# PFIS-V: Modeling Foraging Behavior in the Presence of Variants

**Sruti Srinivasa Ragavan[1], Bhargav Pandya[1], David Piorkowski[1, 2], Charles Hill[1], Sandeep Kaur Kuttal[3], Anita Sarma[1], Margaret Burnett[1]**

[1]Oregon State University
Corvallis, OR, USA
{srinivas, pandyab, hillc, anita.sarma, burnett}@oregonstate.edu

[2]IBM Research
Yorktown Heights, NY, USA
david.piorkowski@ibm.com

[3]University of Tulsa
Tulsa, OK, USA
sandeep-kuttal@utulsa.edu

## ABSTRACT
Foraging among similar variants of the same artifact is a common activity, but computational models of Information Foraging Theory (IFT) have not been developed to take such variants into account. Without being able to computationally predict people's foraging behavior with variants, our ability to harness the theory in practical ways—such as building and systematically assessing tools for people who forage different variants of an artifact—is limited. Therefore, in this paper, we introduce a new predictive model, PFIS-V, that builds upon PFIS3, the most recent of the PFIS family of modeling IFT in programming situations. Our empirical results show that PFIS-V is up to 25% more accurate than PFIS3 in predicting where a forager will navigate in a variationed information space.

### Author Keywords
Variants; Information Foraging Theory.

### ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous

## INTRODUCTION
When engaged in computer-supported creative tasks—such as graphic design, creating a presentation, exploratory programming, or writing this CHI paper—people often need to build upon and compare multiple ideas, fit them together and save the intermediate steps [1]. One effect of such exploration can be the creation of *many*, *many* variants of the same artifact (e.g., image with different colors) [2, 12, 45].

People need to find (or re-find) and compare these earlier variants of the artifact, especially when things go wrong, they reach a dead-end, or they want to opportunistically harvest specific bits and pieces of prior work. For example, working in such ways with multiple variants is especially common in exploratory programming, where programmers—novices and experts—experimentally blend writing new code with reusing bits and pieces of code from different resources, including earlier variants [1, 2, 18, 27, 47].

Finding one's way to the right information in a large information space is difficult—even without the presence of multiple variants. For example, one study reported programmers spending about 35% of their time on just the mechanics of navigating a single variant of source code [19], and another study reported that, when other aspects of information seeking were also taken into account, programmers spent an average of 50% of their time navigating [34].

Information Foraging Theory (IFT) [36] helps explain how people spend their time in their information seeking activities. IFT proposes that a person seeking information follows the "information scent" perceived from the signposts ("cues") to locations of potentially useful information (patches), similar to the way predatory animals in the wild follow the scent to their prey. Since foraging in the presence of multiple variants *adds* a need for the forager to cognitively discern differences among very similar cues and patches, it stands to reason that the cost to foragers in a variationed information space will be higher than when patches and cues are more unique and distinctive.

Indeed, a recent study observed that foragers in a variationed information space adopted additional foraging mechanisms and strategies in the presence of multiple, similar variants [41]. These initial observations suggest the need for a deeper understanding of how well IFT can explain people's behavior in such a foraging situation, and how we might then harness IFT to build tools to support variations foraging. Computational models allow us to do this: they help researchers empirically validate concrete hypotheses derived from study observations and guide tool-building.

Toward this end, we present a new computational model of foraging that is able to account for multiple similar variants in an information space. Our new model, called PFIS-V (PFIS for Variation Space), builds on the PFIS (Programmer Flow by Information Scent) family of predictive IFT models for programming situations [32]. We chose programming as our domain because, as mentioned earlier, variations are a commonly-occurring phenomenon in this already information-seeking-intensive domain; however,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*CHI 2017,* May 06-11, 2017, Denver, CO, USA
© 2017 ACM.
ISBN 978-1-4503-4655-9/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3025453.3025818

the theoretical foundations should apply to other domains (e.g., web design, text documents, slide decks, etc.) as well.

We begin by modeling the properties of variants—such as similarities and differences—that users attended to during their variations foraging, leading us to our first RQ:

*RQ1: How can we account for variants in computationally modeling programmers' foraging behavior?*

We then empirically investigate the effectiveness of our computational model using the data collected from a prior study on variations foraging [41] to answer:

*RQ2: How effective is our new computational model?*

## BACKGROUND AND RELATED WORK

### Variants

Working with variants is a common phenomenon in exploratory, creative tasks. Researchers have built several tools to support variants in both non-programming (such as graphic design [11, 12, 45], personal information management [16]) as well as programming domains (e.g., web design [18, 20], professional software development [6], end-user programming [22, 13]). However, none of these tools (except [13]) support variations in exploratory programming.

Prior work on programmers reusing code from other variants during exploratory programming defined a program *variant* as a "*syntactically valid program that occurs together with other similar, related programs*" [41]. For example, if a programmer took the source code of a website and modified the background color of the home page, the resulting new program is a variant of the original program. The results, grounded in Information Foraging Theory (IFT), explain programmers' foraging behavior while seeking the right variant (and the relevant code snippets within) to reuse. We build on this prior work and use IFT to computationally model programmers' variations foraging.

### Information Foraging Theory (IFT)

Information Foraging Theory is a theory of how people engaged in information-intensive tasks seek information. IFT is based on optimal foraging theory that explains how predators hunt for their prey in the wild [36]. The theory was first developed to explain people's information-seeking behavior in large document collections. Since then, it has been applied to explain user behavior in several other domains like web browsing and programming.

IFT borrows constructs from the optimal foraging theory: a person seeking information in an information environment (e.g., programmer searching for a code snippet in an IDE) is similar to a predator seeking prey in a foraging ground (*predator* = person, *prey* = information, *foraging ground* = information environment). The prey might be present in locations in the information environment called *patches* (e.g., files, classes or methods) and the patches might be connected to each other via *links* (e.g., IDE shortcut to go from one method to another). The network of patches and links together is called the *topology*.

Associated with the links is information that acts as *cues* for the predator: a cue tells the predator what might be at the other end of the link, thereby, becoming signposts to their prey. For example, the words in a method name act as cues about what the method does. The predator uses the scent emanating from these cues to reach the information they seek, just like animals sniff their way to their prey.

One of the ways in which IFT has been applied is to computationally model user behavior during different information-seeking tasks [3, 4, 10, 23, 37, 38, 39]; such models have informed the design of tools and interfaces for people engaged in information-intensive tasks. For example, the WUFIS [3] model uses the structure and content of web sites to predict which page a user would navigate to; this in turn helped lay the foundations for layout and information design in websites [43] and web-search engines [28].

