Not all classes are created equal: Toward a Recommendation System for Focusing Testing

Segla Kpodjedo*, Filippo Ricca**, Philippe Galinier* and Giuliano Antoniol*
{segla.kpodjedo, philippe.galinier}@polymtl.ca, filippo.ricca@disi.unige.it, antoniol@ieee.org

* SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada
** Unitá CINI at DISI, University of Genoa, Italy.

ABSTRACT

When evolving an object oriented system, one relevant question is the following: given a finite amount of resources, what are the most critical classes on which testers should focus their attention? In this paper, we propose a new way for identifying critical classes: classes often changed and playing a key role in the system. We rely on error correcting graph matching (ECGM) and random walks to associate each class with a pair of values representative of the frequency of changes and the class overall connectivity.

With those two metrics, we have a grid for assessing the criticality of any class in the system. Classes with high values in both metrics should be identified and reported to developers, as a residual error in those classes will more likely deeply impact the whole system. We show the feasibility of the proposed approach by studying the Mozilla suite evolution over the year 2007.

Keywords: Software evolution, Software Testing, Error-Correcting Graph Matching (ECGM) algorithm, Random walks.

1. INTRODUCTION

Recommendation systems are useful Software Engineering (SE) tools able to present information items that are of interest to product managers and/or developers. In a typical recommendation system, experts provide inputs and the system aggregates recommendations which direct the Human decisions. This happened, for example, in Tapestry [6]. Tapestry, one of the first recommendation systems, was a system able to manage received mails/documents by using the opinions of other readers (collaborative filtering).

Nowadays, more modern recommendation systems are employed in different SE contexts. The recommendation system proposed in [1] tries to answer the important question "*who should fix this bug?*". When a new bug appears in a bug repository, the system — based on a machine learning algorithm — suggests a small number of developers suitable to fix it. Other recommendation systems focus on a common discovery task in large software systems: *given a particular function, find related functions.* This problem of mining related API calls has attracted a lot of interest and several approaches to resolve it have been proposed in literature (e.g., [13]).

Most of the work in the Object-Oriented (OO) unit and integration testing focus on the important problem of minimizing test effort while assuring the achievement of a testing goal (i.e., a coverage criteria). In integration testing, the goal is often to devise optimal test strategies [2, 10] aiming at minimizing the number of drivers and stubs. On the other hand, in presence of inheritance, an unit testing goal could be to minimize the number of test cases to be developed by reusing test cases of the parent class to test the derived classes [7].

However, these approaches do not fully address the problem of how deeply a class should be tested, leaving the manager by himself in this strategic decision. Consider a product manager and/or tester who want to improve the quality of a large software evolved over time. Once she applies the approaches presented in [2, 7, 10], she will obtain an optimal test or retest strategy. Still she may need to know what are the key classes to focus on, the subset of critical classes where a slipped fault could potentially propagate extensively across the system.

There are some ongoing research about predicting fault-prone modules, i.e. modules more likely to contain the larger number of faults in next releases of a software system. In [14], the authors use a negative binomial regression model based on factors such as LOC (Lines Of Code) and information drawn from a MR (Modification Request) database to predict the fault-prone files in subsequent releases. Each MR typically contains information such as the size of the file, its status (whether new or not), its number of faults in earlier releases, its number of changes etc. Information processed from the available MR database is then used to identify the top 20% most fault-prone files.

On a higher architectural level, [11] targets components of large systems and consider metrics based on dependency graphs between software components and code churns (i.e., changes) of the components between subsequent versions. Dependence metrics are defined between components through several architectural layers while churn metrics for a component involve delta LOCs, changed files and number of changes in the module between two versions. Using logistic regression techniques, the authors then estimate post release failures.

However, the management infrastructure and strong prior

RSSE'08, November 10, 2008, Atlanta, Georgia, USA.

history information required for those techniques may make them difficult to use. Plus, the choice of metrics, especially in [11], may lack strong theoretical background. Finally, the identified fault-prone files generally constitute an important subset of the targeted system (20% in [14], potentially thousands of files for large systems) and while each file or module may be assigned a probability of defect, they can hardly be ranked with respect to their impact on the software should they be faulty.

