Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks

Joseph Lawrance[1,2], Rachel Bellamy[2], Margaret Burnett[1], Kyle Rector[1]

[1]Oregon State University
School of EECS
Corvallis, Oregon 97331
{lawrance,burnett,rectorky@eecs.oregonstate.edu}

[2]IBM T.J. Watson Research
19 Skyline Drive
Hawthorne, New York 10532
rachel@us.ibm.com

ABSTRACT

In recent years, the software engineering community has begun to study program navigation and tools to support it. Some of these navigation tools are very useful, but they lack a theoretical basis that could reduce the need for ad hoc tool building approaches by explaining what is fundamentally necessary in such tools. In this paper, we present PFIS (Programmer Flow by Information Scent), a model and algorithm of programmer navigation during software maintenance. We also describe an experimental study of expert programmers debugging real bugs described in real bug reports for a real Java application. We found that PFIS' performance was close to aggregated human decisions as to where to navigate, and was significantly better than individual programmers' decisions.

Author Keywords

Information foraging, debugging, software maintenance

ACM Classification Keywords

D.2.5 [Software Engineering]: Testing and Debugging; H.1.2 [Information Systems]: User/Machine Systems—Human factors

INTRODUCTION

Is navigating code like navigating the web? Do programmers navigate source code in search of a bug in the same way that people navigate the web in search of particular information? Can the behavior of programmers navigating source code be described using the same theories and models that have been used to describe web navigation?

In software maintenance, code navigation is a central task. The importance of navigation to programming tasks such as maintenance and debugging is beginning to become recognized [7, 15]. One study showed that programmers spend 35% of their on-line time navigating code [13]. Several research efforts, mostly in the software engineering community, have begun research to try to solve this problem [5, 7, 16, 24, 28]. The research falls mainly into two categories: development of new tools without a theoretical basis, and derivation of new descriptive theories ground-up from data.

However, we believe that an existing theory, namely information foraging theory [21], can improve upon both of these approaches. In particular, we propose that information foraging theory can provide the foundations needed for tool development. This theory is more attractive than building new theories particular to navigation, because it has been empirically shown to be a good predictive theory in its own domain (namely, web browsing). Thus, it has mature roots, and in that domain it has become widely accepted and established as a useful basis for tool development. We therefore decided to investigate how information foraging theory might model programmers' navigation behavior in debugging and maintenance.

In this paper, we present PFIS (Programmer Flow by Information Scent), a model and accompanying algorithm to predict programmers' dynamic navigation behavior during program maintenance tasks. Information foraging theory uses the concept of scent to determine where someone will go when searching for information related to their goal. We believe that programmers may be information foragers when debugging, because research has shown that when debugging, programmers create hypotheses and then search for information to verify (or refute) these hypotheses. Furthermore, we conjecture that such hypotheses are linguistically related to the words in the bug reports. According to these assumptions, the bug report defines the programmer's goal and the scent they are seeking.

As with information foraging models that have been used to model web behavior, the PFIS model takes into account both the source code's topology (analogous to links on web pages) and its "scent." Using these concepts, PFIS predicts that programmers will visit the source code with the highest scent in relation to the bug report. The PFIS algorithm builds on the WUFIS (Web User Flow by Information Scent) algorithm [3], adapting and extending the approach used in WUFIS to model programmer navigation during software maintenance.

We also present an experiment that investigates the extent

PFIS can model the places to which programmers navigate to during two software maintenance tasks. We compare the PFIS results to competing possible predictors of where these programmers would navigate: a model based on word "scent" without making use of topology, a model based on topology without scent, and other programmers' navigation patterns both pairwise and aggregated. We finally consider elements of program navigation not modeled by our information foraging model, and their relationships with information foraging.

BACKGROUND AND RELATED WORK

Information Foraging

Information foraging theory emerged from the PARC labs in the mid-90's, led by Pirolli and Card [21]. Inspired by appeals in the psychological literature for ecological accounts of contextually dependent human behaviors, information foraging theory offered a new perspective for those attempting to develop theoretical accounts of HCI that could be applied to tool design. Ecological theories contrast with information processing theories such as GOMS, that at the time did not account for effects of context.

Information foraging theory is based on optimal foraging theory, a theory of how predators and prey behave in the wild. In the domain of information technology, the predator is the person in need of information and the prey is the information itself. Using concepts such as "patch," "diet" and "scent," information foraging theory describes the most likely pages (patches) a web-site user will visit in pursuit of their information need (prey), by clicking links containing words that are a close match to (smell like) their information need. The scent of information comes from the linguistic relationships between words expressing an information need and words contained in links to web pages. The predator/prey model, when translated to the information technology domain, has been shown to mathematically model which web pages human information foragers select on the web [20], and therefore has become useful as a practical tool for web site design and evaluation [4, 18, 23, 29].

The work described in this paper builds on the WUFIS algorithm (Web User Flow by Information Scent) [3, 4], an empirically validated algorithm approximating information foraging theory as defined by Pirolli et al's SNIF-ACT model [22]. The advantage of WUFIS over SNIF-ACT is that WUFIS can be readily applied in new contexts, whereas SNIF-ACT is a fully functioning cognitive model, and must be customized for each information foraging context being studied. The WUFIS algorithm encodes link quality as a function of its match to the users' information need. Someone is more likely to select the link on a page that appears to have the highest probability of eventually leading them to the page best matching their information need. Flow refers to users surfing through the web site moving from page to page by clicking on the highest scent links on the page, and is modeled using spreading activation over the link topology. Scent is computed as a function of the term frequency of words in a query and the term frequency of words in or near a link. The usability of a site for a particular query is determined by running WUFIS to obtain the probable number of users that would reach each page by following cues that best match the query.

This work in this paper also builds upon the work of [17]. Programmers debugging code may be information foragers in that they form hypotheses and then hunt for information to verify these hypotheses. A study of programmers attempting to fix bugs found an interword' correlation between the bug report and the set of classes visited by the programmers [17]. That work presented evidence based on a static view of these interword' correlations, but did not present a model per se. This paper, in contrast, contributes a model of information foraging, together with an algorithm, that takes into account programmers' dynamic program navigation behavior.