Recently, IFT has been applied to programming tasks such as requirements tracing, debugging, maintenance and IDE design [9, 21, 23, 25, 29, 34, 35, 40, 31]. Researchers have built computational models to predict, and thus explain programmers' navigation in source code. The first such model, namely PFIS [24], used source code and task descriptions to predict which classes a programmer, working on a given task, would navigate to. PFIS was then extended to PFIS2 (and later refined to PFIS3 [32]) to model "reactive IFT" [26], that accounted for the evolving foraging goals of programmers as they perform their task. Piorkowski et al. used these models to build a system that recommends parts of code relevant to the current foraging goals (and sub-goals) [33]. However, the PFIS family so far has only accounted for a single program variant and is insufficient to explain foraging in a variationed information space.

### Predicting programmer navigations

Navigating source code is nontrivial [19]: one of the ways researchers have attempted to help programmers navigate source code is by recommending source-code locations that the programmer might want to navigate next. To make their recommendations, these tools use different heuristics. Several tools use the historical activity of the programmer (e.g., past visited locations) to make a prediction [8, 14, 17, 30] while others use textual similarity (to bug report or current location), structure of the program (such as method invocations) [7, 15, 42, 44] or a combination of these [33]. However, none of these tools consider the case of a programmer navigating through multiple program variants.

## MODELING PROGRAMMER NAVIGATIONS

Computational models serve two purposes: (i) they provide a way to validate hypotheses about programmers' foraging behavior, and (ii) the models can be directly used for theory-based tool building. We build on an existing family of predictive models, namely PFIS (Programmer Flow by Information Scent) [24], that model programmer's foraging in a *single* program variant. We chose PFIS because it is grounded in IFT, which is the underlying theory for several computational models of user behavior in several domains

[3, 10, 23], including programming activities like debugging [25] and maintenance [24].

**PFIS3:** Before we describe PFIS3 (the most recent model in the PFIS family), note that predictive models contain two parts: (i) the data model representing the program—its methods, words and links, and (ii) the algorithm that uses the data model to predict programmers' navigations in the program. In the rest of this paper, we use following terminology: *predictive model* refers to any model that predicts programmer navigations; *data model* refers to the underlying representation of the program; and the *algorithm* is responsible for making predictions using the data model (Predictive Model = Data Model + Algorithm).

PFIS3's data model operationalizes the topology of the information space (network of patches and links) as a graph. The information patches (here, methods) are represented as nodes (called *patch nodes)* while links between patches are represented by edges. A link between two patches exists when a programmer can go from one patch to another in a single-step IDE action (e.g., following an IDE shortcut to go to method definitions, scrolling to adjacent methods). The words in patches are modeled as nodes (called *word nodes*), while edges between patch nodes and word nodes indicate that the patch contains the word.

Figure 1 shows a program (left) represented in the data model (right). For each method (patch) in the program (e.g., `sum`, `count`, `average`), there is a patch node (blue ellipse) in the graph, and for every word in the program except reserved keywords (e.g., `numbers`, words in method names), the graph contains a word node (red square).

The method `average` calls methods `sum` and `count`, and a programmer can navigate between these methods using IDE shortcuts; therefore, these patches are linked via an invocation edge (labeled "*inv*"). Similarly, a programmer can navigate from the method `count` to the methods `sum` and `average` by scrolling; therefore, these patches are linked via adjacency edges (labeled "adj"). Thus the links in the graph model the environment's navigation affordances.

Let us look at the method `average`: the patch contains words like average (its name), numbers (parameter), and its content (calls to `sum, count`); these words in the patch serve as cues for a programmer foraging in this patch. The dashed edge (---) between the patch node *average* and the word node *numbers* indicates that the patch contains the word. Since the word `numbers` is also found in patches `sum` and `count`, the *numbers* word node is also connected to *sum* and *average* patch nodes via dashed edges (---).

The PFIS3 algorithm uses the data model to predict programmers' between-patch navigations. It does so by computing the "scent" of links from the programmer's current patch to all other patches the programmer has seen thus far. First, the algorithm activates some patches with initial weights based on the programmer's current location. The algorithm then spreads this activation to other patches based on source-code cues (words), the programmer's past navigation history, and the topology of the information space. The patches are then ranked by their resultant weights, and the algorithm returns the patch with the lowest rank as its prediction for the programmer's next navigation [33].

Let us look at an example (based on Figure 1) of the algorithm's working. Consider a programmer currently in method `count.` Although both `sum` and `average` are *adjacent* patches to `count`, the PFIS3 algorithm spreads higher activation to `average` than `sum`. This is because (a) there is an additional *invocation* link between `count` and `average`, and (b) the patch *`average`* has more words (*numbers*, *count*) in common with `count,` than `sum` (only *numbers*). Therefore, the algorithm predicts a navigation from `count` to `average`. Note that PFIS3 also takes pro-

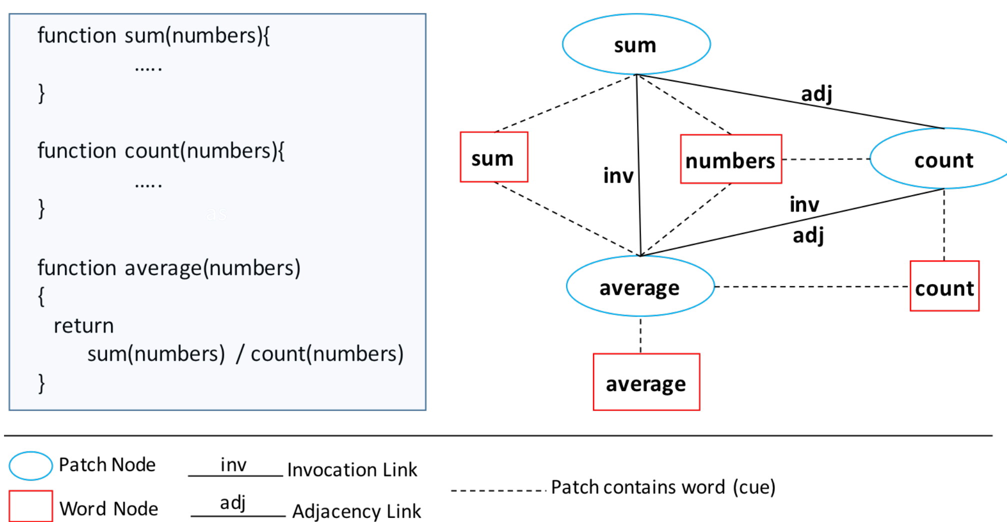



**Figure 1. The PFIS3 data model operationalizes IFT: patches (methods) and words in source code are represented as nodes while links between patches and "patch contain word" relationships are represented by edges.**

grammers' navigation history into account for making predictions, which we ignore here for illustration purposes.

