Reactive Information Foraging: An Empirical Investigation of Theory-Based Recommender Systems for Programmers

David Piorkowski[1], Scott D. Fleming[2], Christopher Scaffidi[1], Christopher Bogart[1], Margaret Burnett[1], Bonnie E. John[3], Rachel K. E. Bellamy[3], Calvin Swart[3]

[1]Oregon State University
Corvallis, OR 97331 USA
{piorkoda,cscaffid,bogart,burnett}@eecs.oregonstate.edu

[2]University of Memphis
Memphis, TN 38152 USA
Scott.Fleming@memphis.edu

[3]IBM T.J. Watson Research
Hawthorne, NY 10532 USA
{bejohn,rachel,cals}@us.ibm.com

ABSTRACT

Information Foraging Theory (IFT) has established itself as an important theory to explain how people seek information, but most work has focused more on the theory itself than on how best to apply it. In this paper, we investigate how to apply a reactive variant of IFT (Reactive IFT) to design *IFT-based tools,* with a special focus on such tools for ill-structured problems. Toward this end, we designed and implemented a variety of recommender algorithms to empirically investigate how to help people with the ill-structured problem of finding where to look for information while debugging source code. We varied the algorithms based on scent type supported (words alone vs. words + code structure), and based on use of foraging momentum to estimate rapidity of foragers' goal changes. Our empirical results showed that (1) using both words and code structure significantly improved the ability of the algorithms to recommend where software developers should look for information; (2) participants used recommendations to discover new places in the code and also as shortcuts to navigate to known places; and (3) low-momentum recommendations were significantly more useful than high-momentum recommendations, suggesting rapid and numerous goal changes in this type of setting. Overall, our contributions include two new recommendation algorithms, empirical evidence about when and why participants found IFT-based recommendations useful, and implications for the design of tools based on Reactive IFT.

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

In the past two decades, Information Foraging Theory (IFT) [23] has revolutionized our understanding of how people seek information during information-intensive tasks, and in particular, of how people browse the web. Models based on the theory have accurately predicted the links that people will click as they navigate through web sites [3, 7, 22], and researchers have used the theory to design better tools to help people navigate the web [4, 18, 28]. This work on web foraging has focused on tasks where people have well-defined, unchanging information goals.

However, when solving ill-structured problems, people's information goals are likely to change as they process and assimilate new information. In particular, numerous examples of ill-structured problems relating to software development have been discussed in the literature (e.g., [8, 9]), and in this paper, we focus on software development tasks in which software developers wrangle with finding and fixing a bug (i.e., debugging). Empirical studies show that software developers faced with such problems often start by seeking an overall understanding of a program's structure and runtime behavior [25, 26], but they soon start to have questions that lead to exploring specific code [25]. Answers to these questions spark multiple new questions [13, 15, 25, 26]. Thus, the developers' goals evolve, unlike in the web foraging tasks that researchers studied using IFT.

Reactive Information Foraging Theory is a relatively new variant of IFT that we have previously developed to incorporate the notion of evolving goals [16]. Reactive IFT has been shown to predict software developers' navigation choices as they completed debugging tasks [16, 20]. IFT says that a person seeking information follows *information scent* emitted from *cues* in the environment in a manner similar to how a predator foraging in the wild follows scent to prey [22]. Related work provided models for inferring a web forager's goals from cues followed [3], and Reactive IFT led to analogous models that infer evolving goals of software developers [16, 20].

In this paper, we explore how to apply Reactive IFT to provide tools that infer, in real time, developers' evolving goals and use those inferences to recommend places in code

that satisfy those goals. Several questions arise about how to operationalize the theory to build useful tools.

One question is how much of a person's recent history to consider when inferring the current goal. As noted above, a developer typically investigates code related to a certain goal for a while—which we refer to here as "building up foraging momentum" related to that goal—before shifting to a new goal. Reactive IFT indicates that the software developer's current momentum, reflected by recent navigation actions, can be used to infer the goal, but it does not specify how many recent navigations to use for this analysis.

Another question that arises in using IFT for tool design is which cues to consider for inferring goals. The web foraging research (e.g., [3, 22]) has emphasized operationalizing cues as words that appear in web pages that have been visited. Program code tends to be less "wordy" and more structured than natural-language text, and some prior models of developer navigation have hinted that at least in this domain, it might be useful to consider both word and code-structure cues to infer developer goals [16, 20].

Two other subsidiary questions also arise. First, because the navigational behavior of developers changes over the course of a task (as noted earlier), it is questionable whether different operationalizations will be more or less appropriate at different points in a task. Second, related to this point, different developers might obtain different value from Reactive IFT-based tools at different points in a task, raising the question of when and why recommendations are useful.

Consequently, our specific research questions were:

- RQ1: How do a Reactive IFT-based tool's assumptions about foraging momentum affect its ability to infer a developer's goal and produce useful recommendations?
- RQ2: Does a Reactive IFT-based tool that considers word *and* code-structure cues yield more useful recommendations than another tool that considers only word cues?
- RQ3: Does a Reactive IFT-based tool's ability to provide useful recommendations change as a task progresses?
- RQ4: When and why do software developers find Reactive IFT-based tools' recommendations useful (or not)?

RELATED WORK

Empirical studies have revealed many cases where software developers navigate through source code. For example, during debugging tasks in a laboratory study, developers spent 35% of their time browsing and searching through code [13]. Aside from debugging, code navigation is also essential for implementing new features, porting code, documenting code, extracting reusable code, and virtually every software development activity.

Code navigation is a key part in a learning process whereby the developer finds the information needed to work out a concrete goal to complete the task at hand. At the start of a maintenance task in laboratory and field studies, a developer usually searched for code that could serve as an initial focus point [25, 30]. Over the course of a task, the developer typically began to ask questions about how pieces of code were related [6, 25]. Developers also frequently navigated between locations in code, revisiting places to learn about structural relationships [19, 26]. As developers discovered answers to their questions, they broadened their focus to include more code [26]. Eventually, they planned specific changes that they believed would eliminate a defect or create a new feature [13, 26].