1.1 Our proposal

Inspired by the work of Saul et al. [13], we believe that random walks can lead to recommendations on which classes a manager/tester should focus limited testing resources to minimize the risk of a fault propagating consequences across an entire application. In particular, we propose the application of a Google-inspired Error-Correcting Graph Matching (ECGM) algorithm to identify critical classes.

Based on the two following assumptions: (i) "the most important classes" must be tested more deeply and (ii) frequently changed classes are the most complex and then the most fault-prone [8], our algorithm computes, for each class, the PageRank [12] and the Evolution Cost [9]. Random walks are implemented using a basic PageRank algorithm, which, considering the relations among classes, measures for each class its relative importance in a System and assigns it a numerical weighting. The Evolution Cost quantifies how much the class and its relations (i.e., aggregations, associations and generalizations) changed in a time frame. The recommendation to the product manager and/or tester is a scatter-plot (X-axis=Evolution Cost, Y-axis=PageRank) indicating the "critical classes" (i.e., important classes that are changed frequently in the past). By interpreting the class position in the plane (Evolution Cost, PageRank) as a dominance relation [5], we obtain a partial order between classes, and the Pareto front, further reducing the manager and developers effort to identify most critical classes. Other interesting facts can be derived such as classes with high variation over time of their PageRank value, i.e classes gaining or losing dramatically "importance" in the system.

We applied our algorithm to 24 subsequent snapshots of Mozilla reverse engineered class diagram. First, we collected Mozilla snapshots every 15 days over the last year (2007). Second, with available tools, we reverse engineered the class diagrams and reformulated the class diagram evolution problem as an ECGM problem. Then, we computed the Evolution Cost and recovered traceability for each class present in the last snapshot (Dec 30, 2007). Finally, we plotted the Page Rank and the Evolution Cost to find the region containing the "critical classes".

It is worth underlying that at this stage of our research, we do not have empirical evidence of a correlation between criticality of classes and other unwanted characteristics such as error proneness, severity of defects or consequence of defects in critical classes. We believe that our conjecture of a relation between class criticality and possible consequences of errors is reasonable and, as such, testing effort should focus on the most critical classes. However, further work is needed to verify if our conjecture is also supported by empirical data.

This paper is organized as follows. Section 2 summarizes our ECGM algorithm and describe in detail the meaning of PageRank and Evolution Cost. Section 3 reports preliminary results of the ECGM application to Mozilla. Section 4 proposes some initial ideas of a tool — Eclipse plug-in —, based on our approach, that we want to implement. Finally, Section 5 concludes and outlines future work.

2. THE ECGM ALGORITHM AND ITS APPLICATION TO SOFTWARE EVOLUTION

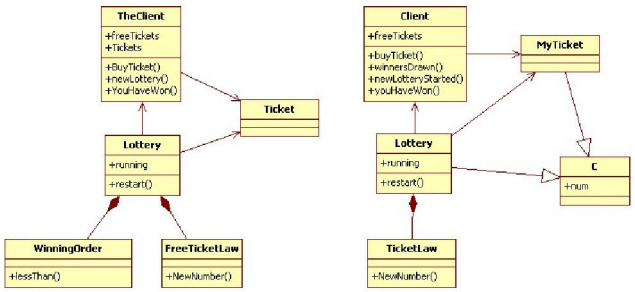

Figure 1: Example of class diagrams to be matched

Class diagrams can be thought of as labeled graphs with nodes being the classes and edges representing the relations between classes. Labels on edges can specify the type of the edge (i.e., association, aggregation or inheritance) while node labels can specify properties such as the class name. Given two class diagrams of the same software at different stages of evolution - as illustrated in Figure 1 - we are interested into finding a mapping, i.e., a correspondence, between them. A solution can be represented as a correspondence table linking classes of the two diagrams and specifying added or deleted classes. Added and deleted classes are classes without a linked element in one of the two class diagrams.

To apply ECGM algorithms to study software evolution, we envisage the following steps. First, software artifacts, class diagrams in our case, are represented as graphs. Once the graphs are available, we build a mapping between them via an ECGM algorithm. We are interested in finding an optimal or a near optimal mapping, i.e., a mapping minimizing a cost function representative of the problem at hand.