Program Navigation and Maintenance

In recent years, the software engineering community has begun to study program navigation and tools to support it. For example, Robillard et al. studied navigation qualitatively with a controlled experiment of 5 graduate student developers [24], from which they derived a descriptive theory. Their theory focuses on the importance of methodical investigation; it does not suggest information foraging principles, but it is consistent with them. However, as they point out, elements of their experimental design may have encouraged methodical investigation. Fundamental differences from our work are that our use of theory focuses on the potential cause of methodical investigation (namely, information foraging) rather than on its presence and effect, our study does not create new theories but rather investigates the applicability of existing theory, and our theory is intended to be predictive rather than simply descriptive.

In Ko et al.'s investigation of developer behavior during software maintenance, the participants—student developers working in Eclipse with 9 source files—spent 35% of their time navigating source code [13]. Their surprising result made clear the importance of trying to understand how programmers go about navigating, and how to help them save time while doing so. This finding led to the development of a descriptive model of program understanding [16], which proposes that the cues (e.g., identifier names, comments, and documentation) in an environment are central to searching, relating, and navigating code in software maintenance and debugging. Although they did not investigate scent per se, their model is philosophically similar to information foraging theory.

DeLine et al. [8] conducted empirical work into problems arising in software developers' navigations through unfamiliar code. Their work turned up two major problems: developers needed to scan a lot of the source code to find the important pieces (echoing the finding of [13]), and they tended to get lost while exploring. These results inspired the idea to combine collaborative filtering and computational "wear" [10] from users' interaction histories into a concept they call "wear-based filtering".

A number of software engineering tools have begun to be developed that are also based on concepts of "togetherness" as defined by developer navigation/editing actions. Some of these approaches harvest simultaneous change information from source version control systems such as CVS, to obtain this type of information (e.g., [27, 31, 32] and others harvest togetherness directly from developer behavior logs (e.g., [7, 12, 25, 26, 28]). A system that is particularly pertinent is Hipikat [5] which remembers paths traversed by earlier members of a team, and uses hand-crafted textual similarity to support search for code-related artifacts. Evaluations showed that newcomers using Hipikat achieve results comparable in quality and correctness to those of more experienced team members.

These tools and analyses have not been grounded in theory, but their empirical success shows that they are useful. Our premise is that the information foraging model may explain why they are useful, and may usefully guide the development of future program maintenance and debugging tools.

THE PFIS ALGORITHM

To examine whether information foraging theory can predict the classes and methods programmers will visit, and the paths they will take as they navigate through code in search of relevant places to fix bugs, we created PFIS. PFIS is based upon the web user flow by information scent (WUFIS) algorithm [3], which combines information retrieval techniques with spreading activation. As WUFIS does for web path following, PFIS calculates the probability that a programmer will follow a particular "link" from one class or method in the source code to another, given a specific information need.

Consider the notion of *links* in the domain of program navigation. According to information foraging theory, the path an information forager will take is determined by the scent of proximal cues of a particular link in relation to their information need. In WUFIS, hyperlinks serve as the way information foragers navigate between pages, and thus the words in or near hyperlinks serve as proximal cues an information forager can use to choose among which links to follow in pursuit of a goal. In PFIS, we define a *link* to be any means a programmer can use to navigate directly in *one click* to a specific place in source code, excluding scrolling between methods within a class or browsing among classes within a package. Thus, the definition of links takes into account the features of the programming environment. As in hyperlinks, links in programs have proximal cues associated with them: for example, a link from a method definition to a method invocation includes the name of the object of the invoked method, the name of the invoked method, and the names of the variables passed in as parameters to the invoked method.

For example, the Eclipse Package Explorer and Outline views allow programmers to navigate from packages to classes to fields and methods (one click each). Eclipse also allows programmers to open definitions and search for references to a method, variable, or a type in one click. Therefore, packages link to member classes, classes link to their fields and methods, methods link to the methods they invoke, and variables link to their type.

Due to the many one-click links, program navigation has two fundamental differences from web page navigation. First, what counts as a link is well-defined in a web site, whereas every identifier in a program may be (and is, in Eclipse) associated with a link to some definition or use. Second, source code has a much denser "link" topology than web pages, so there are many more ways to navigate to the same place. Such differences meant that, to extend the ideas of WUFIS to program navigation required defining the notion of links in source code, finding ways to process them, and defining the terms in and near a link that should be used and how to compute the scent of a link.

Another difference between PFIS and WUFIS is that PFIS is necessarily more "real world" than WUFIS. Some assumptions/controls that simplify the problem domain were made when developing WUFIS into the Bloodhound usability service [4], but they are not viable for the domain of program navigation. This implementation is an important point of comparison for the work presented here, because it was used in a study validating the predictions made by the WUFIS algorithm.

These simplifying assumptions/controls were (1) to disallow the use of search, which is not a reasonable limitation to place on a programmer attempting to maintain software (in our study programmers could choose to go anywhere once they had looked at the bug report); (2) to have only one web page open at a time (in our study programmers could keep class files open in tabbed panes); (3) to give pages that did not have any links in them a link back to the starting page, which we could not do since we wanted to use PFIS on real-world software without modifying it; and (4) to remove the scent for links on the desired target document. This latter simplification was based on the assumption that people stopped searching when they reached the target and hence would not select any of the links on a target page. We cannot assume a target destination, because there is often no one "correct" target for a code maintenance task.

PFIS is summarized in Figure 1. We explain how each step is accomplished next.

Central to WUFIS is a description of the link topology of the web site, describing each link in terms of which page it is on, and which page it points to. For example, Figure 2 shows on the left four nodes, and the links between them. In WUFIS, the nodes are web pages; in PFIS the nodes are anything that is the destination of a link, e.g., method definitions, method invocations, variable definitions, etc. The link topology is described by the matrix on the right. For step 1 of the PFIS algorithm, to create the link topology of source code, we created an Eclipse JDT plugin [9] to traverse each class and method in each compilation unit, and

Algorithm PFIS (Programmer Flow by Information Scent)
Given: Bug report, body of source code
Returns: A vector containing for each package, class, method, and variable, the probability that a programmer will visit that area of source code given the bug report.