Various tools are aimed at easing these navigations by providing shortcuts to recommended places in the code that might hold valuable information. Several tools provide "history" links back to code that the developer has recently visited [12, 19]. Others offer shortcuts to places that other developers historically have read and/or edited after the current location [6, 10]. Still other tools provide shortcuts to code or other artifacts based on textual similarity, textual proximity, method-invocation, or nesting [5, 11, 27, 29].

A limitation of these tools is that they do not explicitly take into account the *evolution in goals* that typically occurs as a developer learns during a task. For example, upon visiting a Java method's code at the start of debugging, a developer might have no idea what needs to be edited; upon revisiting the location later, the developer might have formulated a goal to make certain kinds of edits. Yet most of the tools above would provide exactly the same shortcuts in both situations, regardless of the developer's new goal. Of the above tools, only those that show a "history" of recent places would behave differently during a particular task: they would stop showing links to locations that were not recently visited, which is an implicit model of evolving goals at best.

IFT, as we discuss below, offers a start toward addressing these limitations. Our objective in the current study is to investigate how to use this theory to help software developers find the information that they need during tasks.

BACKGROUND: INFORMATION FORAGING THEORY

IFT is a theory of how people seek information during information-intensive tasks [23]. It is based on the assumption that humans generally tend toward rational strategies that enable them to locate and process information efficiently. It was inspired by biological theories of how animals seek food in the wild. In IFT, a *predator* (person seeking information) pursues *prey* (valuable sources of information) through a *topology* (collection of navigable paths through an information environment). To find prey, the predator follows *information scent* that he/she obtained from *cues* in the environment. Cues are associated with navigation options, and the *scent* on cues is the predator's assessment of the value and cost of information sources obtained by taking the associated navigation options.

IFT was originally applied to user-web interaction. Models based on the theory have predicted the web pages to which

people will navigate during particular tasks [3, 7, 21]. Leveraging this predictive power, IFT has also inspired principles for the design of web sites [17, 28].

More recently, researchers have attempted to translate the IFT results from the web domain to software development [14, 15, 16]. To reconcile developers' evolving goals with the web-based models, which were focused on static, unchanging goals, we proposed Reactive Information Foraging Theory to describe evolving goals. A model based directly on this theory, *PFIS2* [16] (later refined to *PFIS3* [20]), was able to accurately predict developer navigation.

In this paper, we build upon this earlier work, adapting these models to build recommender *tools* that try to satisfy developer goals. In particular, we investigate the research questions mentioned earlier about how the operationalization of momentum and scent affect tool usefulness, as well as about when and why developers find the resulting tools useful. Ours is the first application of Reactive IFT as a theory for assisting software developers.

EMPIRICAL METHOD

In our empirical study, we invited professional developers to complete a debugging task using a new Eclipse plug-in tool that supplied links to places in the code that might provide information needed for the task. Within this plug-in, we activated different recommendation algorithms based on different operationalizations of momentum and scent. Specifically, we considered different algorithms in a 2×2 factorial design (with one factor for the operationalization of momentum and another factor for the operationalization of scent). We then measured what proportion of the time participants went to locations recommended by different algorithms. In our analysis, we also examined whether algorithms' recommendations were more or less useful at different periods of the task, and we qualitatively analyzed what kinds of benefits participants obtained.

Study Environment

We implemented our recommendation system as a plug-in for the Eclipse IDE (Figure 1). Interface elements 1–4 in Figure 1a are all those that commonly appear in the Java perspective of Eclipse: (1) Package Explorer view, (2) Outline view, (3) Java editor, and (4) Console view. Interface element 5 is our Recommendation view.

Figure 1b depicts a close-up of this Recommendation view, which had three areas: (1) the current method (i.e., the one that the text cursor last entered), (2) the current recommendations, and (3) methods bookmarked, or *pinned*, by the developer. Each time the developer navigated to a method, the current method updated to reflect the navigation, and the recommendations were recalculated (using whichever of our algorithms was activated at the time). The developer could also manually drag the current method or any recommendation into the pinned area to save it for later.

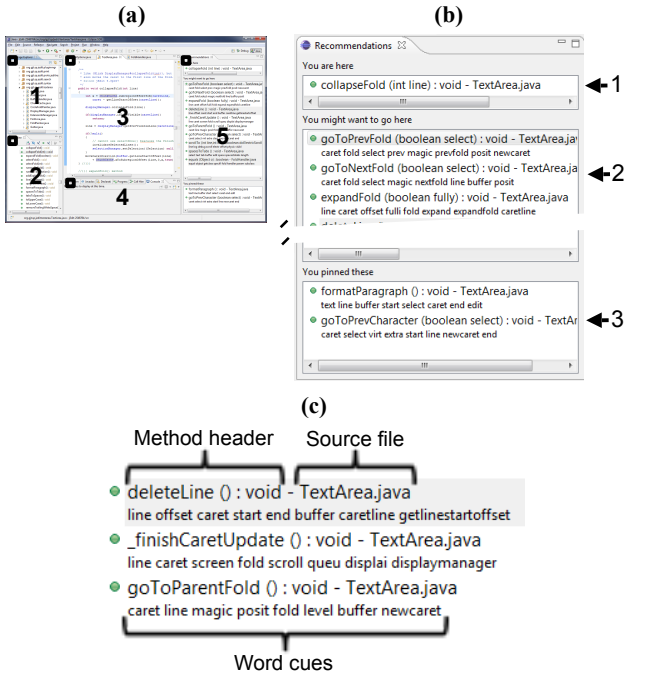

Figure 1. Eclipse interface with our Recommendation view: (a) Eclipse layout, (b) Recommendation view, and (c) examples of three recommendations.