We resort on meta-heuristics, more precisely a Tabu search algorithm, to search for optimal or near optimal solutions. In order to guide the search toward regions containing promising solutions and speed up computation, we consider local and global information about the nodes of the graphs. Our algorithm exploits similarity of nodes from different graphs, based on their number of edges and hierarchical position in the overall graph structure. This latter heuristic is implemented via a Google inspired PageRank algorithm [9].

2.1 The ECGM Model

A graph with labels from two finite alphabets of symbols $\sum_V$ (vertices' labels) and $\sum_E$ (edges' labels) is defined as a triple (V, L_V, L_E) where V is the finite set of elements, called nodes or vertices; $L_V : V \rightarrow \sum_V$ is the node labeling function and $L_E : V \times V \rightarrow \sum_E$ is the edge labeling function. Let $g_1 = (V_1, L_{V1}, L_{E1})$ and $g_2 = (V_2, L_{V2}, L_{E2})$ be two graphs. An ECGM from g_1 to g_2 is a bijective function $m : \hat{V}_1 \rightarrow \hat{V}_2$ where $\hat{V}_1 \subseteq V_1$, $\hat{V}_2 \subseteq V_2$. We say $x \in \hat{V}_1$ is *matched* to node $y \in \hat{V}_2$ if $m(x) = y$. Furthermore, any

node from $V_1 - \hat{V}_1$ is said to be *deleted* from g_1, and any node from $V_2 - \hat{V}_2$ is said to be *inserted* in g_2 under m. More formal definitions of ECGM can be found in [4]. In essence, any ECGM can be thought of as a set of edit operations that transform a given graph g_1 into another graph g_2. We call *node matching* a couple $(n_1, m(n_1)) \in (\hat{V}_1 \times \hat{V}_2)$. An ECGM solution, called *matching*, is then a set of those couples with the constraint that a node is matched to at most one node. Penalties are assigned to every distortion found by the solution. We distinguish edit operations leading to distortions into *node/edge deletions*, *node/edge insertions* and *node/edge matching errors*. Given $(n_1, m(n_1))$, a *node matching error* refers to the dissimilarity between n_1 and $m(n_1)$. *Edge matching* refer to any edge replacement from $\hat{V}_1 \times \hat{V}_1$ to $\hat{V}_2 \times \hat{V}_2$. Two types of *edge matching errors* are to be considered: replacing a missing edge by an existing edge (*structural error*) or replacing one edge by another (*label error*).

2.2 Modeling software evolution as an ECGM

A straightforward mapping of a class diagram into a graph may disregard elements of classes such as the class name, the attributes and methods. However, these are important elements in software evolution and they have to be modeled as node properties and matched by the ECGM algorithm. For each class, we considered the following class characteristics: the class name and the signatures of attributes and methods.

As an example, when we consider the graphs of Figure 1, an optimal solution is to match "TheClient" to "Client", "Ticket" to "MyTicket", "Lottery" to "Lottery", "FreeticketLaw" to "TicketLaw" (potential *node matching errors*); to delete "WinningOrder" (*node deletion*) and any of its adjacent edges (*edge deletion*); to insert "C" (*node insertion*) and any of its adjacent edges (*edge insertion*). As for *edge matching*, in the example, the relation between "Lottery" and "Ticket" is substituted by the one between "Lottery" and "MyTicket".

Each class is assigned an Evolution Cost expressing its amount of change between the versions. This value is the sum of internal and structural changes. We refer to internal changes as changes occurring for the name of the class, the set of attributes and the class signature. Refactoring of a class name is measured using the Levenshtein distance while similarity in the attributes and signatures (between two versions), is computed using a Jaccard index[1]. Jaccard index for the attributes sets is computed considering attribute type, visibility and name collapsed in a string; much in the same way is computed methods sets similarity; the Levenshtein distance and the two similarities are then combined into the internal change measure.

Structural changes are refered to as changes occuring within the class and affecting its relations with its peers. Three cases are possible from one version `A` to another `B`. A modified relation with a connected class (an association in `A` becoming an aggregation in `B`, for instance), a relation missing in `B` but existing in `A` (the class no longer uses another class), a new relation with another class (a relation missing in `A` but existing in `B`).