Step 1. Determine link topology of source code and store them in matrix T.
Step 2. Determine set of proximal cues around each link.
Step 3. Determine proximal scent of each link to the bug report, and store the resulting similarity scores in matrix PS.
Step 4. Normalize matrix PS so that each column sums to 1.00 (i.e., generate a column-stochastic matrix).
Step 5. Define the starting place (class or method) for the spreading activation, and create an entry vector E with the starting location given a 1.
Step 6. Calculate the probability of programmers going from the starting location to other documents by multiplying the entry vector E=A(1) by PS, to create an activation vector A(2).
Step 7. Repeat step 6 to simulate programmers traversing through the link topology. The final activation vector A(n), when normalized, contains for each location the expected probability that a programmer will visit that location given the bug report.

Figure 1: The PFIS algorithm.

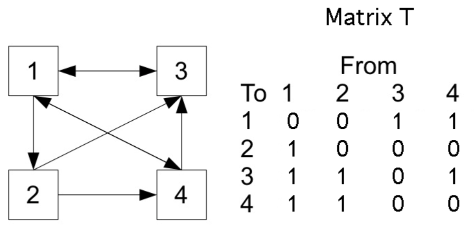
used the Java Universal Network/Graph Framework [11] to construct the link topology (adjacency matrix) **T**, which gives us the beginning **(i)** and end points **(j)** for each link that a programmer can follow.

Steps 2 and 3 determine the proximal scent of each link relative to the bug report (Figure 3). Proximal scent is the information foraging term referring to "scent" near the link. For step 2 of PFIS, we developed a special tokenizer for words in cues, so that CamelCase identifiers (e.g., "NewsItem.getSafeXMLFeedURL()") would be split into their constituent words ("news item get safe xml feed url"), and also employed a standard stemming algorithm on the constituent words.

For step 3 of PFIS, the scent is determined by the similarity of words in the bug report to the text that labels the link and in close proximity to the link. We used Lucene [6], an open-source search engine API to index the proximal cues (the text) associated with each link. Lucene uses TF-IDF [1], a technique commonly used in information retrieval to weight the importance of words in documents. For our purposes, we treated the bug report as the query, and the proximal cues of each link as a document. Lucene determined the cosine similarity of each link in relation to the bug report to determine the scent of each link. We used these results as weights for the edges in **T**, producing a proximal scent matrix **PS**.

In step 4, PFIS normalizes **PS** so that each column sums to 1, thus producing a column-stochastic matrix. In effect, each column contains the probability that a programmer will follow a link from one location to another. Thus, at the end of step 4, the proximal scent relative to the bug report has been calculated, reflecting the information foraging premise that links in the source with proximal cues close to the important words in the bug report will smell more strongly of the bug, and are thus more likely to be followed.

Steps 5, 6 and 7 simulate programmers navigating through the source code, following links based on scent. Spreading activation is an iterative algorithm used widely by HCI



From → To	1	2	3	4
1	0	0	1	1
2	1	0	0	0
3	1	1	0	1
4	1	1	0	0

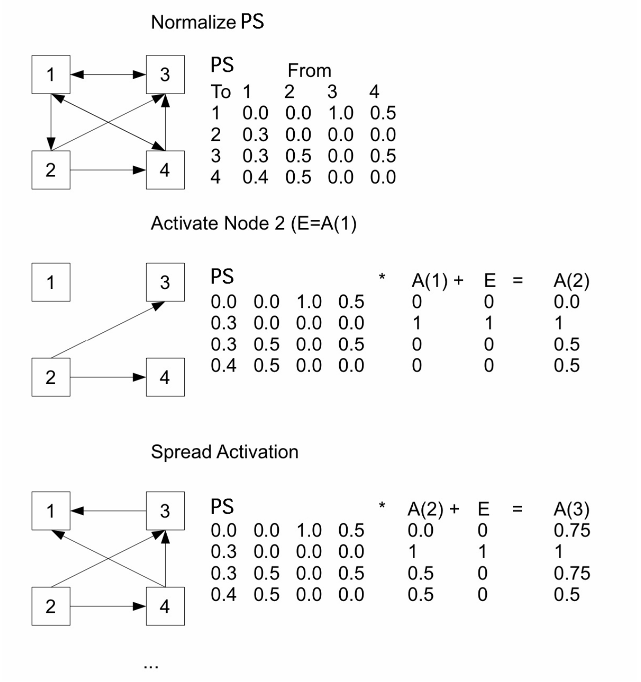
Figure 2: Link topology (adjacency) matrix 'T'

Matrix T $\xrightarrow{\text{apply weights to links and normalize}}$ Matrix PS

From → To	1	2	3	4
1	0	0	1	1
2	1	0	0	0
3	1	1	0	1
4	1	1	0	0

From → To	1	2	3	4
1	0.0	0.0	1.0	0.5
2	0.3	0.0	0.0	0.0
3	0.3	0.5	0.0	0.5
4	0.4	0.5	0.0	0.0

Figure 3: Create the normalized proximal scent matrix 'PS' by weighting edges in 'T' according to the cosine similarity scores computed using Lucene.



PS

From → To	1	2	3	4
1	0.0	0.0	1.0	0.5
2	0.3	0.0	0.0	0.0
3	0.3	0.5	0.0	0.5
4	0.4	0.5	0.0	0.0

PS				*	A(1) +	E	=	A(2)
0.0	0.0	1.0	0.5		0	0		0.0
0.3	0.0	0.0	0.0		1	1		1
0.3	0.5	0.0	0.5		0	0		0.5
0.4	0.5	0.0	0.0		0	0		0.5

PS				*	A(2) +	E	=	A(3)
0.0	0.0	1.0	0.5		0.0	0		0.75
0.3	0.0	0.0	0.0		1	1		1
0.3	0.5	0.0	0.5		0.5	0		0.75
0.4	0.5	0.0	0.0		0.5	0		0.5

Figure 4: Example of application of spreading activation to matrix PS.

theories in which phenomena spread, and by information foraging theory in particular. It calculates how widely the spreading emanates. For PFIS, spreading activation calculates the likely spread of programmers to locations in source code, which can be interpreted as the expectation that a programmer trying to resolve a particular bug report will navigate to those locations in the program.