While PFIS3 or other IFT models predict user behavior in a single variant of an artifact, none of them consider multiple variants. This leads us to our first research question:

*RQ 1: How can we account for variants in computationally modeling programmers' foraging behavior?*

We answer RQ1 in two stages: RQ 1a addresses representation of variants in data models, and RQ 1b focuses on algorithms that account for variants while making predictions.

*RQ 1a: How do we represent variants to computationally model foraging in the presence of variants?*

The PFIS3 data model can represent a *single* variant of a program. We extend this data model to represent *multiple* variants in *four* different ways, each making different assumptions about programmers' foraging among variants.

*Variant-unaware data model*

The *variant-unaware* data model is very similar to the PFIS3 data model: it represents patches (along with associated cues and links) from multiple variants in the same way PFIS3 represents a single variant of a program (Recall that a variant is an *entire copy* of a program). Such a data model is unaware of the properties of variants (such as the similarities and differences between variants), which programmers are aware of, and use in their foraging [41].

Figure 2(a) shows the variant-unaware representation of a program with four variants. We represent methods in the program using letters A, P, Q, R and S: nodes represent patches and edges represent links between them. A programmer starts with the first variant and makes changes to method R (whereby, $R' = R + \Delta R$) to create a new variant (and subsequently creates other variants too). While a programmer may be aware that patches R, R', and R" (or all of P) are similar, related patches across different variants, the variant-unaware model is unaware of these properties.

Since this data model captures the navigation affordances currently available in IDEs (and our study environment [41]), we use it as the baseline for our comparisons.

*Variant-aware data model*

One property of variants is that they have similar patches: prior results showed that programmers capitalized on this property while foraging [41]. We modeled this property by introducing "*variant-of*" links between *similar* patches across variants, thereby making it "variant-aware". We define two patches (in different variants) to be *similar*, if they have the *same fully-qualified names* (folder, file, class and method names) relative to the variant that contains them.

See the variant-aware data model in Figure 2(b): the "variant-of" links (dotted lines) between all patches P, or R, R', and R" indicate similar patches across different variants.

*Variant and equivalence aware data models*

Often, a small change in one variant (such as font color and size) might result in a new variant of a program. This leads to a lot of not just similar, but *identical* (exactly the same) patches among temporally close variants [41]. In a programming context, this means that methods can have the same fully-qualified names (folder, file, class *and* method names), as well as *exactly the same content* (parameter names, definition and words) across different variants.

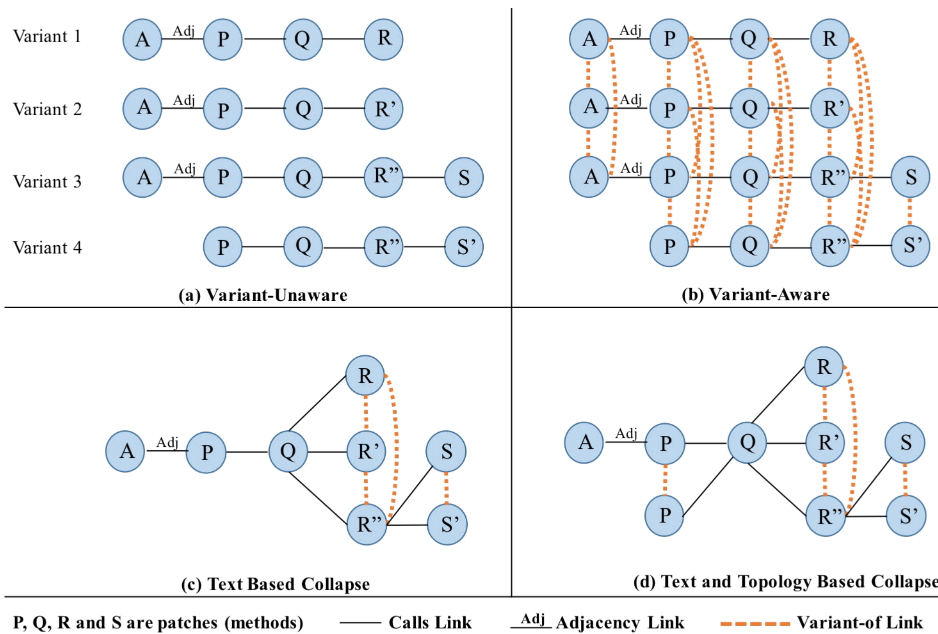



**Figure 2. In order to introduce variant-awareness, we added "variant-of" edges (dotted lines) between similar patches in different variants (b). Further, we collapsed equivalent patches based on text-based similarity (c), or text-and-topology-based similarity (d).**

From a foraging perspective, this means that the cues and the scent from these identical patches are also the same; hence, it does not matter which of these identical patches a programmer forages in. Since these patches are all equivalent to the programmer, we call them "*equivalent*" patches.

From a modeling perspective too, it does not matter which one of these equivalent patches a predictive algorithm predicts. Therefore, we collapse multiple equivalent patches into one super-patch to form a variant-and-equivalence-aware data model. We do this in two ways:

In *text-based equivalence* (denoted by suffix t), we call two patches equivalent if their contents are identical, i.e., they have exactly the same text. From a forager's perspective, this models the case where a programmer *does not* differentiate between patches (and treats them as equivalent) as long as they have the same content.

However, programmers *might* differentiate patches across variants even though they might have the same content! Consider the following scenario: two variants of a program contain identical patches P. In the first variant, there is a patch A that is right above (adjacent) to P, while there is no adjacent patch A in the second variant. In some situations, a programmer foraging in the two variants might perceive the two patches as different. In fact, prior studies on programmer navigation have shown that source-code structure is an important factor in modeling programmer navigations [32]. This leads us to an alternate definition for equivalence:

In *text-and-topology-based equivalence* (denoted by suffix t,t), two patches are equivalent if: (a) they have exactly the same content, *and* (b) their neighbors (patches linked by invocation and adjacency links) in the topology are *similar* (not necessarily identical).

See Figure 2 (b): patch P has links to patch A in the first three variants, but not in the last variant. The variant-and-equivalence-aware(t) only considers text similarity; therefore, patch P from all variants are collapsed into one single patch in Figure 2(c). On the other hand, the variant-and-equivalence-aware(t,t) considers similarities in both text as well as topology: therefore, it collapses the patch P in the first three variants (having same neighbors A and Q) alone into one node, while excluding the patch P in the last variant (since it has no neighbor A).