[1]The Jaccard index is a statistic providing a measure of similarity defined as the size of the intersection divided by the size of the union of the sets.

2.3 Tabu Search and PageRank Algorithms

Our ECGM algorithm rely on a Tabu Search (TS) algorithm guided by global information on the nodes from the PageRank algorithm and local node features such as the number of edges. PageRank [3], one of the main components behind the first versions of Google, basically measures the relative importance of each element of a hyperlinked set and assigns it a numerical weighting. In essence, the more references (incoming arcs) an element (vertex) gets from other elements (preferably important), the more importance it deserves. Note that if a node is the only reference of a very important node, it might be valued more important than another node getting some references from low ranked nodes. Further details can be found in [12]. Using PageRank, we can compute a metric representative of global structure for each vertex of a given graph. Once combined with local metric, this metric allows us to have a more accurate assessment of the structural similarity of two nodes; structural similarity that is used to guide the TS search and provide hints on class criticality to developers.

3. ECGM APPLICATION TO MOZILLA

To demonstrate the feasibility of our approach, we apply it to a well-known, industrial strength, open-source system: the Mozilla suite.

Mozilla is an open-source suite implementing a Web browser and other tools such as mailers and newsreaders. It was ported on almost all software and hardware platforms. It is developed mostly in C++, with C code accounting for only a small fraction of the system. All components in the Mozilla suite have been extracted (this corresponds to the CVS "suite" checkout tag). More details can be found at http://developer.mozilla.org/ in the Mozilla Source Code (CVS) section.

The Mozilla class diagram evolved over time, increasing the number of classes and relations. At the end of December 2007, classes were about 9,000 with 23,000 relations. In essence, our ECGM algorithm has to match graphs of about 10,000 nodes and 25,000 edges to compute the Evolution Cost.

We extracted the class diagrams of Mozilla twice per month over the last year (2007). Subsequently, class diagrams were mapped into graphs as outlined in the background work section.

Then, we run our ECGM algorithm coded in C++ — compiled with g++ and run on a Linux Bi Processor Opteron 64-bit with 16 Gb RAM running Redhat Advanced Server version 4 — starting from the last snapshot (December 2007) going toward the first (January 1, 2007). The average time for a matching was of about 382 seconds measured by the Unix time utility.

From the 23 ECGM results between the subsequent pairs of consecutive snapshots we collected:

- threads of matched classes, i.e., classes that maintained a (almost) stable structure of relations (associations, inheritances and aggregations) with other classes through 2007.

- the Evolution Cost for each class through the year — calculated as explained in the background session and using the threads of matched classes.

- the PageRank for each class in the last snapshot (December 2007).

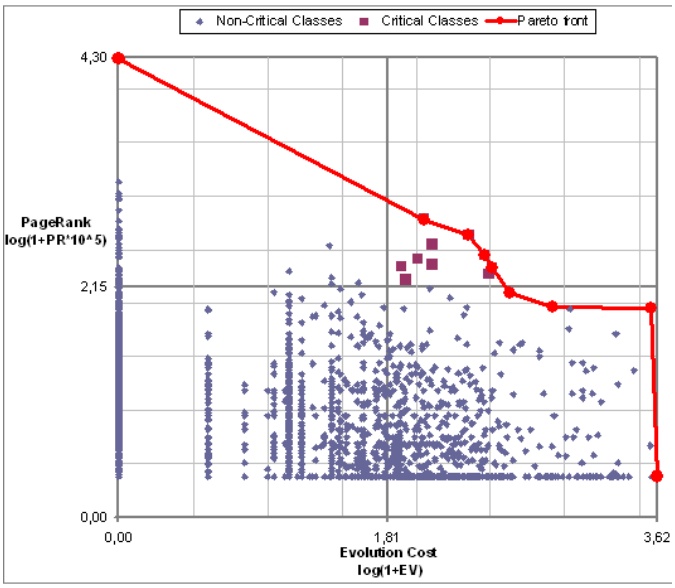

Figure 2: PageRank / Evolution Cost View of Mozilla classes - Pareto front drawn in red.