Spreading activation takes an activation vector ***A***, a scent matrix ***PS***, an entry vector ***E***, and a scalar parameter α. The parameter α scales ***PS*** by the portion of users who do not follow a link. In the initial iteration, the activation vector equals the entry vector. Activation is updated (spread) in each iteration ***t*** as follows [3]:

$$A(t) := \alpha PS * A(t-1) + E$$

In each iteration, activation from the entry vector is spread out to adjacent nodes, and activation present in any node is spread to neighboring nodes according to the scent, i.e., the edge weights in ***PS***. In the final iteration, activation vector ***A*** represents the activation of each node (package, class, method, field) in our topology ***T***. Normalizing ***A***, we interpret ***A*** as the probability of a hypothetical user visiting that node in ***T***. See Figure 4.

EXPERIMENT

Design, Participants, and Materials

We recruited 12 professional programmers from IBM. We required that each had at least two years experience programming Java, used Java for the majority of their software development, were familiar with Eclipse and bug tracking tools, and felt comfortable with searching, browsing, and finding bugs in code for a 3-hour period of time.

We searched for a program that met several criteria: we needed access to the source code, it needed to be written in Java, and it needed to be editable and executable through Eclipse, a standard Java IDE. We selected RSSOwl, an open source news reader that is one of the most actively maintained and downloaded projects hosted at Sourceforge.net. The popularity of newsreaders and the similarity of its UI to email clients meant that our participants would understand the functionality and interface after a brief introduction, ensuring that our participants could begin using and testing the program immediately.

RSSOwl (Figure 5) consists of three main panels: to the left, users may select news feeds from their favorites, to the upper right, users can review headlines. On selecting a headline, the story appears in the lower right panel of the application window.

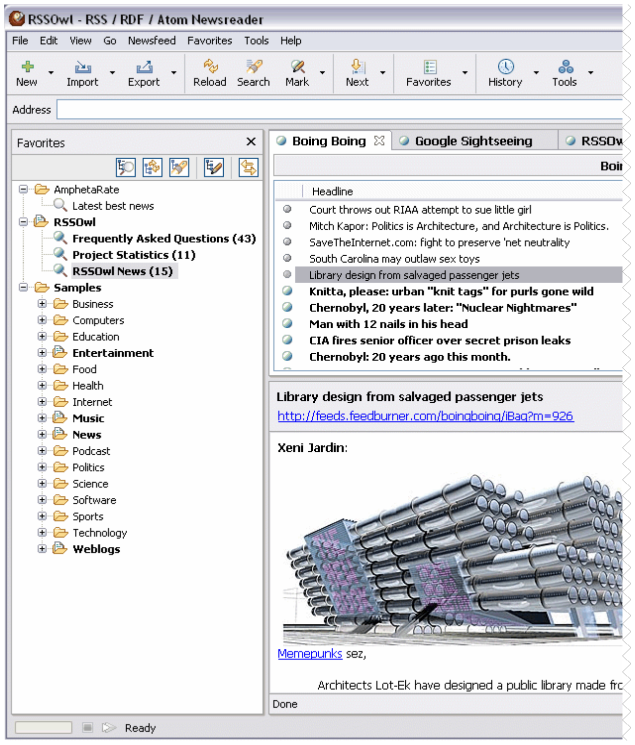
Having decided upon the program, we also needed bug reports for our participants to work on. Since we were interested in source code navigation and not the actual bug fixes, we wanted to ensure that the issue could not be solved within the duration of the session. We also decided that one issue should be about fixing erroneous code and the other about providing a missing feature. From these requirements, we selected two issues: 1458101: "HTML entities in titles of atom items not decoded" and 1398345: "Remove Feed Items Based on Age." We will refer to the first as issue B ("Bug") and the second as issue MF ("Missing Feature"). Each participant worked on both issues, and we counterbalanced the ordering of issues among subjects to control for learning effects. The former involves finding and fixing a bug, and the latter involves inserting missing functionality, requiring the search for a hook.

The issues we assigned to developers were open issues in RSSOwl. We considered looking at closed issues whose solution we could examine, but this would have meant locating an older version of RSSOwl for participants to work on, and would have required us to ensure that participants would not find the solution accidentally by browsing the web. Therefore, we decided that our participants would work on open issues, cognizant of the risk that RSSOwl's own developers could close the issues during the study, updating the web-available solution with the correct code in the process. (Fortunately, this did not happen.)

Procedure

Upon their arrival, after participants filled out initial paper work, we briefly described what RSSOwl is, and explained to our participants that we wanted them to try to find and possibly fix issues that we assigned to them. We then set up the instant messenger so that participants could contact us remotely. Then we excused ourselves from the room. We observed each participant remotely for three hours.

We recorded electronic transcripts and video of each session using Morae screen and event log capture software.



Court throws out RIAA attempt to sue little girl
Mitch Kapor: Politics is Architecture, and Architecture is Politics.
SaveTheInternet.com: fight to preserve 'net neutrality
South Carolina may outlaw sex toys
Library design from salvaged passenger jets
Knitta, please: urban "knit tags" for purls gone wild
Chernobyl, 20 years later: "Nuclear Nightmares"
Man with 12 nails in his head
CIA fires senior officer over secret prison leaks
Chernobyl: 20 years ago this month.

Library design from salvaged passenger jets
http://feeds.feedburner.com/boingboing/iBag?m=926
Xeni Jardin:
Memepunks sez,
Architects Lot-Ek have designed a public library made frc

Figure 5. RSSOwl, an RSS/RDF/Atom news reader

Participants had been instructed to think aloud, and were reminded to do so if they fell silent for an extended period. We archived the changes they made, if any. The electronic transcripts, videos, and source code served as the data sources we used in our analysis.

RESULTS

For each of the participants in each task, we analyzed the video and tallied the frequency and duration of visits for each of the class files. We will refer to these two metrics as visits and time span, respectively.