Each recommendation displayed words to help participants assess its relevance (Figure 1c). These words were sorted based on their importance according to the amount of weight given to them internally by the recommender algorithm active at the time. The Recommendation view also distinguished recommendations to methods the participant had previously visited from recommendations to methods not already visited by highlighting the latter in gray.

For data collection, our plug-in recorded a log of participant interactions with Eclipse. Additionally, we video-recorded each session and automatically logged screen captures.

Recommendation Algorithms Considered

To investigate whether operationalizing scent as words + structure would produce better recommendations than words alone (RQ2), we implemented one recommender algorithm based on words + structure, PFIS-R, and another based on words alone, TFIDF-R. The PFIS-R recommender is based on the PFIS3 predictive model [20], motivated by PFIS3's success in predicting developer navigation. The other algorithm, TFIDF-R, is based on a vector space model commonly used in information retrieval [1]. These two algorithms represent the main camps of how to model information foraging: strong reliance on words (e.g., TFIDF, LSA) and balancing words with information structure (e.g., [3, 14, 15, 16, 18]). Both share a reliance on words, because word-based approaches have dominated IFT (e.g., [3, 4, 7, 22]) and the literature contains some evidence that words can be used to predict where developers will navigate [14].

RQ1 required a need to manipulate the operationalization of the *foraging momentum*—specifically, how much history to use for making recommendations. To take momentum into account, we parameterized each algorithm with δ, the number of navigations to look back in making a recommendation. An algorithm with δ=1 uses only the contents of the last method visited and thus ignores any momentum that the developer might have built up; we refer to this algorithm configuration as *low momentum*. In contrast, an algorithm with δ=10 uses the contents of the last 10 methods visited and will thus be influenced by momentum built up during those navigations; we refer to this algorithm configuration as *high momentum*. Thus, the greater the value of δ, the more momentum the recommender assumes in making its recommendations. Combining two momentum configurations, δ=1 and δ=10, with the two different algorithms (PFIS-R vs. TFIDF-R) gives the 4 possible combinations of our 2×2 factor design.

The algorithms take as an input the sequence of methods to which a developer has navigated so far. In the current study, we recorded a navigation to a method *m* each time the developer performed an action that caused the text cursor to enter the text of *m*. For our purposes, the text of a method comprises the method's signature, body, and (Javadoc) comments. Both PFIS-R and TFIDF-R normalize this textual input by excluding punctuation and *stop words*, which include common words, like *the*, as well as the list of Java keywords. Camel-case words are broken into separate case-insensitive words, although the original camel-case word is retained as well (e.g., *setFoldText* would be treated as an input of *setfoldtext*, *set*, *fold*, and *text*). For the remainder of the paper, *method text* refers to normalized method text.

The algorithms produce as output 10 recommendations, divided evenly between methods previously visited by the developer and methods not previously visited. Developers commonly revisit methods to recover mental state and explore new methods to find and understand the program's essential elements [19]. Thus, by recommending both visited and unvisited methods, we sought to support both types of navigation. If there are fewer than 5 previously visited recommendations, the algorithms fill out the remainder of the 10 recommendations with new recommendations, and vice versa. The algorithms would sometimes need to resolve ties when generating this list. The reason is that although the algorithm ranks methods by computing a continuous measure of relatedness (as discussed in the algorithm details below), the variables in those calculations take on a small number of distinct values in practice. When selecting the 10 methods to recommend, ties were resolved by choosing nondeterministically among the tied methods.

PFIS-R Recommender

Figure 2 summarizes the PFIS-R recommendation algorithm, which builds on the earlier PFIS2 and PFIS3 models by informing its recommendations using word cues and code-structure cues (i.e., call dependences) in methods that the developer previously visited. In brief, PFIS-R maintains a graph of word and method vertices such that edges between method vertices capture the structural relationships between methods, and edges between words and methods capture lexical relationships between methods. The algorithm *spreads activation* [22] over this graph, starting from the vertices for the last δ methods that the developer visited. To account for momentum, vertices for methods that the developer visited more recently are initialized with greater activation. The algorithm considers methods with the highest resulting activation to be the best recommendations, and those are the ones that sort first in Figure 1c.

PFIS-R Algorithm

Definitions:

- Method set M: set of all methods in the code base.
- Navigation history H: sequence of methods to which the developer has navigated so far.
- Word set W: set of all words in all method text in M.
- Graph $G = (V_m \cup V_w, E_m \cup E_w)$ such that V_m and V_w have a one-to-one relationship with M and W, respectively. For all m_a, m_b in M and their associated method vertices mv_a, mv_b in V_m, E_m contains one and only one edge connecting mv_a and mv_b if and only if the body of m_a contains a call to m_b or the body of m_b contains a call to m_a. For each word w in W and method m in M, E_w contains one and only one edge connecting wv and mv if and only if w is in the method text of m.

Steps for making recommendations:

- Set activation of each vertex in G to 0.
- For each method m such that m is the kth method in H and $|H|-\delta < k$, increment the method vertex mv by $0.9^{|H|-k}$.
- Spread activation (α=0.85 and edge weights=1) such that only word vertices receive activation.
- Spread activation again (α=0.85 and edge weights=1) such that only method vertices receive activation.
- Recommend methods with greatest activation.

Figure 2. Formal definition of the PFIS-R Algorithm.

TFIDF-R Recommender

In contrast to PFIS-R,TFIDF-R bases its recommendations on lexical similarity only. TFIDF-R treats the code base as a corpus of documents with the (normalized) text of each method being a document. The algorithm maintains a word-by-document matrix *MW* that specifies the importance of each word in *W* to each method in *M* by computing the tf–idf (term frequency–inverse document frequency [1]) weight for every word-method combination. It uses *MW* to assess the lexical similarity between methods by constructing a document-by-document matrix *MS* that specifies the cosine similarity measure [1] for all pairs of documents. *MS*(*m*) denotes the vector of cosine similarity scores associated with a particular method *m* in *MS*.