As summarized in previous sections, a PageRank score measures the relative importance of each class in a system accounting for overall structure of relations among classes. Meanwhile, the Evolution Cost quantifies the amount of change for a class and its relations in a time frame.

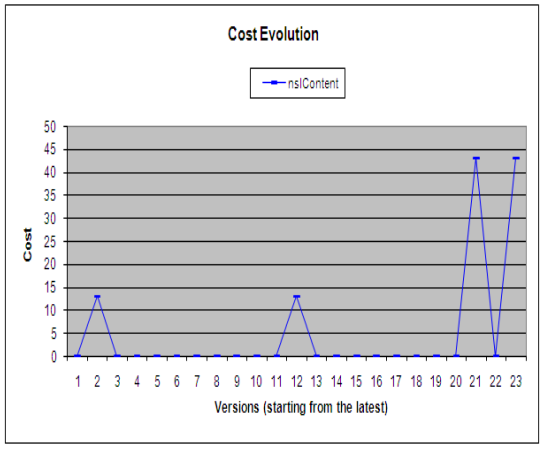

Figure 3: Cost Evolution of the Mozilla class `nsIContent` - From Dec 2007 to Jan 2007.

Figure 2 shows a scatter-plot, named *Evolution cost and PageRank view*, that plots Evolution Cost (X-axis) and PageRank (Y-axis)[2] through the last year of Mozilla's evolution. Points near to the origin represent classes "not much important" and changed rarely. Points far from the origin (upper right corner) represent classes with high PageRank and high Evolution Cost. These classes, potentially "critical classes" are marked in red using the square symbol; they should be carefully considered by tester/manager to distribute resources and assign testing effort. Using a Pareto front as presented in Figure 2, some important classes such as `nsISupports` - which is the highest ranked in Mozilla - can also be pinpointed.

For each class of the system, the tester/manager can visualize the trend of Evolution Cost and PageRank by means of a line chart. Such graphics can help the tester/manager to infer useful information of a specific class. For example Figures 3 and 4 show, respectively, the trend of Evolution Cost and PageRank for the class `nsIContent` during the year 2007 (23 snapshots). Graphics are drawn from December to January, thus first point (X-axis) stands for December 30, 2007 while point 24 is January 1, 2007. The class `nsIContent` is in both the critical zone and the Pareto Frontier (second point starting from the left). It has been changed 4 times in 2007 (see Figure 3) and when we checked in the code to understand, for example, evolution from 01/01/2007 to 15/01/2007, we discovered that `nsIContent` adds a call to the class `nsIURI`. The PageRank value of `nsIContent` (see Figure 4) increased throughout 2007 from a little under 5*10e-03 to a little above 6*10e-03, with some small jumps roughly matching the observable changes of the class in Figure 3. In the case of `nsIContent`, considering the high rank of this class, more testing effort should be allocated on the class itself and its callers every time this class changed. The point is that a high ranked class plays an important role in the application and if it changes quite often or dramatically, there are more chances to have residual bugs potentially impacting various system functionalities.

There are other interesting facts when one looks at the PageRank evolution of a class. From one version to another, some classes may have a dramatic increase or decrease in their PageRank value. A good example is the class `Pool` which went from 7,3*10e-03 on 01/15/2007 to 0,01*10e-03 on 02/01/2007. After investigation, we discovered that most of the classes that used to call `Pool` just disappeared from the system. A manager should carefully consider these kinds of abrupt evolution facts and make sure they are the result of a wanted restructuring operation.

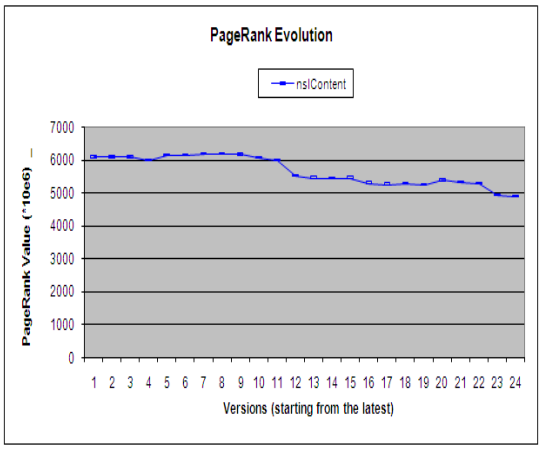

Figure 4: Page Rank Evolution of the Mozilla class `nsIContent`. - From Dec 2007 to Jan 2007.