We then ran the PFIS algorithm for each task, applying the spreading activation algorithm over 100 iterations (at which point activation had settled and varied little between iterations), with an α of 1 to simulate users navigating within source code. Spreading activation requires us to specify the starting point of navigation (entry vector $\boldsymbol{E}$). In our study, we did not specify where programmers should start, so to construct our entry vector, we simply recorded in which classes or methods participants actually started. This generated a series of activation vectors describing the probable number of programmers to have visited each location.

Although PFIS reasons at a finer granularity than classes, we combined the methods' results by class for uniformity of comparison with other methods.

How well does PFIS model navigation compared with historical program navigation by other humans?

We first compare PFIS' ability to model any one programmer's navigation to human wisdom, namely the actual navigation patterns of all the other programmers in our study.

First, we computed the Spearman correlation between the "hold one out" aggregate program navigation among all but one of the programmers to predict the remaining programmer's navigation for each metric in each task. This collection of combined human judgments can be seen in Figures 6 and 7 as "Collective visits" and "Collective time span." In Figures 6 and 7, "Classes" represents the correlation between each programmer's navigation and the "hold one out" count of programmers who visited each class. Because programmers may visit classes multiple times, "Classes" differs from "Collective visits."

Second, we computed the Spearman correlation between each pair of programmers for each task (Issue B and MF) and each metric (time span and visits). This comparison shows how well any one programmer could predict another programmer's navigation, representing situations with a low level of available human history, such as for new projects or very small teams. These results are summarized in Figures 6 and 7 as "Pairwise visits" and "Pairwise time span."

Comparing the PFIS boxes in these figures to the "Collective" boxes shows that PFIS came reasonably close to aggregated human judgments. Further, PFIS was significantly better as a predictor of program navigation than a fellow

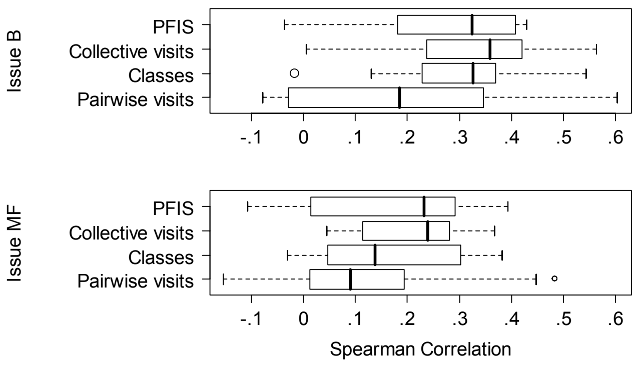


Figure 6. Correlation between each of the 12 participant's visits among classes and the respective model.

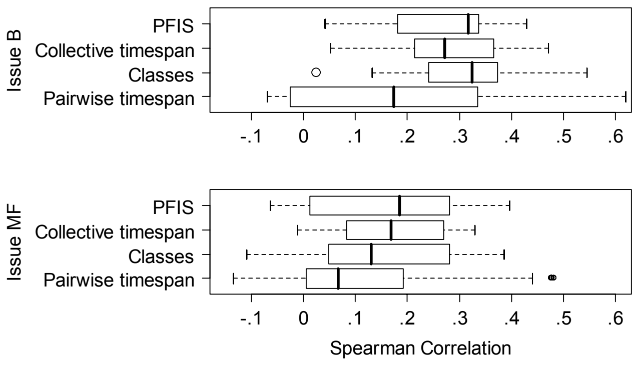


Figure 7. Correlation between each of the 12 participant's time span among classes and the respective model.

programmer (Pairwise boxes), on average. We determined this using bootstrap resampling, which shows that the 95% confidence interval of the difference between PFIS and the pairwise correlation of programmers does not span zero for each task and metric.

Previous tools (discussed earlier) have empirically shown the effectiveness of historical program navigation data as the basis for recommending which classes are relevant to an issue. Our results suggest that (1) information foraging theory may *account* for these tools' success, (2) when there is no historical data available, PFIS is a reasonable substitute, and (3) when there is only a little historical data available, PFIS may outperform these tools.

How well do PFIS and other models predict the places in the source code to which programmers will navigate?

The PFIS model, like other information foraging work, includes the notions of both scent and topology. Therefore, to consider whether scent or topology might work better alone than in combination, we devised two additional models that model scent only and topology only.

The scent-only model was the interword correlation model of [17], which works at the granularity of classes. We chose to include this model because of its basis in Pirolli's information foraging calculations of interword correlation [20] and its early indications of success in predicting programmer navigation. As in Pirolli's calculations, it weights terms in documents according to the term-frequency inverse

		Issue B	Issue MF
Interword correlation	time span	0.191	0.162
	visits	0.204	0.171
PFIS-Topology	time span	0.221	0.194
	visits	0.221	0.185
PFIS	time span	0.271	0.160
	visits	0.278	0.167
Collective	time span	0.281	0.167
	visits	0.328	0.212

Table 1: Average correlation between each programmer's navigation and the predictions of each model.

document frequency (TF-IDF) formula, commonly used in information retrieval systems [1]. This model is equivalent to the results returned by a standard (vector space model) search engine.

The PFIS-Topology model uses spreading activation on the topology aspect of PFIS only (in effect, replacing ***PS*** with ***T*** in the spreading activation iteration formula). Topology, as we have explained, is the collection of links. Since most links are method calls, the topology is similar to a call graph plus links to each method's definition. We used this model because many software tools are based on topological information, and we wanted to see how such tools potentially compare with new tools based on information foraging.

Table 1 shows the average correlations between each programmer's navigation choices and these three models' predictions (rows 1-3), with the aggregated human judgments repeated for ease of comparison (row 4).

Particularly for Issue MF, the performance of the models were quite similar. (Using bootstrap resampling, we determined that the 95% confidence interval of the difference between each pair of models spans zero, indicating no significant difference.)

We also determined that the difference between PFIS on Issue B versus Issue MF (for each metric) was significant, which raises a new question: are there fundamental differences in the ways programmers navigate when working on solving a bug versus working on adding new features? We will return to this issue in a later section.