To account for the developer's foraging momentum, TFIDF-R sums the $MS(m)$ vectors for the last δ methods to which the developer navigated, decaying older navigations at a rate of 0.9. Specifically, for each method m such that m is the kth method in H, $decayedMS(m) = 0.9^{|H|-k} \cdot MS(m)$. A final recommendation vector VR sums the decayed vectors for the last δ methods in H. The algorithm regards the methods with the greatest values in VR as the best recommendations, and those sort first in Figure 1c.

Participants

We recruited eleven professional software developers from a large company to participate in an empirical study comparing the usefulness of our four different algorithm configurations. Technical failures in the study environment invalidated the data for two of these participants; thus, we analyzed only the data for the remaining nine participants. We asked each participant to fix a defect in the jEdit software project. None of the participants had ever seen the jEdit code before, and with 6,468 methods, the jEdit code base provided a large space to forage within. The defect was from an actual bug report, #2548764, which described a problem with jEdit's text "folding" functionality.

Procedure

Each study session took about 180 min. Participants began by filling out a pre-questionnaire that gathered background information. Next, they engaged in two back-to-back task periods. Prior to the first task period, we introduced participants to the video equipment and started recording. Each task period was associated with a different treatment (i.e., one of our four algorithm configurations). Table 1 lists the treatment assignments, which we balanced to account for learning effects.

Each task period began with a short tutorial task on the tool. Some algorithms may have produced poor recommendations in the first task period, and we did not want participants who received such treatments to ignore recommendations in the second task period. Consequently, the tutorial in the second task period informed the participant that the tool's recommendation algorithm had been switched and asked the participant to repeat the tutorial task to see how the recommendations had changed.

Within each task period after the corresponding tutorial, participants worked on the jEdit debugging task for 35 minutes. To assess how participants used recommendations and when they were useful, we asked them to "talk aloud" as they worked. At the end of each task period, we interrupted participants and had them fill out a questionnaire that asked for their opinion of the recommendations.

Algorithm	Task period assignment for each participant								
	2	3	5	6	7	8	9	10	11
PFIS-R(δ=1)	2nd			2nd			1st	1st	
PFIS-R(δ=10)		2nd	2nd		1st	1st			2nd
TFIDF-R(δ=1)	1st		1st			2nd	2nd		1st
TFIDF-R(δ=10)		1st		1st	2nd			2nd	

Table 1. Treatment assignments to participants and task periods. (Participants 1 and 4 removed due to technical failures.)

After both task periods, the study session ended with a 35-minute semi-structured retrospective interview in which an interviewer stepped through events of interest in the screen capture videos and asked the participant questions about those events. For each time the participant navigated to a method, the interviewer asked, "What did you learn from this place?" and for each time the participant clicked on a recommendation, the interviewer asked the participant why he/she did that. Due to time constraints, the interviewer was sometimes unable to ask about all these events of interest in a session.

Analysis Procedure

Ultimately, a recommender's quality is its usefulness, but evaluating usefulness raises challenges. For instance, a participant may ignore recommendations that would otherwise be useful because the participant is unfamiliar with the tool. As professional developers, our participants have honed their navigation strategies over years of experience, and they might favor their practiced strategies over a relatively unfamiliar tool.

To address this problem, we conducted two analyses. The first was a quantitative analysis of *hit rate*. This analysis compared the algorithms' recommendations to the places that participants actually navigated, regardless of whether they actually used the tool to navigate there. The second was a qualitative analysis of *demonstrated usefulness*, which assessed whether recommendations participants followed via the tool were useful in the participants' opinions.

Hit rate

To assess hit rate, we computed the top-10 recommendations from the tool algorithm that was active at the moment before each user made each navigation. We scored the recommendations as a *hit* if the user navigated to a recommended location within the next s navigations. It was difficult to estimate the window of time in which a recommendation might have been usefully pertinent to a participant, so we explored two time-window sizes: a hit within the next 10 navigations (s=10) and a hit on the next navigation (s=1). Note that the time-window size s is concerned with how we score the hit rate of recommendations (over future navigations), not to be confused with the sensitivity to momentum factor δ in this study, which also had values of 1 and 10 (the number of past navigations from recent history).

Recall that nondeterminism arises in the case of ties between recommendations. To account for this nondeterminism, we took the best tie-breaking choice to make a best-case list of recommendations (maximizing hit rate), and we took the worst tie-breaking choice to make the worst-case list (minimizing hit rate), thereby bounding the effects of nondeterministic tie-breaking.

Given the two choices of *s* and the two choices of tie-breaking, we obtained four separate assessments of each recommendation algorithm's hit rate. Each of these was treated as a separate dependent variable in a separate regression. Because our dependent variable was dichotomous (indicating whether or not a recommendation was considered a hit), we used logistic regression. In addition to algorithm, our regression model also included task period and participant as categorical independent variables (i.e., indicator variables). After performing the regression, we used the Wald test to assess whether each variable (algorithm configuration, participant, and task period) had a statistically significant effect on hit rate. Using the coefficients provided by the regression, we also computed the overall hit rate of each algorithm, after subtracting out the effects of participant and task period.

Demonstrated usefulness

To assess recommendation usefulness, we looked at the recommendations that the developers followed to see if the recommendations helped the developers make progress. To do this analysis, we qualitatively analyzed the verbalizations the participants made during task performance and their answers to our probes during the retrospective interview. In particular, for each followed recommendation, our interviewer asked the participant "What did you learn from this place?" If the participant responded negatively (e.g., "I learned nothing") or with apparent uncertainty (e.g., "I don't know"), we coded the navigation as not useful; otherwise, we coded it as useful.