Notice that in both cases the variant-and-equivalence-aware data models preserve the notion of similar patches (variant awareness), denoted by the "variant-of" links between similar patches in both Figure 2(c) and (d).

One concluding remark about the two variant-and-equivalence-aware data models is that they represent variations to an artifact by only capturing the differences between the variants. This is unlike the variant-unaware or variant-aware data models which also capture redundant, similar patches. For example, in Figure 2, both variant-unaware and variant-aware data models retains four copies of patch Q, even though Q has not changed across variants (there is no Q'), whereas the two variant-and-equivalence-aware representations contain only one patch Q. This result can be directly applied in practical cases where variants need to represented or stored.

Having thus addressed the data modeling problem for program variants (RQ 1a), we next focus on the predictive algorithm (RQ 1b). Before we proceed to develop a new algorithm to predict programmer navigations in a variationed information space, we first modify PFIS3 to predict programmer navigations using the above four data models.

PFIS3 models foraging in a single program variant where a patch node in the data model represents a single patch in the program; therefore, the PFIS3 algorithm does not distinguish between patches and patch nodes. However, the PFIS3 approach fails for variant-and-equivalence-aware data models where one patch node might represent many equivalent (collapsed) patches. Therefore, we tweaked PFIS3 to distinguish between patches and patch nodes, and predict *patch nodes* in the graph instead of *patches* in the program. With this tweak, PFIS3 can predict programmer navigations in four configurations using the four data models; these serve as our baseline for comparisons.

Although PFIS3 can now predict using data models representing variants, its algorithm does not consider the properties of variants that programmers capitalize on, during their foraging [41]. Therefore, we extend the PFIS3 algorithm to account for these properties of variants in order to accurately model programmer behavior.

*RQ 1b-How can we predict programmer navigation in a variationed information space?*

PFIS3 models programmers' navigations to only patches that s/he has already visited (or seen): all other navigations are considered "*unknown*" because the programmer has never seen them. However, in the presence of variants, even though a programmer has not seen a particular patch, s/he might know about it if s/he has already seen a similar patch in a different variant. We therefore extend PFIS3 to PFIS-V (PFIS for Variants) to model this phenomenon that is unique to variations foraging.

Consider a programmer – let us call her Alice – foraging in a method `count` in variant V1 of a program. She then navigates to a different variant V2 and again sees a method `count` in V2. Even before Alice makes a decision to navigate to, or forage in the method `count` in V2, she already has some prior knowledge about the method; therefore, the method `count` is not entirely unknown to Alice. IFT assumes that an information forager makes rational decisions [37]; therefore, when Alice does decide to navigate to the method `count` in variant V2, it stands to reason that she has one of the two expectations: the method `count` in the two variants are identical, or, they are both different (depending on her foraging goal).

**Definitions**:

- Patch set $P$: set of all patches in the topology that the programmer has seen so far.
- Word set $W$: set of all words in all patches in $P$.
- Graph $G = (N_p \cup N_w, E_p \cup E_w)$, where,
  - $N_p$: set of nodes representing patches in $P$ (a patch node can represent a single non-collapsed patch or multiple equivalent patches when collapsed).
  - $N_w$: set of nodes representing the words in $W$.
  - $E_p$: set of edges between two patch nodes, when the patches are linked by an adjacency, invocation or a "variant-of" link.
  - $E_w$: set of edges between a word node and a patch node, where the patch contains the word.
- Navigation history $H$: sequence of patches to which the programmer has navigated so far.

**Steps to predict the (k+1)$^{th}$ patch in H:**

- If programmer has not seen exact patch earlier:
  If programmer has seen a similar patch $P_s$: assume content of $(K+1)^{th}$ patch is exactly similar to $P_s$.
  Else, return "unknown"
- Set activation for each node in $G \rightarrow 0$.
- For the $k^{th}$patch $p$ in $H$ and $|H| - \delta < k$,
  - Increment activation for patch node for $p$ by $0.9|H| - k$.
- Spread activation ($\propto = 0.85$, edge weights=1) such that only word nodes receive activation.
- Spread activation ($\propto = 0.85$, edge weights=1) such that only patch nodes receive activation.
- Rank the patch nodes in the decreasing order of activation.
  - If $t$ patch nodes are tied at rank $r$, assign $rank = [r + \frac{t-1}{2}]$ to all $t$ patch nodes.
- Return the rank for the node representing the $(k+1)^{th}$ patch.

**Figure 3. The PFIS-V's algorithm.**
**(Lines in blue indicate the differences from PFIS3)**

Indeed, prior results showed that programmers looked for similarities and differences (based on their foraging goals) while foraging among similar patches in different variants [41]. From the perspective of modeling such programmer's behavior, there is no way of knowing what difference the programmer expects to find (between similar patches in different variants); hence the cues and scents for such a navigation cannot be modeled until the programmer actually navigates to the patch. Therefore, PFIS-V assumes and models the case that the programmer expects the two similar patches to be identical, as shown in Figure 3.

Recall that PFIS-V (and PFIS3) can predict programmer navigations in four configurations (using the four data models): we refer to these configurations using the notation "PFIS-V / data model", e.g., PFIS-V/variant-unaware.

## EVALUATION

### Methodology

We evaluated PFIS-V using the data containing over 650 click-based navigations [32], collected in our prior study with seven novice programmers [41]. Participants in the study made changes to a JavaScript-based game called Hextris, working in Cloud9, a web-based IDE, on a programming task that lasted 50 minutes. Participants' IDE actions as well as their screen interactions were recorded using Cryolite, a Cloud9 logger [48] and screen capture software.

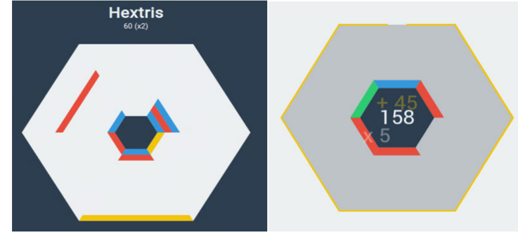



**Figure 4. Participants were asked to move the score and multiplier above the hexagon, "like it was before" (Left: Before, Right: After)**

The participants were asked to make the following changes to the latest version of the game (Figure 4(a)): (i) move the score indicator above the hexagon, (ii) move the score multiplier above the hexagon, and (iii) change the score color; all "like it was before" (Figure 4 (b)). The phrase "like it was before" was used in order to avoid explicitly mentioning that the solutions to the tasks were present in earlier variants. There were over 700 variants.