Does PFIS account for all the class to class traversals made by programmers?

In the previous sections, we examined whether information foraging theory could model the *set* of users' visits to classes, by taking into account the relationships between classes, methods and variables. We now consider the *sequence* of these visits by examining the "edges" from one class to another traversed by our participants. Classes that are visited one directly after another could indicate a relationship between these classes, and when present, we would like to see if it is predicted by information foraging theory, and if not to understand what might account for it.

Out of 37,056 (mathematically, this is 193 permute 2) possible directed edges between classes, our analysis produced 3,612 class-class pairings based on the topology of links, leaving 33,444 not in the topology. Of the 3,612 in the topology, 1,162 were predicted by PFIS, as shown in Figure 8. Thus, according to PFIS, for the maintenance tasks in our study, only 3% of all possible class-class traversals were potentially relevant.

Did PFIS pick the right 3%? It was pretty close for Issue B, predicting 42 of the 61 pairs of classes (69%) that were traversed more than once. Only 7 visited pairs were not explained by either scent or topological relationships. For issue MF, PFIS was not as stellar, but of the 103 pairs traversed more than once, it still predicted 33 (32%). However, 47 pairs were not explained by scent or topological relationships. These relationships are shown in Figure 9.

These results were borne out statistically (Table 2). For Issue B, PFIS predicted edge traversals remarkably well—almost as well as it predicted the allocation of time and visits to classes (recall Table 1). However, it only weakly predicted edge traversals for Issue MF, also shown in Table 2.

We further investigated the pairings visited more than once for each issue that were *not* in the topology, to determine why the topology did not contain these edges. As shown in Table 3, some of the edges for issue MF could not be explained by the topology because two participants each added a class to the source code to implement the missing feature. Some class-class pairings had words in common even if they did not share any links between them (class-class scent). For Issue MF, we noticed some traversals that could be explained by indirect links. In such cases, one class would contain a variable of an interface type, and the other class would implement the interface. In some cases, membership to a common package explained the relationship between two classes (which did not fulfill the one-click definition of "link"). The remaining edges were a collection

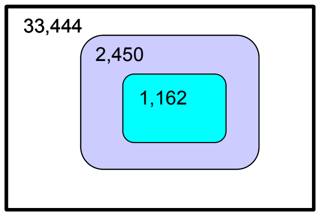


Figure 8: Edges. (Top): Inner oval: edges predicted by PFIS. Middle oval: edges in topology (only). Outer oval: edges not in topology or PFIS.

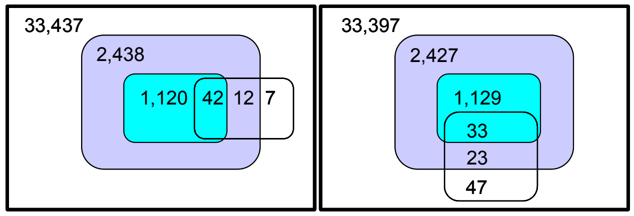


Figure 9: (Left): Same as Figure 8, but from perspective of Issue B; right rectangle highlights links visited more than once for B. (Right): From perspective of Issue MF; bottom rectangle highlights links visited more than once for MF.

Edge traversals predicted by	Issue B	Issue MF
PFIS-Topology	0.477	0.18
PFIS	0.46	0.19

Table 2: Spearman correlation between edge visits and proximal scent, call graph relations. All correlations shown were significant at p < 0.01.

Edges explained by	Issue B	Issue MF
Added class	0	11
Class-class scent	3	6
Indirect link not modeled	0	8
Membership to same package	3	10
Other	1	12

Table 3: Why are edges outside the topology?

of other explanations, including use of search and switches among multiple source code windows.

Back-links may be an important factor. Some edges in the rows of Table 3 were traversed when programmers went back to a class they had just come from, or cycled between two classes; namely in 6 out of Issue B's 32 edges that were traversed more than 3 times, and 27 out of 54 such edges for issue MF. Such back-links were not in the PFIS model (and have not been modeled by web-oriented information foraging models either), but these results show that they should be considered for inclusion in such models.

A final point is that not all scent is textual. Information foraging theory as applied to the web has pointed this out, but PFIS does not yet model it.

For example, programmers often derive hypotheses from observing run-time behavior. A possible corroboration of this notion of hypotheses' interactions with information foraging may be that many of the edge traversals throughout the rows of Table 3 involved GUI classes. Thus, we next consider hypotheses and scent.

Participants' Hypotheses

Prior research into debugging suggests that programmers form hypotheses about the reasons and places relevant to bugs, and that much of debugging revolves around attempts to confirm, refine, or refute those hypotheses [2, 14, 30].

Recall that PFIS performed better on issue B than on issue MF. A key assumption behind the PFIS model is that, when the issue being pursued starts with a bug report, the programmer forms hypotheses that linguistically relate to the bug report. Implicit in this assumption is the premise that a model can omit hypotheses and still be able to predict the necessary places to navigate well enough to be useful.

We decided to probe this premise by investigating whether programmers' hypothesis formation played fundamentally differing roles in the two issues. We investigated this question via content analysis of four transcribed participant sessions, two sessions for each issue. Two of the authors independently coded each transcript, replaying the videos at the same time to maintain context, coding the formation of an entirely new hypothesis, and coding for expanding or revising an existing hypothesis. The two coders reached agreement on over 90% of their codes.

These sessions revealed interesting differences in the timing, generality, and process of hypothesis formation for Issue B versus Issue MF. For example, when working on Issue B, participant 96s was able to formulate a specific and concrete hypothesis about exactly what needs to change within 5 minutes into the task. Participant 82 likewise formed a very concrete hypothesis for Issue B, and even more quickly. (Times are mm:ss)

Participant 96s (05:05): This is saying that there is HTML in the thing that shows up in the headline part.

Participant 82s (01:45): These escape characters... the apostrophes are not making it in ... Somehow we want to have escaped XML because this is in CDATA.

