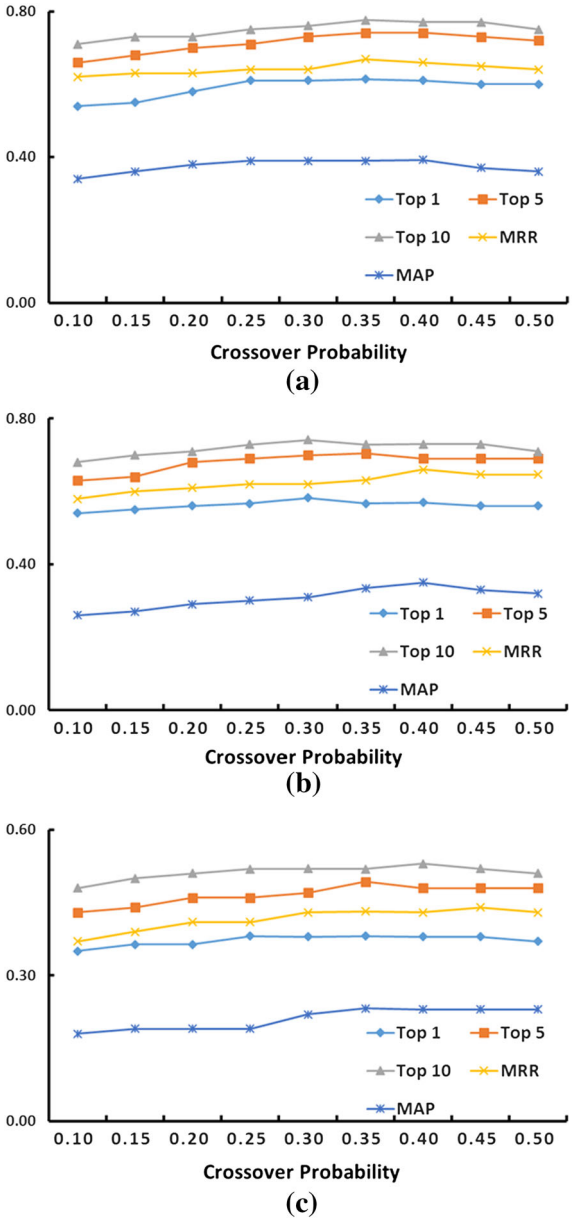

Fig. 7 Top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different crossover probabilities for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2

Figure 5 presents the top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different population size for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2, respectively. We notice the performance of SUPLOCATOR is stable when we vary population size from 100 to 1000. Thus, in practice, the value of population size has a limited impact to the performance of SUPLOCATOR.

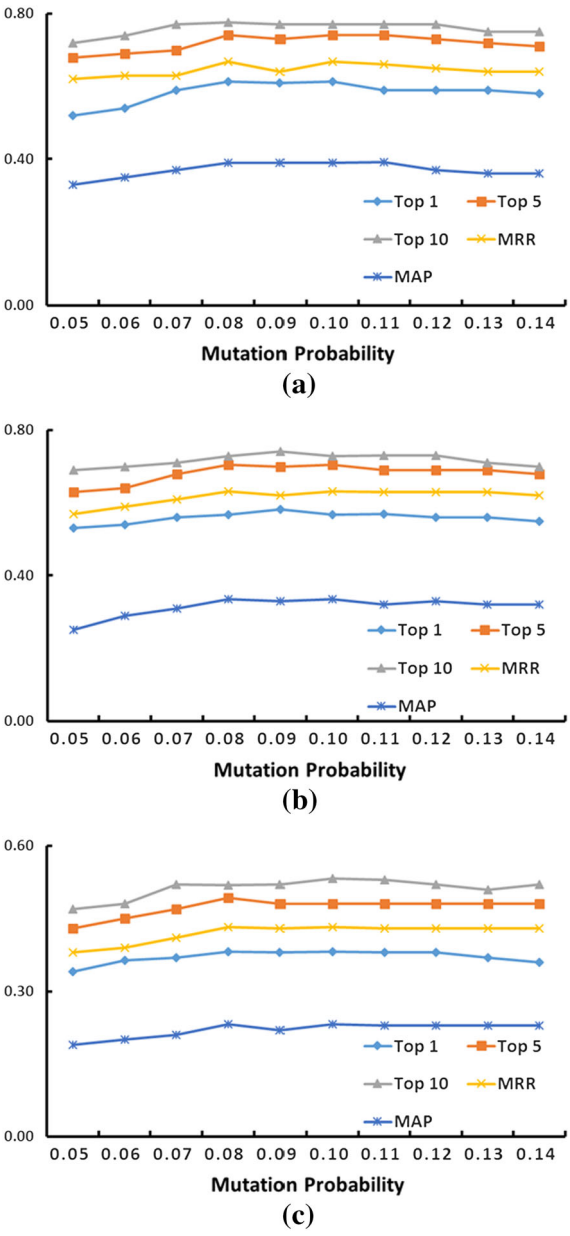

Fig. 8 Top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different mutation probabilities for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2

Figure 6 presents the top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different maximum number of generations for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2, respectively. We notice when we increase the maximum number of generations from 50 to 200, the performance of SUPLOCATOR is slightly increased. When we increase the maximum number of generations from 200 to 500, the performance of SUPLOCATOR is relatively stable. Note that the larger the maximum number

of generations, the more time it takes to run our SUPLOCATOR. Thus, in practice, we recommend users to set the maximum number of generations as 200.