### PFIS-V evaluation

We evaluated PFIS-V by predicting the *between-patch* navigations (navigations between methods)—within or across variants—that were made by participants during the programming task in the above-said study. We then analyzed the accuracy of these predictions, thus answering RQ2:

*RQ 2: How effective is our new computational model, namely PFIS-V?*

We answered this question by considering two aspects:

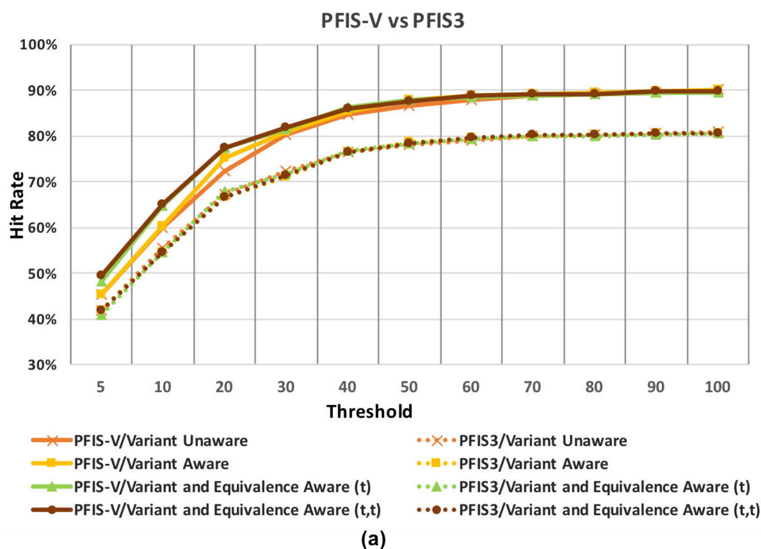
#### *Unknown rates*

Recall that models like PFIS3 and PFIS-V cannot predict programmer navigations in all cases; in some cases, they return "Unknown". The *unknown rate* is the percentage of navigations for which a model returns "Unknown". For example, an unknown rate of 60% means that the model *failed* to predict 60% of all programmer navigations.

Figure 5(b) compares the unknown rates of PFIS-V (blue) and PFIS3 (yellow): PFIS-V had lower unknown rates than PFIS3 for all seven participants (Note that lower is better). In other words, in the presence of variations, PFIS-V could predict (and hence model) programmer navigations in cases *where PFIS3 failed to make a prediction*. On an average, PFIS-V predicted 9.25% more navigations than PFIS3; for individual participants this number was as high as 20.19% (P07). Note that while the unknown rate indicates how often a model fails to make a prediction, it does not measure the accuracy of the predictions.

#### *Hit rates*

*Hit rate,* on the other hand, can measure the accuracy of predictions. Recall that the PFIS family returns the rank at



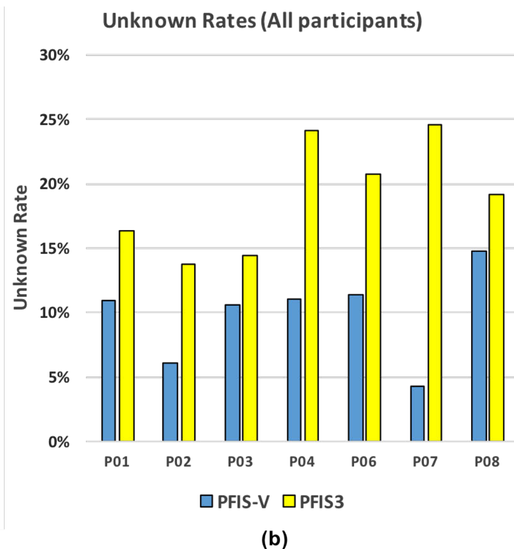




**Figure 5. Not only can PFIS-V predict navigations that PFIS3 cannot, PFIS-V also does so with a higher accuracy than PFIS3.**

which the algorithm predicts the navigation (to be) made by a programmer (as shown in Figure 3). $Hit\ rate\ (N = k)$ read as "hit rate with threshold k", refers to the percentage of navigations that a model predicts at $rank \leq k$. In other words, a "hit rate (N=10) of 60%" means that for 60% of the navigations, the actual navigation made by the programmer was in the top 10 predictions made by the model.

In order to compare the accuracy of predictions made by two models, we compare their hit rates. Following the work of Piorkowski et al. on PFIS3 [32, 33], we generally use hit rate (N=10), unless specified otherwise. See Figure 5(a): the graph is a plot of average hit rates (across all participants) yielded by the different configurations of PFIS3 and PFIS-V, for different thresholds. Irrespective of the underlying data model (line colors), we see that on an average, *PFIS-V (solid lines) had higher average hit rates than PFIS3 (dotted lines)*, suggesting that programmers did capitalize on their knowledge of prior variants, as modeled by PFIS-V.

Although we found that the increase in hit rates when averaged across all participants was modest (about 10%), we saw a much higher increase in accuracy in predictions from PFIS3 to PFIS-V for individual participants. In the case of participant P07, PFIS-V had an increase of 25% accuracy as compared to PFIS3 (PFIS-V modeled with an accuracy of 65%, while PFIS3 attained a 40% accuracy). Further, PFIS-V could model individual participants' navigations with an accuracy as high as 83% (participant P01) [46]. Note that our results are based on only one study in a specific environment; further studies are needed to generalize these results. We now unpack the results of our evaluation further.

### PFIS-V configurations

Recall that PFIS-V can predict in four different configurations, using four different data models. Each of the four data models make different assumptions about the programmers' mental model of the variationed information space and their foraging behavior. To empirically validate our assumptions about programmers' foraging behavior, we further investigated the four PFIS-V configurations.

In Figure 5(a), the different colored solid lines show the average accuracy (hit rates) of PFIS-V for different data model configurations. Focusing only on PFIS-V, we see that the variant-aware (yellow) configuration yielded higher hit rates than the variant-unaware (orange) one, while the variant-and-equivalence-aware configurations (overlapping green and brown solid lines) yielded even higher accuracy than the previous models. Thus, *PFIS-V/variant-and-equivalence-aware(t,t) made the most accurate assumptions about programmers' variations foraging*.

However, further analysis revealed that the above result does not hold for *all* participants: in fact, there existed *two* groups of participants. See Figure 6: the average accuracy of PFIS-V varied with the data models for Group-1 participants (lines do not overlap), while no such differences were observed for Group 2 (the four lines almost overlap).