RESULTS: HIT RATE AND USEFULNESS

Hit Rate

Participants averaged one navigation every 90 seconds, or 447 navigations in all (Figure 3). We used this data to analyze hit rate. Figure 4 depicts the hit-rate results by algorithm configuration for window sizes s=10 and s=1. Our data revealed significant differences in the hit rates for the four configurations. Specifically, we consistently found $\chi^2(3) \geq 9$ and $p \leq 0.03$ regardless of whether we used s=1 or s=10, worst- or best-case tie-breaking, and δ=1 or δ=10.

Low vs. high sensitivity to momentum (RQ1)

Each low-momentum (δ=1) algorithm consistently scored a significantly better hit rate than the corresponding high-momentum (δ=10) algorithm (Figure 4). We found $z \geq 2.45$, $p \leq 0.01$ regardless of whether we used s=1 or s=10, worst- or best-case tie-breaking, and TFIDF-R or PFIS-R. Thus, with respect to RQ1, we found that low-momentum algorithms outperformed high-momentum.

Word and code-structure cues vs. word cues alone (RQ2)

We found no meaningful difference between PFIS-R and TFIDF-R, regardless of whether we used s=1 or s=10, or worst- or best-case tie-breaking. Neither low-momentum (δ=1) configuration of PFIS-R and TFIDF-R outperformed the other (at $p < 0.05$). Between high-momentum (δ=10) configurations, PFIS-R outperformed TFIDF-R, but the difference was not statistically significant (at $p < 0.05$). Additionally, we found no significant interaction between algorithm and sensitivity to momentum (δ)

First vs. second task period (RQ3)

Participants navigated almost twice as frequently during the second task period as during the first (Figure 3). Comparing between task periods (Figure 5), we saw suggestive differ-

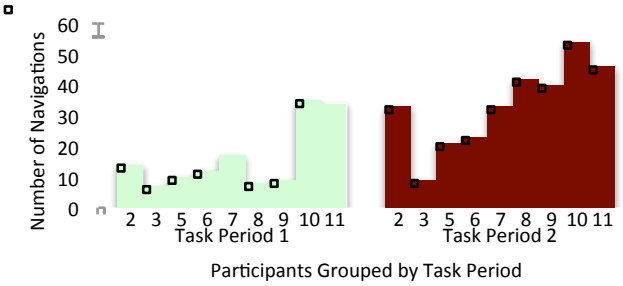

Figure 3. Number of navigations by each participant.

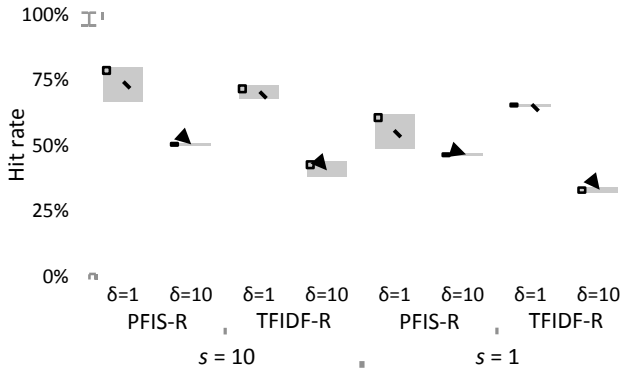

Figure 4. Hit rate of each algorithm (averaged over all task periods and participants) for s=10 (left half) and s=1 (right half), with rectangles indicating ranges between best- and worst-case tie-breaking. The δ=1 algorithms consistently outperformed the δ=10 ones (see arrows).

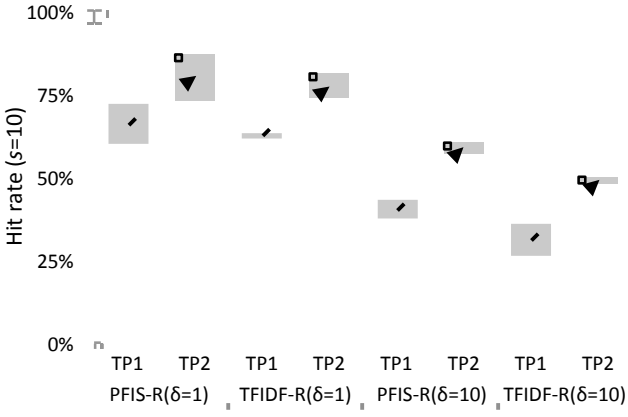

Figure 5. Hit rate (s=10) of each algorithm in the first (TP1) and second (TP2) task periods, with rectangles indicating ranges between best- and worst-case tie-breaking. Hit rates increased in the second task period (see arrows).

ences when evaluating recommendations with respect to the developer's next navigation (s=1), but these differences were not statistically significant (at $p < 0.05$). However, with respect to the developer's next 10 navigations (s=10), both algorithms improved significantly in the second task period over the first (pairwise $z = 2.37$, $p = 0.02$ for worst-case tie-breaking; $z = 3.44$, $p < 0.001$ for best-case).

Demonstrated Usefulness

All participants clicked some recommendations (Figure 6), for 62 clicks in total, and we collected interview responses for 44 clicks. Figure 7 summarizes the proportion of those clicks that participants reported as useful. We did not test for statistical significance due to the low number of clicks in each of the treatments.

Low vs. high sensitivity to momentum (RQ1)

The participants reported as useful a greater proportion of the recommendations from low-momentum algorithms than the recommendations from high-momentum algorithms (RQ1). This tendency triangulates with our finding that the low-momentum (δ=1) algorithms demonstrated a higher hit rate than the high-momentum (δ=10) ones.

Word and code-structure cues vs. word cues alone (RQ2)

Also triangulating with our hit-rate results, our demonstrated-usefulness results showed no consistent difference between PFIS-R and TFIDF-R.

First vs. second task period (RQ3)

Similar to what we saw with hit rate, the participants clicked recommendations more frequently during the second task period than during the first.

Opinion Questionnaire

Figure 8 illustrates the opinion results. Participants completed a total of 18 opinion questionnaires (1 per task period) that asked the question (5-point Likert) "Was this tool valuable in getting you to useful parts of the source code?" Again, we omitted statistical tests when analyzing this data due to the low number of clicks in each treatment.