Figure 7 presents the top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different crossover probabilities for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2, respectively. We notice when we increase the crossover probability from 0.10 to 0.35, the performance of SUPLOCATOR is increased. When we increase the crossover probability from 0.35 to 0.50, the performance of SUPLOCATOR is decreased. If the crossover probability is set too large (e.g., 0.5) or too small (e.g., 0.1), the performance of SUPLOCATOR is low. And the performance of SUPLOCATOR is relatively stable when we set the crossover probability in the range of 0.3 and 0.4. Thus, in practice, we recommend users to set the crossover probability between 0.3 and 0.4.

Figure 8 presents the top-k accuracies (k = 1, 5, and 10), MRR and MAP of SUPLOCATOR with different mutation probabilities for bugs in Eclipse JDT, Eclipse SWT, and Equinox p2, respectively. We notice when we increase the mutation probability from 0.05 to 0.08, the performance of SUPLOCATOR is increased. When we increase the mutation probability from 0.08 to 0.14, the performance of SUPLOCATOR is decreased. If the mutation probability is set too small (e.g., 0.05) or too large (e.g., 0.14), the performance of SUPLOCATOR is low. And the performance of SUPLOCATOR is relatively stable when we set the mutation probability in the range of 0.07 and 0.09. Thus, in practice, we recommend user to set the mutation probability between 0.07 and 0.09.

References

Abdi, H.: Bonferroni and Šidák corrections for multiple comparisons. In: Salkind, N.J. (ed.) Encyclopedia of Measurement and Statistics. https://www.utdallas.edu/~herve/Abdi-Bonferroni2007-pretty.pdf (2007). Accessed 12 Aug 2016

An, L., Khomh, F., Adams, B.: Supplementary bug fixes vs. re-opened bugs. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 205–214. IEEE (2014)

Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1–10. IEEE (2011)

Baeza-Yates, R., Ribeiro-Neto, B., et al.: Modern Information Retrieval, vol. 463. ACM Press, New York (1999)

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.: Learning to rank using gradient descent. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 89–96. ACM, New York (2005)

Canfora, G., De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S.: Multi-objective cross-project defect prediction. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 252–261. IEEE (2013)

Dagenais, B., Hendren, L.: Enabling static analysis for partial Java programs. ACM Sigplan Notices **43**, 313–328 (2008)

Dai, N., Shokouhi, M., Davison, B.D.: Learning to rank for freshness and relevance. In: Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 95–104. ACM, New York (2013)

Deb, K.: Multi-objective Optimization Using Evolutionary Algorithms, vol. 16. Wiley, Hoboken (2001)

Freund, Y., Iyer, R., Schapire, R.E., Singer, Y.: An efficient boosting algorithm for combining preferences. J. Mach. Learn. Res. **4**, 933–969 (2003)

Goldberg, D.E., Holland, J.H.: Genetic algorithms and machine learning. Mach. Learn. **3**(2), 95–99 (1988)

Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)

Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. (CSUR) **45**(1), 11 (2012)
Hassan, A.E., Holt, R.C.: Predicting change propagation in software systems. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings, pp. 284–293. IEEE, Washington, DC (2004)
Hassan, A.E., Holt, R.C.: Replaying development history to assess the effectiveness of change propagation tools. Empir. Softw. Eng. **11**(3), 335–367 (2006)
Herzig, K., Zeller, A.: Mining cause–effect-chains from version histories. In: 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE), pp. 60–69. IEEE, Washington, DC (2011)
Herzig, K., Zeller, A.: The impact of tangled code changes. In: 2013 10th IEEE Working Conference on Mining Software Repositories (MSR), pp. 121–130. IEEE (2013)
Hopkins, W.G.: A New View of Statistics. Will G. Hopkins (1997)
Joachims, T.: Optimizing search engines using clickthrough data. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 133–142. ACM, New York (2002)
Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(7), 654–670 (2002)
Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 351–360. ACM, New York (2011)
Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. J. ACM **46**(5), 604–632 (1999)
Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
Liu, T.Y.: Learning to rank for information retrieval. Found. Trends Inf. Retr. **3**(3), 225–331 (2009)
Liu, Y., Khoshgoftaar, T.M., Seliya, N.: Evolutionary optimization of software quality modeling with multiple repositories. IEEE Trans. Softw. Eng. **36**(6), 852–864 (2010)
Lohar, S., Amornborvornwong, S., Zisman, A., Cleland-Huang, J.: Improving trace accuracy through data-driven configuration and composition of tracing features. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 378–388. ACM, Saint Petersburg (2013)
Malik, H., Hassan, A.E.: Supporting software evolution using adaptive change propagation heuristics. In: IEEE International Conference on Software Maintenance, 2008. ICSM 2008, pp. 177–186. IEEE (2008)
Meffert, K., Rotstan, N., Knowles, C., Sangiorgi, U.: Jgap-java genetic algorithms and genetic programming package. http://jgap.sourceforge.net/ (2011). Accessed 12 Aug 2016
Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J., Nguyen, T.N.: Recurring bug fixes in object-oriented programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 315–324. ACM (2010)
Page, L., Brin, S., Motwani, R., Winograd, T.: The Pagerank Citation Ranking: Bringing Order to the Web (1999)
Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A.: How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 522–531. IEEE Press, Piscataway (2013)
Panichella, A., Oliveto, R., De Lucia, A.: Cross-project defect prediction models: L'union fait la force. In: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week, pp. 164–173. IEEE (2014)
Park, J., Kim, M., Ray, B., Bae, D.H.: An empirical study of supplementary bug fixes. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pp. 40–49. IEEE Press, Washington, DC (2012)
Park, J., Kim, M., Bae, D.H.: An empirical study on reducing omission errors in practice. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 121–126. ACM (2014)
Poshyvanyk, D., Marcus, A., Ferenc, R., Gyimóthy, T.: Using information retrieval based coupling measures for impact analysis. Empir. Softw. Eng. **14**(1), 5–32 (2009)
Rao, S., Kak, A.: Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 43–52. ACM, New York (2011)
Robillard, M.P.: Automatic generation of suggestions for program investigation. ACM SIGSOFT Softw. Eng. Notes **30**, 11–20 (2005)

Robillard, M.P., Murphy, G.C.: Concern graphs: finding and describing concerns using structural program dependencies. In: Proceedings of the 24th International Conference on Software Engineering, pp. 406–416. ACM, New York (2002)

Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 345–355. IEEE (2013)

Saul, Z.M., Filkov, V., Devanbu, P., Bird, C.: Recommending random walks. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Fundations of Software Engineering, pp. 15–24. ACM, New York (2007)

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W.M., Ohira, M., Adams, B., Hassan, A.E., Ki, M.: Studying re-opened bugs in open source software. Empir. Softw. Eng. **18**(5), 1005–1042 (2013)

Sivanandam, S., Deepa, S.: Introduction to Genetic Algorithms. Springer, Berlin (2007)

Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 365–375. ACM, New York (2011)

Tao, Y., Dang, Y., Xie, T., Zhang, D., Kim, S.: How do software engineers understand code changes? An exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, p. 51. ACM, New York (2012)

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

Tsai, M.F., Liu, T.Y., Qin, T., Chen, H.H., Ma, W.Y.: Frank: a ranking method with fidelity loss. In: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 383–390. ACM, New York (2007)

Wang, T., Harman, M., Jia, Y., Krinke, J.: Searching for better configurations: a rigorous approach to clone evaluation. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 455–465. ACM, New York (2013)

West, D.B., et al.: Introduction to Graph Theory, vol. 2. Prentice Hall, Upper Saddle River (2001)

Wilcoxon, F.: Individual comparisons by ranking methods. Biom. Bull. **1**, 80–83 (1945)

Xia, X., Feng, Y., Lo, D., Chen, Z., Wang, X.: Towards more accurate multi-label software behavior learning. In: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week, pp. 134–143. IEEE (2014a)

Xia, X., Lo, D., Shihab, E., Wang, X., Zhou, B.: Automatic, high accuracy prediction of reopened bugs. Autom. Softw. Eng. **22**(1), 75–109 (2014b)

Xia, X., Lo, D., Wang, X., Zhang, C., Wang, X.: Cross-language bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, pp. 275–278. ACM (2014c)

Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 54–65. ACM, New York (2005)

Ying, A.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C.: Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. **30**(9), 574–586 (2004)

Zhang, Q., Li, H.: MOEA/D: a multiobjective evolutionary algorithm based on decomposition. IEEE Trans. Evol. Comput. **11**(6), 712–731 (2007)

Zhou, Z.H.: Ensemble Methods: Foundations and Algorithms. CRC Press, Boca Raton (2012)

Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 14–24. IEEE (2012)

Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S.: Mining version histories to guide software changes. IEEE Trans. Softw. Eng. **31**(6), 429–445 (2005)

Zimmermann, T., Nagappan, N., Guo, P.J., Murphy, B.: Characterizing and predicting which bugs get reopened. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 1074–1083. IEEE, Piscataway (2012)