Before we reason about the differences between these two groups, let us first review the foraging activities of the programmers in the study. Participants had to find and reuse snippets of code from an earlier variant, which included multiple foraging activities: (i) find the right "source" variant from over 700 variants, (ii) forage within that source variant to find the task-relevant patches, and (iii) make changes to the variant containing the latest version of the code ("destination" variant). In IFT terminology, foraging for the right variant from among several variants is called *between-variant foraging*, while foraging within *a* variant to find relevant patches is called *within-variant foraging*. In both these kinds of foraging, participants used different types of *cues* to lead them to their prey [41].

| Partici-pant | Variant-unaware | | | Variant-aware | | | Variant-and-equivalence aware(t) | | | Variant-and-equivalence aware(t,t) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Patch Nodes | Edges | Variant-of Edges | Patch Nodes | Edges | Variant-of Edges | Patch Nodes | Edges | Variant-of Edges | Patch Nodes | Edges | Variant-of Edges |
| P04 | 276 | 623 | 0 | 276 | 817 | 194 | 261 | 752 | 159 | 263 | 756 | 163 |
| P07 | 326 | 821 | 0 | 326 | 2288 | 1467 | 218 | 996 | 443 | 291 | 1780 | 1032 |

**Table 1. For Group-1 participants, the graphs for the variant-and-equivalence-aware are smaller (lesser number of nodes and edges) than the other data model graphs.**

*Between-variant foraging: two behaviors*

Let us now revisit the two participant groups: the primary difference between the two groups of participants is that they used different types of cues for their between-variant foraging. In order to find a right variant for reuse, Group-1 participants (2 out of 7) predominantly looked for differences in the source code between variants, while Group-2 participants (5 out of 7) exclusively used the variants' timestamp, changelogs or differences in the game's output. In IFT terms, Group-1 participants foraged in source-code patches using source-code words as cues, whereas Group-2 participants foraged extensively in non-source-code patches (like output, changelog) and used both textual and non-text-based (visual) cues. Both groups of participants, then, primarily used words in source-code as cues to locate the right patches within the variant (within-variant foraging).

Ideally, a computational model like PFIS-V should aim to model both within- and between-variant foraging of programmers, irrespective of the types of cues they might use or the types of patches they might forage in. Currently, PFIS-V can very well model within-variant foraging, where participants foraged in source-code patches and used as cues words in the source code.

However, in the case of between-variant foraging, PFIS-V modeled the foraging behavior of Group-1 participants (who used the text-based source-code words as cues) very well, but failed to model non-source-code patches and cues (like game's output), which Group-2 participants heavily used. This is because the current state of IFT-based computational models only account for text-based patches and cues in both their data models as well as algorithms. Significant additional work is needed to investigate accounting for non-source-code patches and cue types in IFT-based computational models; we leave this for future work.

Since PFIS-V can only completely model foraging in source code patches, using source-code-inspired cues, we focus our further evaluation of PFIS-V to such foraging behavior, i.e., for Group-1 participants.

*PFIS-V: Modeling foraging in source-code patches*

Figure 6 compares the average hit rates of the different PFIS-V configurations for the two groups of participants. The graph for Group 1 shows four distinct lines: the (average) accuracy of PFIS-V differed across the data models. The accuracy of different PFIS-V configurations were as follows: variant-and-equivalence-aware(t,t) > variant-and-equivalence-aware(t) > variant-aware > variant-unaware.

This indicates that the PFIS-V/variant-and-equivalence-aware(t,t) model was the most accurate model of programmers' foraging behavior, thereby validating the assumptions of that data model: (i) participants recognized and were aware of the similarities in patches between variants (variant-awareness), (ii) participants made comparisons between similar patches in different variants (equivalence-aware), and (iii) they compared similar patches between variants in terms of both their content and code structure (t,t).

The PFIS-V/variant-and-equivalence-aware(t,t) model has another advantage too. Table 1 shows the size of the graphs for the different data models for the two Group-1 participants: the size of the graph (number of nodes and edges) for the variant-and-equivalent-aware data models was found to be smaller than the other two data models. However, note

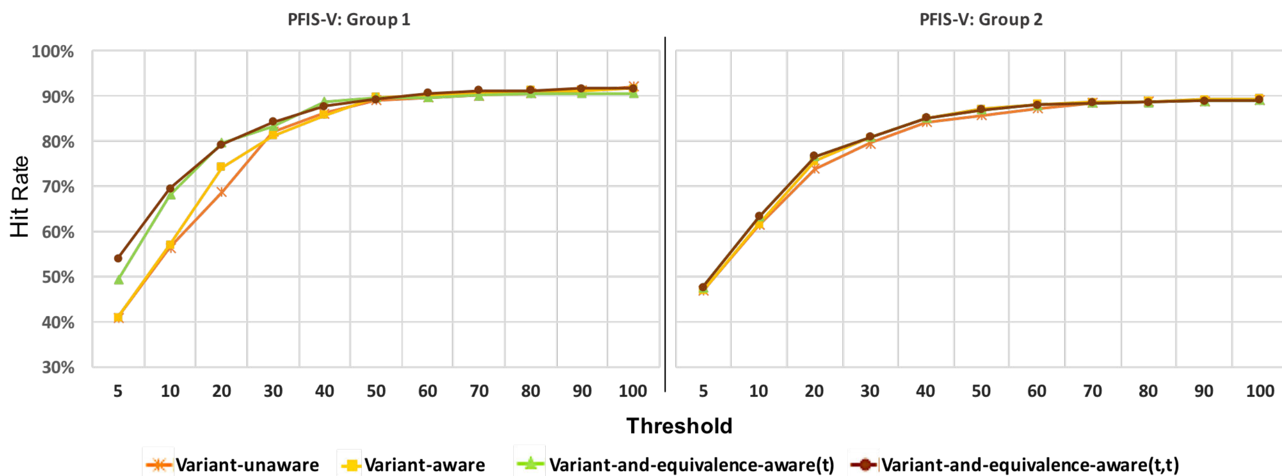



**Figure 6. While the accuracy of PFIS-V varied with the different data models for Group-1 participants, no such differences existed in Group-2 participants.**

that the size of variant-and-equivalence-aware(t,t) was *slightly* larger than variant-and-equivalence-aware(t).

This result about PFIS-V configurations finds its application in tool design: an operationalization of the variant-and-equivalence-aware(t,t) data model might be a compelling option for any tool that supports a large number of variants, especially for text-based artifacts (Group-1 situations). The smaller memory footprint of this model suggests that a similar representation of variants can be space-efficient, while the high modeling accuracy implies that the resultant tool can better address programmers' foraging requirements in an exploratory programming context.