Low vs. high sensitivity to momentum (RQ1)

Triangulating with our hit-rate and demonstrated-usefulness results, participants rated the low-momentum algorithms more favorably than the high-momentum ones (Figure 8b).

Word and code-structure cues vs. word cues alone (RQ2)

Similar to our hit-rate and demonstrated-usefulness results, the opinion results showed no suggestive difference between TFIDF-R and PFIS-R.

First vs. second task period (RQ3)

Participants seemed to rate all algorithms more favorably in the second task period than in the first (Figure 8c), consistent with our hit rate analysis where the second task period had a higher hit rate than the first.

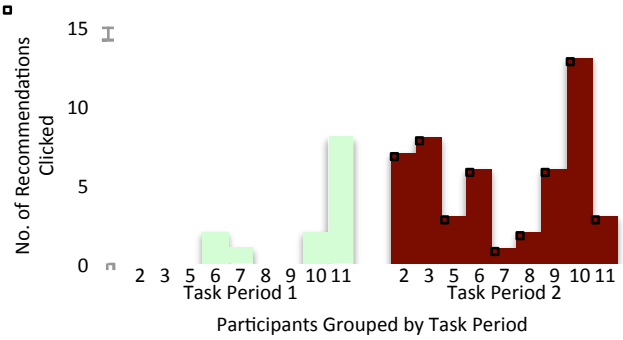

Figure 6. Number of times that participants clicked recommendations. Annotations indicate participant ID.

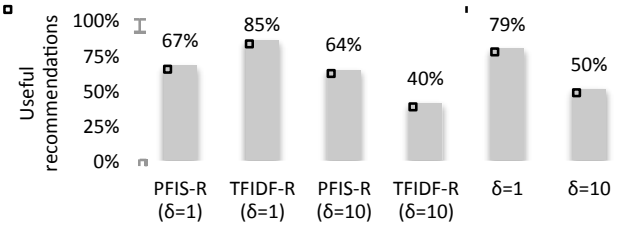

Figure 7. Percentage of recommendations that participants reported learning something from, grouped by algorithm and by low and high momentum.

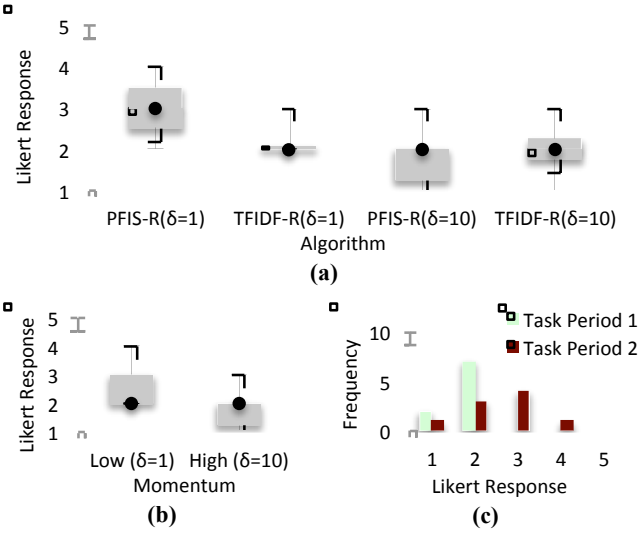

Figure 8. Results of questionnaires on the value of the recommendation system (1 = Entirely worthless, 5 = Very valuable).

RESULTS & DISCUSSION: WHAT PARTICIPANTS SAID

To better understand when and why participants found recommendations useful (RQ4), we qualitatively analyzed the talk-aloud and retrospective interview recordings.

Using Recommendations for Efficiency

Benefits of recommendations based on code structure

Some participants benefitted from PFIS-R's use of code structure in making recommendations. For example, Partic-

ipant 2 found code-structure-based recommendations useful while using an exception stack trace to navigate through code. He used the stack trace to open a method that contained a call to another method `recalculateLastPhysicalLine` (i.e., following the call graph structure):

> Participant 2: "I wanted to go to the declaration of [recalculateLastPhysicalLine] and you know, my god, the [recommendation system] had it sitting there so I thought I'll go over and select it… I like that it was bigger and it was right there so it just seemed like I would both go to it and maybe learn something in the process…"

This quote reveals a case where a structure-based recommendation led directly to a desired method, making it efficient to navigate there in a single mouse click. PFIS-R's use of code-structure cues made a difference in this case. In contrast, running the TFIDF-R algorithm (with δ=1 and δ=10) on Participant 2's navigations revealed that this words-only algorithm would not have made this recommendation (even in the best case).

Recommendations as working set

Several participants used recommendations to efficiently navigate back to previously visited methods; e.g.:

> Participant 9: "At this point, I'm kind of abusing the recommendations as a history because they are the fastest way to get where I want to go."

As another example, Participant 8 took advantage of the implicit "history" when, using Eclipse's debugger, he accidently stepped out of a method `transactionComplete` that he meant to inspect:

> Participant 8: "So that kicks me out to the catch and I remembered it was in the transactionComplete method… I had remembered that this recommendation had shown a bunch of transactionCompletes, so I was just clicking around to just find where I was."

It could have taken Participant 8 considerably more effort to get back to `transactionComplete` (e.g., by rerunning the debugger) without the recommendation. Participants received historical recommendations from both low- and high-momentum algorithms because all algorithms made recommendations to previously visited methods.

Participants who used recommendations to get back to previously visited methods were essentially using the recommendations as a sort of *working set*. Previous research has shown that developers tend to navigate frequently to methods in their working set [19, 20], and several successful tools have been developed that emphasize working set [2, 6, 12]. A novel feature of our operationalization of Reactive IFT is that it implicitly supports working set while also helping the user explore new places.