[2]the plotted values are the logarithms of Evolution Cost and PageRank.

4. AN ECLIPSE PLUGIN

Currently, we are considering the possibility of building an Eclipse plug-in for our approach. We have chosen Eclipse as development toolkit because of its popularity. Furthermore, the Eclipse Plug-in Development Environment (PDE) provides a nice environment for creating plug-ins and integrating them with the Eclipse Platform.

When the plug-in is launched, during the testing or monitoring phase of a project, our ECGM algorithm is executed on the project's versions specified by the user and the *Evolution cost and PageRank view* is computed. The tester/developer could use it during the testing phase to concentrate his/her testing effort on the "critical classes" and to keep under control the overall software evolution. Points of the *Evolution cost and PageRank view* should be clickable: clicking the right button of the mouse on the class of interest should popup a menu with several items. The tester could then select one item (*local functionality*) for visualizing information such as (i) line charts trends of Evolution Cost and PageRank for the class, (see Figures 3 and 4); (ii) its code in the java editor; (iii) the list of its associated test cases; (iv) the list of directly related classes; (v) some code coverage measures, etc.

An extended version of this menu with *global functionality* could be also useful to help the project manager, usually interested in aspects of monitoring, analysis and project estimation, to have information on the overall project and to take important and strategic decisions about the testing phase. Useful higher level menu functionality could be visualizing things like (i) the threads of matched classes; (ii) the new classes (i.e. classes added in the last project version); (iii) the classes with high variation of their PageRank value; (iv) the distribution of test cases per class; (v) the code coverage measures for the "critical classes", etc.

5. CONCLUSION

In this paper we have proposed a recommendation system, based on ECGM and PageRank. Our recommendation system aims at identifying critical classes in an OO application. Critical classes are classes frequently subject to changes and with high connectivity with other classes. We do not have empirical evidence of a correlation between criticality of classes as defined in this paper and error proneness, severity or priority of defects; however, classes identified by our approach are indeed playing an important role in the software application and they are also frequently changed. For this reason we believe they should be tested more deeply.

To demonstrate the feasibility of implementing such a recommendation system, we applied the ECGM algorithm to the 2007 Mozilla evolution, identifying the critical classes.

Future work will be devoted to:

- studying correlation between our index and bug severity/priority. We surmise that severity and priority of bugs in the upper right classes are on average higher than classes on the lower left area;
- implementing the Eclipse plug-in, only sketched here, supporting our approach.

6. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering – ICSE 2006*, pages 361–370, 2006.

[2] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, 2003.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[4] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18(9):689–694, 1997.

[5] K. Deb. Multi-objective genetic algorithms: Problem difficulties and construction of test problems. *Evolutionary Computation*, 7(3):205–230, 1999.

[6] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[7] M. Harrold, J. McGregor, and K. Fitzpatrick. Incremental testing of object-oriented class structures. *Software Engineering, 1992. International Conference on*, pages 68–80, 1992.

[8] T. Illes-Seifert and B. Peach. Exploring the relationship of a file's history and its fault-proneness: An empirical study. In *Proceedings of Testing: Academic & Industrial Conference*, page (to appear), 2008.

[9] S. Kpodjedo, P. Galinier, and G. Antoniol. A google-inspired error correcting graph matching algorithm. Technical Report EPM-RT-2008-06, avalaible at https://web.soccerlab.polymtl.ca/repos/soccerlab/technical-reports/EPM-2008-06.pdf, Ecole Polytechnique de Montreal, 06 2008.

[10] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.-H. Durand. Testing levels for object-oriented software. *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 136–145, 2000.

[11] T. B. N. Nagappan. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *First International Symposium on Empirical Software Engineering and Measurement ESEM2007*, 2007.

[12] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[13] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 15–24, New York, NY, USA, 2007. ACM.

[14] E. J. W. Thomas J. Ostrand and R. M. Bell. Automating algorithms for the identification of fault-prone files. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 219–227. ACM New York, NY, USA, 2007.