In contrast, for Issue MF, the hypotheses were more open-ended, possibly because there were multiple correct solutions to the problem. For example, Participant 85s's first hypothesis, given in the first minute, was simply a broad hypothesis about what he had to accomplish. It was 23 minutes into Issue MF, after the subject had investigated the code and referred back to the bug report, until he provided a hypothesis about how to actually address part of Issue MF:

Participant 85s (00:56): We want to remove the items based on the unread age and based on the read age.

Participant 85s (23:21): Now we want to add the expiration.

Participant 84s decided that it would be hard to develop a useful hypothesis about Issue MF without a better understanding of RSSOwl, so after forming a general hypothesis about "adding" at six minutes in, he changed his strategy, deciding to experiment with the system before attempting to refine the hypothesis. His experiments continued, without further hypothesis verbalizations for 22 more minutes. At that time he finally began to become form a concrete hypothesis about a suitable "hook" for adding the feature.

Participant 84s (06:28): So what we are looking to do is to add — I think what I'll start doing is trying to archive RSS feed entries after say some amount of time and then I'll make it increasingly more complex.

Participant 84s (28:55): When am I going to run this archive feature? The answer would seem to be is this something that is going to be run automatically? Yes.

Thus, hypothesis formation appears to be different in nature between these two issues.

One possible cause may be the wording of the bug reports, which are shown in Figure 10. Note that Issue B's bug report is fairly specific about symptoms and circumstances, which could have enabled the early formation of concrete hypotheses demonstrated by our participants. This could be simply a matter of better wording and content in these particular reports, but we propose that it could be in part inher-

HTML entities in titles of atom items not decoded In an atom feed such as crookedtimber.org/feed/atom/ you can find both the post titles and the post contents expressed as escaped-HTML. The post contents (including entities) are rendered correctly by RSSOwl, but the post titles that contain HTML entities are not. The entities, like &#8217; are not expanded by RSSOwl in post titles. I have attached a snapshot of that feed at this point in time.
Remove Feed Items Based on Age This is based on the assumption that the ability to archive feeds is available. Create an option to delete feed items after a certain amount of time has passed. Kind of like a rule saying 'Delete all feed items that are 3 months old'.

Figure 10: Snapshot of the bug reports' contents. (Top): Issue B. (Bottom): Issue MF.

ent in the traditions of reporting these two types of software issues. Reporting bugs often entails enumerating specific circumstances gone wrong with the assumption that the specifications are fairly well understood. In contrast, reporting the need for missing features emphasizes providing reasonably complete specifications for the desired feature. These differences are reflected in the bug reports in the figure. They are also reflected in comments that were posted to these bug reports. For Issue B, three of the four comments related to the possible location of the bug. (The fourth was about how easy/difficult the fix might be.) For Issue MF, the three comments elaborated upon the specifications.

We have previously suggested that it may be possible to use participants' words typed into "search" tools as surrogates for their hypotheses [17]. Eleven of our twelve participants used search, and searching was at least somewhat involved in the process of their work on their hypotheses.

Early concrete hypothesis formation for Issue B was apparent in our participants' search behaviors. They used search more for Issue B (59 searches, versus 32 for Issue MF), and what they searched for were low-level "how to" items on the web (34% of their searches), and locations in the code base (66%). An example of a "how to" web search was participant 82s's search for "converting strings to HTML java", and an example of a location search was his search for "addListener" in the code base.

In Issue MF, our participants searched much less than in Issue B—specifically, only 54% as much. The tasks were varied in order, so learning effects did not account for this difference. More to the point, the Issue MF searches were almost all in the code base (94%), looking for a hook. For example, Participant 85s's *only* two searches were in the code base, looking for "items.put" and "Date".

These results suggest that there are (at least) two relationships between hypotheses and search strings: searches in attempting to *form* a concrete hypothesis, and searches to *pursue* that concrete hypothesis after it is formed. For Issue B, most of the searches were of the pursuit type (since concrete hypotheses were formed fairly quickly), whereas for Issue MF, most of the searches were of the formation type, and were used only about half as much as with Issue B.

DISCUSSION AND IMPLICATIONS

The ultimate goal of this work is to provide theoretical grounding for tools to support software maintenance. The results from our predictive model are consistent with a number of descriptive theories of debugging [2, 16, 30], but also add to the theoretical understanding of debugging by providing a dynamic model. The model can be used in descriptive, explanatory, and predictive manners.

The PFIS model's performance shows that it already allows us to provide independent evidence about the premises behind current systems, such as Hipikat. It also allows us to reason about new design possibilities. In the domain of web navigation, information foraging and WUFIS have been used as the basis for both automated usability evaluation [4], and in browsing and navigation tools [19]. Similar applications of the theory could be developed for program navigation.

For example, just as ScentTrails [19] has been used to successfully speed up web navigation by highlighting hyperlinks to indicate paths to search results, our results suggest that source code navigation could be enhanced by highlighting links in class files with high scent for the bug report under consideration. Scent-based indicators could also be added to existing software tools based on other ways of discovering relationships between source code, such as developer navigation and action histories [27, 28, 31, 32], or structural or lexical relationships [5, 7, 8]. Scent indicators may also enhance the use of call graphs and program slices during maintenance, by indicating an additional relationship between parts of source code. Fault localization tools, which use multiple information sources to make a best guess about the location of a bug, may also benefit by using scent as an additional factor.

Information foraging theories have also been used as the basis for web site usability evaluation tools. In an analogous fashion, the PFIS model could be used for usability analysis of bug reports. More helpful bug reports may get written if scent-based feedback is provided to bug report authors regarding how well their report is narrowing the possible set of places the bug might be located. PFIS could also be used to evaluate proximal scent strength within source code itself, which could ultimately be used by programmers to improve their naming and commenting practices.

The above design suggestions are speculative, but they demonstrate how PFIS, as a predictive model, has the potential to both inform and evaluate tool development, as has been the case for information foraging in web navigation.