Using Recommendations for Discovery

"Aha! moments"

Some participants followed recommendations to methods that they were apparently unaware of and expressed excitement about how useful the recommendation turned out to be. For instance, Participant 2 was having difficulty finding code to focus on. He perused the recommendations:

> Participant 2: "'You might want to go here.' … 'collapseFold' … 'expandFold' ... OK, 'collapseFold.' [Selects collapseFold recommendation; reads code comments.] 'Collapses the fold associated at the specified line index.' *OK! Now, this is where I want to be!* Collapsing the fold."

Participant 6 was having similar difficulty finding code to focus on when he turned to the recommendations:

> Participant 6: "[Selects loadMenu recommendation; reads code comments.] 'Creates a menu, the menu label is set from the name property, name.label propery.' *Oh! Oh! Yes!* … This thing looks like the class that might put up the menu. *I like that!*"

Our choice to make half of the recommendations be to previously unvisited methods created opportunities for participants engaged in exploratory navigation to have "aha moments" such as these. However, not all exploratory navigations were to unvisited methods. Participant 6 had already visited `loadMenu` when he followed the recommendation. On his first visit, he did not notice anything interesting about the method. It was only after he followed a recommendation to the method that he discovered needed information in the method.

Misleading cues

In some instances, participants navigated to methods that contained cues that generated strong scent, but the recommendations did not help them discover methods that would satisfy their goals. For instance, Participant 6 wanted to find the method that implemented the action for one of the items in jEdit's Edit menu. All the methods that he navigated to contained words (i.e., cues) related to his goal, such as *menu*, *edit*, and *action*. He then noticed a recommendation for `JEditAbstractEditAction` that included the keywords *edit* and *action*:

> Participant 6: "I was looking for the method that would get run when someone picked on that [Edit] menu item. So again, [the recommendation] could be the mnemonic suggestion of the class that maybe that was some kind of action that would get run in the Edit menu."

Participant 6 clicked the recommendation, but unfortunately the method did not contain code for implementing menu-item actions. Instead, it contained code that implemented the menu framework.

The recommendation system (running TFIDF-R, δ=10) could not help Participant 6 because it could not relate the word cues that he was following to the method that would satisfy his goal. That method contained entirely different word cues from the menu-framework code. Because the participant was navigating through the framework code, building momentum on those cues, the algorithm did not recognize the relevance of the needed method.

Code structure cues used by PFIS-R might help overcome this problem generally; however, in this particular case, PFIS-R would not have had access to the structural information needed to connect the menu and action code because that information was contained in a properties file that was not part of jEdit's Java code base. This problem highlights the importance of having complete structural information in operationalizing Reactive IFT.

DISCUSSION: IMPLICATIONS AND OPEN QUESTIONS

Sensitivity to Momentum

The Reactive IFT tools that were less sensitive to participants' foraging momentum (i.e., low momentum, δ=1) produced better recommendations than those more sensitive to momentum (i.e., high momentum, δ=10). This result runs counter to previous work on recommender systems, which suggested that using more history was better for predicting future navigation [16, 20]. Outside software maintenance, successful recommender systems use historical behavior going back days, weeks, and years [24]. In light of this past work, our result that δ=1 produced better recommendations than δ=10 was unexpected.

One interpretation is that this result may be due to participants' goals evolving rapidly and repeatedly. Studies have shown that as developers navigate through code, they continually ask new questions [13, 25, 26]. If participants' goals did change frequently, it would put the high-momentum operationalizations at a considerable disadvantage, because a high-momentum operationalization considers cues associated with a mix of goals, some of which are no longer relevant.

These results underscore the importance of understanding a predator's foraging momentum in operationalizing Reactive IFT. For these developers engaged in debugging, low-momentum was apparently better at inferring goals, but depending on the foraging context (e.g., government agents performing intelligence analyses or end users debugging spreadsheets), a higher foraging momentum may produce better outcomes. One limitation of the current study is that we examined only two values of δ. Future studies could further investigate the effect of momentum.

Changes in Foraging Behavior over Time

Changes in participants' foraging behavior as the task progressed may have been responsible for the algorithms' improved hit rate in the second task period. Studies have shown that developers pursue different kinds of information [6, 25, 30] and engage in different types of activities [15] as a task progresses. In our study, the participants may have had difficulty finding strong scents in the earlier stages of debugging. As they foraged, they may have homed in on places with stronger scent. Since our algorithms relied on scentful cues to approximate goals, they might have been less accurate earlier in the task when the participants were navigating through low-scent methods, and became more accurate as scent increased.

One possible way to handle low-scent periods is to switch algorithms during such periods. For instance, an algorithm might detect changes in a user's foraging momentum and swap in the most appropriate algorithm for approximating the user's goal. Open questions for future research include how to detect a user's shifts in momentum and how to make recommendations in the absence of scentful cues.

Beyond Word Cues

Although not reaching statistical significance, the hit-rate results showed a tendency to favor PFIS-R over TFIDF-R, and this tendency triangulates with the suggestive difference between PFIS-R and TFIDF-R in the opinion questionnaire results (Figure 8a). This tendency was consistent with the qualitative data. We observed many participants following call dependences while debugging. Moreover, we saw one instance (Participant 6) where a method that would have satisfied the participant's goal contained none of the words that the participant followed scent from.

The implication for tools is that when operationalizing Reactive IFT for a new context, the tool may need to consider other cues in addition to words. Word cues exclusively have dominated the work on web foraging, and they may be particularly effective in that context due to the large volume of unstructured natural language text that web pages contain. However, in a context like programming, where there is less natural language text and more structure, other types of cues may be valuable.

One limitation of the current study is that we only investigated two models for using word and structural scent. Future work may investigate other models that analyze other forms of scent or that are tailored to the needs of different users and different contexts.