The results of our evaluation of PFIS-V thus answers our second research question (RQ2) on the effectiveness of PFIS-V in predicting navigations: PFIS-V predicted more navigations than PFIS3, for all participants. Particularly PFIS-V modeled Group-1 participants more accurately in both between- and within-foraging scenarios, whereas it failed to model the between-variant foraging of Group-2 participants. These results uncover implications (discussed later) for both IFT-based modeling as well as tool design.

**PFIS-V factors**

An important aspect of models predicting programmer navigations, such as PFIS, is the *predictive factors* (heuristics) of the model. For example, a *recency* factor model predicts navigations based on how recently the programmer has visited that location. Piorkowski et al. identified seven such *factors* (based on prior literature) that can help predict programmer navigations [32].

PFIS-V is a *multi-factor* model: it combines three factors to make predictions, while a *single-factor* model uses only one. The three factors in PFIS-V are: (i) *recency* (did the programmer visit the patch recently?), (ii) *text similarity*, (is the patch similar to bug report or current patch?) and (iii) *source topology* (is the patch linked to current patch?).

PFIS-V combines these component factors by assigning them specific weights. The $p$ and $\propto$ in the PFIS-V algorithm in Figure 3 are indeed weights assigned to different factors: note that the weights are the same as in PFIS3. Studying the predictions made by individual factors can provide insights for tuning these weights in PFIS-V, thereby improving predictive accuracy. To this end, we study the predictions for one Group-1 participant (P07), in the variant-and-equivalence-aware(t,t) configuration (since it yielded highest accuracy, as discussed in RQ2).

Figure 7(a) compares the hit rates (Y-axis) of PFIS-V with its constituent single-factor models. Consistent with Piorkowski et al.'s results in single variant situations [32], we see that *PFIS-V (blue) is a more accurate predictor of programmer navigations than its constituent single factors*.

To see how each factor contributes to PFIS-V's accuracy, consider a scenario where a programmer navigates to a new (earlier unvisited) patch by scrolling to an adjacent method. Since the programmer has *never* navigated to the patch earlier, the recency single-factor model returns "Unknown". However, the source-topology factor that considers adjacent patches can predict this navigation. Similarly, in variations foraging scenario, the "variant-of" links (source topology) or text similarity can predict navigations to similar patches in different variants. Thus, when one factor fails to predict, another factor fills in the gap and makes a prediction: PFIS-V can make accurate predictions due to such synergy.

See Figure 7: for P07, recency yields higher hit rates (which is good) as well as has higher unknown rates (a limitation). On the other hand, although text similarity and source topology yield lower hit rates, they predict navigations where

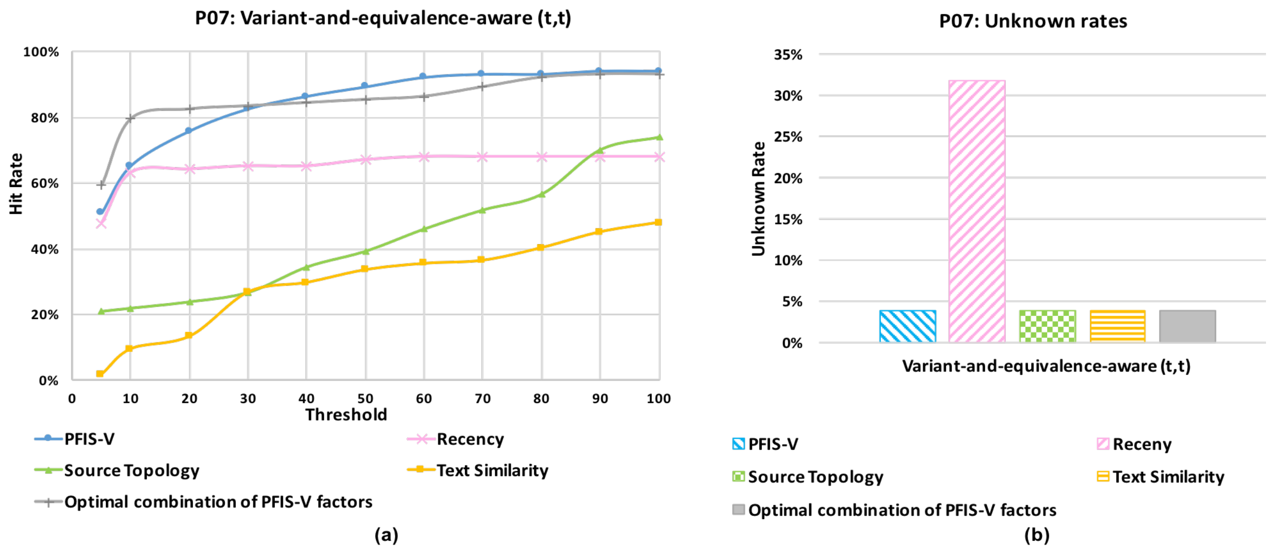



**Figure 7. PFIS-V combines three factors (recency, source topology, text similarity): a synergy of these three factors leads to better predictive accuracy (higher hit rates (a), lower unknown rates (b)) than that of the individual factors.**

recency cannot. Thus, *no single factor* in PFIS-V outperforms others in predicting programmer's navigations.

Given that a combination of factors in PFIS-V results higher predictive accuracy than just individual factors, we estimated the maximum accuracy that can be obtained by combining them: this serves as a benchmark for PFIS-V. We did this by computing the predictions made by an optimal combination of the three PFIS-V factors. For each attempted prediction, an optimal combination model yields the best prediction made by its component models. For example, if the three PFIS-V factors predicted a navigation with ranks 4, 20 and 30 respectively, their optimal combination model returns 4; therefore, at threshold N=10, it yields a hit (though only one constituent model yields a hit).

Figure 7(a)'s grey line shows the hit rates from such an optimal combination model (for P07). At threshold N=10, the hit rate of the optimal combination model is 60%, while that of PFIS-V is 51%. Further, the two models have the same unknown rates (Figure 7(b)). The marginal difference in hit rates between the two models is an indicator of PFIS-V's success, given that the optimal combination model is only a *theoretical ideal* while PFIS-V is a *practical model*.

## DISCUSSION

### Designing for variants

The presence of variants in large information spaces, such as programs, adds additional cognitive load as people have to forage for differences among *similar* variants. Modern software tools that represent variants (such as Git, a version control system) aim to support this foraging behavior by considering the *equivalence* among variants in their interfaces, i.e., these tools minimize cognitive load on users by leaving out whatever is similar, and only showing the differences between consecutive variants. However, from an IFT perspective, there are missing aspects in these tools. For example, the navigation affordances in tools that support variants are constrained, e.g., the absence of links from one patch to a similar patch across different variants [5].