CONCLUSION

In this paper we presented PFIS (Programmer Flow by Information Scent), an information foraging model of programmers' navigation during maintenance, and evaluated it empirically. The main results of the evaluation were that:

- The PFIS model's performance was close to aggregated human decisions and better than individual fellow programmers' decisions as to where to navigate.
- PFIS missed only a small fraction of the 61 traversals that occurred more than once for Issue B. However, for Issue MF, it missed more of them. Many of the edges PFIS missed were topological relationships not usually considered by information foraging algorithms, such as back-links, and scent relationships not in the topology. These provide opportunities for future improvements.
- Our results suggest that the difference in prediction levels for the bug versus new feature may be due in part to differences in how information foraging related to hypotheses. We conjecture that this is due to inherent differences between the reporting of bugs versus feature requests, with the former tending to describe scent-carrying aspects such as circumstances and locations, but the latter describing specifications, which may have less scent.

Most important, our results suggest that information foraging's ability to predict programmer navigation during maintenance is indistinguishable from aggregated historical program navigation data. This in turn suggests that information foraging can provide a theoretical account of program navigation in software maintenance.

ACKNOWLEDGMENTS

This study was supported in part by IBM Research, by the EUSES Consortium via NSF ITR-0325273, and by an IBM International Faculty Award. This paper benefited greatly from the comments of the reviewers and the meta-reviewer.

REFERENCES

1. Baeza-Yates, R., Ribeiro-Neto, B. *Modern Information Retrieval*, Addison Wesley Longman (1999).
2. Brooks, R. Towards a theory of the cognitive processes in computing programming, *Int. J. Human-Computer Studies* 51 (1999), 197-211.
3. Chi, E., Pirolli, P, Chen, K. and Pitkow, J, Using information scent to model user information needs and actions on the web. In *Proc. CHI 2001*, ACM Press (2001).
4. Chi, E., Rosien, A., Supattanasiri, G., Williams, A., Royer, C., Chow, C., Robles, E., Dalal, B., Chen, J., Cousins, S. The Bloodhound project: Automating discovery of web usability issues using the InfoScent simulator. In *Proc. CHI 2003*, ACM Press (2003).
5. Cubranic, D., Murphy, G., Singer, J., and Booth, K., Hipikat: A project memory for software development, *IEEE Trans. Soft. Eng. 31*, 6 (2005), 446-465.
6. Cutting, D. Lucene, http://lucene.apache.org/java/docs/
7. DeLine, R., Czerwinski, M. and Robertson, G., Easing program comprehension by sharing navigation data, In *Proc. VLHCC*, IEEE (2005), 241-248.
8. DeLine, R., Khella, A., Czerwinski, M., and Robertson, G. Towards understanding programs through wear-based filtering, In *Proc. SoftVis*, ACM Press (2005), 183-192.
9. Eclipse Documentation: JDT Plug-in Developer Guide, http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html
10. Hill, W., Hollan, J., Wroblewski, D., McCandless, T. Edit wear and read wear, In *Proc. CHI 1992,* ACM Press (1992).
11. JUNG: Java Universal Network/Graph Framework, http://jung.sourceforge.net/
12. Kersten, M., Murphy, G. Mylar: A degree of interest model for IDEs, In *Proc. AOSD* (2005).
13. Ko, A., Aung, H., and Myers, B. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks, In *Proc. ICSE 2005*, IEEE (2005), 126-135.
14. Ko, A., Myers, B., A framework and methodology for studying the causes of software errors in programming systems, *J. Visual Langs. Computing 16*, 1-2, (2005).
15. Ko, A., Myers, B., and Chau, D. A linguistic analysis of how people describe software problems, In *Proc. VLHCC*, IEEE (2006), 127-136.
16. Ko, A., Myers, B., Coblenz, M., and Aung, H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, *IEEE Trans. Soft. Eng. 32*, 12 (2006), 971 - 987.
17. Lawrance, J., Bellamy, R., Burnett, M. Scents in programs: Does information foraging theory apply to program maintenance? In *Proc VLHCC,* IEEE (2007).
18. Nielsen, J. Information foraging: Why Google makes people leave your site faster http://www.useit.com/alertbox/20030630.html. (June 30, 2003.)
19. Olston, C. and Chi, E., ScentTrails: Integrating browsing and searching on the web, *ACM Trans. Computer-Human Interaction 10*, 3 (2003), 177-197.
20. Pirolli, P. Computational models of information scent-following in a very large browsable text collection. In *Proc. CHI 1997,* ACM Press (1997), 3-10.
21. Pirolli, P. and Card, S. Information foraging, *Psychology Review 106*, 4, (1999), 643-675.
22. Pirolli, P., and Fu, W. SNIF-ACT: A model of information foraging on the World Wide Web. *Lecture Notes in Computer Science 2702*, Springer (2003), 45-54.
23. Pirolli, P., Fu, W., Chi, E. and Farahat, A., Information scent and web navigation: Theory, models and automated usability evaluation. In *Proc. HCI International,* Erlbaum (2005).
24. Robillard, M., Coelho, W., and Murphy, G. How effective developers investigate source code: An exploratory study, *IEEE Trans. Soft. Eng. 30,* 12 (2004), 889-903.
25. Schneider, K., Gutwin, C., Penner, R., and Paquette, D. Mining a software developer's local interaction history, In *Proc. Intl. Wkshp Mining Software Repositories,* (2004).
26. Schummer, T., Lost and found in software space, In *Proc. HICSS 2001*.
27. Shirabad, J., Lethbridge, T., Matwin, S. Mining the maintenance history of a legacy system, In *Proc. ICSM*, IEEE (2003).
28. Singer, J., Elves, R., Storey, M. NavTracks: Supporting navigation in software maintenance, In *Proc. ICSM*, IEEE (2005).
29. Spool, J., Profetti, C., and Britain, D., Designing for the scent of information, *User Interface Eng.* (2004).
30. Vans, A. and von Mayrhauser, A., Program understanding behavior during corrective maintenance of large-scale software, *Int'l J. Human-Computer Studies* 51(1), 1999.
31. Ying, A., Murphy, G., Ng, R. and Chu-Carroll, M. Predicting source code changes by mining change history, *IEEE Trans. Software Engineering 30*, 2004, 574-586.
32. Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. Mining version histories to guide software changes, In *Proc. ICSE*, IEEE, (2004).