CONCLUSION

This paper is the first to investigate how to bring Reactive Information Foraging Theory to recommender tools for information-intensive, ill-structured problems. The algorithms we investigated with professional software developers in a debugging task showed that:

- RQ1: Surprisingly, assuming low foraging momentum (using only one navigation to inform its choices) produced better recommendations than those produced by assuming high momentum (the past ten navigations).
- RQ2: There was suggestive but inconclusive evidence that participants found the tool to be more valuable when it used words + structure than when it used words only.
- RQ3: The tool's recommendations were more useful to participants later in the task, suggesting that tools may need to be sensitive to shifts in users' foraging behavior.
- RQ4: Recommendations helped participants by revealing useful places that the participants were unaware of and also by facilitating navigation to known places.

Most importantly, this study has demonstrated Reactive Information Foraging Theory's potential as a basis for the design of tools that help people solve ill-structured problems. We believe that such tools could help people in contexts beyond software development—for example, helping users debug spreadsheets, security analysts understand relationships among cartels, and the myriad other people who wrangle with complex, ill-structured problems every day.

ACKNOWLEDGMENTS

We thank our participants for their help, IBM for support under an OCR grant, and AFOSR for support under FA9550-09-1-0213. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of AFOSR, IBM, or the U.S. Government.

REFERENCES

1. Baeza-Yates, R., Ribeiro-Neto, B. *Modern Information Retrieval*, Addison Wesley Longman, 1999.
2. Bragdon, A., Zeleznik, R., Reiss, S., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, J. *Proc. CHI*, ACM (2010), 2503–2512.
3. Chi, E., Pirolli, P, Chen, K., and Pitkow, J. Using information scent to model user information needs and actions on the web. *Proc. CHI*, ACM (2001), 490–497.
4. Chi, E., Rosien, A., Supattanasiri, G., Williams, A., Royer, C., Chow, C., Robles, E., Dalal, B., Chen, J., and Cousins, S. The Bloodhound project: Automating discovery of web usability issues using the InfoScent simulator, *Proc. CHI*, ACM (2003), 505–512.
5. Cubranic, D, and Murphy, G. Hipikat: Recommending pertinent software development artifacts. *Proc. ICSE*, ACM/IEEE (2003), 408–418.
6. DeLine, R., Khella, A., Czerwinski, M., and Robertson, G. Towards understanding programs through wear-based filtering. *Proc. Softvis*, ACM (2005), 183–192.
7. Fu, W. and Pirolli, P. SNIF-ACT: A cognitive model of user navigation on the World Wide Web, *Human-Computer Interaction 22*, 4 (2007), 355–412.
8. Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., and Kwan, I. End-User Debugging Strategies: A Sensemaking Perspective. *ACM Trans. Comput.-Hum. Interact.* (to appear).
9. Gross, P. and Kelleher, C. Non-programmers identifying functionality in unfamiliar code: Strategies and barriers. *Proc. VL/HCC*. IEEE (2009), 75–82.
10. Hill, W., Hollan, J., Wroblewski, D., McCandless, T. Edit wear and read wear. *Proc. CHI*, ACM (1992), 3–9.
11. Jakobsen, M. and Hornbaek, K. Evaluating a fisheye view of source code. *Proc. CHI*, ACM (2006), 377–386.
12. Kersten, M. and Murphy, G. Mylar: A degree-of-interest model for IDEs. *Proc. ASOD*, ACM (2005), 159–168.
13. 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*, (2006), 971–987.
14. Lawrance, J., Bellamy, R., Burnett, M., and Rector, K. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks, *Proc. CHI*, ACM (2008), 1323–1332.
15. Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., and Fleming, S. How programmers debug, revisited: An information foraging theory perspective, *IEEE Trans. Soft. Eng.* (to appear 2012).
16. Lawrance, J., Burnett, M., Bellamy, R., Bogart, C., and Swart, C. Reactive information foraging for evolving goals. *Proc. CHI*, ACM (2010), 25–34.
17. Nielsen, J. Information foraging: Why Google makes people leave your site faster. http://www.useit.com/alertbox/20030630.html. June 30, 2003.
18. Olston, C. and Chi, E. ScentTrails: Integrating browsing and searching on the web. *ACM Trans. Comput.-Hum. Interact. 10*, 3 (2003), 177–197.
19. Parnin, C. and Gorg, C. Building usage contexts during program comprehension. *Proc. ICPC*, IEEE (2006), 13–22
20. Piorkowski, D., Fleming, S. D., Scaffidi, C., John, L., Bogart, C., John, B. E., Burnett, M., and Bellamy, R. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. *Proc. VL/HCC*, IEEE (2011), 109–116.
21. Pirolli, P. Computational models of information scent-following in a very large browsable text collection. *Proc. CHI 1997*, ACM Press (1997), 3–10.
22. Pirolli, P. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford Univ. Press, 2007.
23. Pirolli, P. and Card, S.K. Information foraging. *Psychological Review 106*, 4 (1999), 643–675.
24. Resnick, P. and Varian, H. Recommender systems. *Commun. ACM 40*, 3 (1997), 56-58.
25. Sillito, J., Murphy, G., and De Volder, K. Questions programmers ask during software evolution tasks. *Proc. FSE*, ACM (2006), 23–34.
26. Sillito, J., De Voider, K., Fisher, B., and Murphy, G. Managing software change tasks: An exploratory study. *Proc. ISESE*, IEEE (2005), 1–10.
27. Sinha, V., Karger, D., and Miller, R. Relo: Helping users manage context during interactive exploratory visualization of large codebases. *Proc. VL/HCC*, IEEE (2006), 187–194.
28. Spool, J., Profetti, C., and Britain, D., Designing for the scent of information, *User Interface Engineering*, (2004).
29. Storey, M, Best, C, Michaud, J, Rayside, D, Litoiu, M, and Musen, M. SHriMP views: An interactive environment for information visualization and navigation. *Ext. Abstracts CHI '02*, (2002), 520–521.
30. Wiedenbeck, S. and Evans, N. Beacons in program comprehension. *ACM SIGCHI Bulletin 18*, 2 (1986), 56–57.