Our model allows researchers and tool builders to evaluate hypotheses of programmers' foraging behavior in their environment. For example, one can posit that a programmer treats a method as different if the method got moved within a file. Such a hypothesis can be validated by comparing the modeling accuracies of PFIS-V while using text-based and text-and-topology-based similarity.

Our work also has several implications for tools. For example, the higher accuracy of variant-aware data models reveals the importance of navigation affordances between similar patches across variants ("variant-of" links). Similarly, the comparison results of the four data models reveals that the variant-and-equivalence-aware(t,t) model makes the closest assumptions about programmers' foraging. Therefore, tools aiming to support variation foraging can directly import this data model as their underlying data structure to represent variants.

### Non-source code cues

Almost all IFT-based predictive models in programming thus far only model foraging among source-code patches, thereby only considering source-code-word cue types for predicting navigation. This is especially true for the PFIS family of models. However, our study participants used non-source-code patches and cues for their between-variant foraging, in exploratory programming scenarios [41].

In fact, our evaluations showed that there were two distinct types of foragers. Group 1 used source-code cue types and PFIS-V was able to accurately model foraging. However, Group 2 used non-source code cue types (e.g., output and changelogs) during their between-variant foraging, which PFIS-V does not capture in its model. This interfered with PFIS-V's ability to model Group 2's foraging behavior, thereby also limiting our ability to harness the model to understand (and predict) these programmers' foraging.

This reveals a gap in current IFT models—for programming as well as other variationed information spaces. For programming, modeling only source code cue types is insufficient in the context of multiple variants. Expanding to other types of information patches, such as outputs with visual content, mixed-media patches, semantic use of color, etc., can lead to significant new thought about comparing variants. Therefore, IFT-based computational models need to be extended to account for such patches. However, computing similarities and differences between output or test results patches between variants is a non-trivial problem, let alone modeling programmers' foraging behavior heavily involving such visual comparisons, as in our study [41]. We consider this to be an important new research opportunity in the area of computationally modeling variations foraging.

## CONCLUSION

In this paper, we present a new computational model (PFIS-V) to model people's foraging behavior through an information space with variants. Our evaluation shows that PFIS-V predicted up to 20% more navigations than PFIS3, the latest of the PFIS family of models. Further, PFIS-V's predictions were up to 25% more accurate than PFIS3.

Such a computational model provides two benefits to HCI: researchers can use the model to gather evidence to support or refute hypotheses about variations foraging, and the model can guide as well as be directly imported to tools.

Finally, our results point to a challenging open research problem for predicting people's foraging behaviors over variationed information space with multiple types of information patches. We posit that it is time that IFT-based models account for such foraging behavior, which is especially important in variations foraging situations.

## ACKNOWLEDGMENTS

We thank the reviewers for their feedback and our participants for their help. This work was funded in part by NSF 1253786, 1302113, 1314384 and 1559657 and David Piorkowski's IBM PhD fellowship.

## REFERENCES


1. Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09), 1589-1598.*

2. Margaret M. Burnett, and Brad A. Myers. 2014. Future of end-user software engineering: Beyond the silos. In *Proceedings of the on Future of Software Engineering*, 201-211.

3. Ed H. Chi, Peter Pirolli, and James Pitkow. 2000. The scent of a site: A system for analyzing and predicting information scent, usage, and usability of a web site. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '00), 161-168.

4. Ed H. Chi, Peter Pirolli, Kim Chen, and James Pitkow. 2001. Using information scent to model user information needs and actions and the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'01), pp. 490-497.

5. Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. 2015. Software history under the lens: a study on why and how developers examine it. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'15),* pp. 1-10.

6. Paul Clements and Linda Northrop. 2001. *Software Product Lines: Patterns and Practice*. Addison-Wesley Professional.

7. Davor Cubranic, Gail C. Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering* (ICSE '03), 408-418.

8. Robert DeLine, Amir Khella, Mary Czerwinski, George Robertson. 2005. Towards understanding programs through wear-based filtering. In *Proceedings of the ACM Symposium on Software Visualization*, 183-192.

9. Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. In *ACM Transactions on Software Engineering and Methodology*, 22,14: 41 pages.

10. Wai-Tat Fu, and Peter Pirolli. 2007. SNIF-ACT: A cognitive model of user navigation on the World Wide Web. *Human Computer Interaction* 22.4 (2007): 355-412.

11. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology* (UIST '08), 91-100.

12. Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. 2010. D. note: Revising user interfaces through change tracking, annotations, and alternatives. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 493-502.

13. Austin Z. Henley, and Scott D. Fleming. 2016. Yestercode: Improving code-change support in visual dataflow programming environments. *In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'16)*.

14. William C. Hill, James D. Hollan, Dave Wroblewski, Tim McCandless. 1992. Edit wear and read wear. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '92), 3-9.

15. Mikkel R. Jakobsen, Kasper Hornbaek. 2006. Evaluating a fisheye view of source code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '06), 377-386.

16. Amy K. Karlson, Greg Smith, and Bongshin Lee. 2011. Which version is this? Improving the desktop experience within a copy-aware computing ecosystem. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '11), 2669-2678.

17. Mik Kersten and Gail C. Murphy. 2005. Mylar: A degree-of-interest model for IDEs. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development* 159-168.

18. Scott R. Klemmer, Michael Thomsen, Ethan Phelps-Goodman, Robert Lee, and James A. Landay. 2002. Where do web sites come from? Capturing and interacting with design history. In *Proceedings of the SIGCHI Conference on Human factors in Computing Systems* (CHI '02), 1-8.

19. Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. In *IEEE Transactions on Software Engineering.* 32, 12: 971-987.

20. Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: Example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '11), 2197- 2206.

21. Sandeep Kaur Kuttal, Anita Sarma, Gregg Rothermel. 2013. Predator behavior in the wild web world of bugs: An information foraging theory perspective. In *IEEE*

*Symposium on Visual Languages and Human Centric Computing (VL/HCC'13)*, 59-66.

22. Sandeep K. Kuttal, Anita Sarma, and Gregg Rothermel. 2014. On the benefits of providing versioning support for end users: an empirical study. In *ACM Transactions on Software Engineering and Methodology* 21, 2:9.
23. Joseph Lawrance, Rachel Bellamy, Margaret Burnett. 2007. Scents in Programs: Does information foraging theory apply to program maintenance? In *IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC 2007), 15-22.
24. Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. 2008. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '08), 1323-1332.
25. Joseph Lawrance, Christopher Bogart, Margarett Burnett, Richard Bellamy, and Kyle Rector. 2009. How people debug, revisited: An information foraging theory perspective. In *IEEE Transactions on Software Engineering*, 117-124.
26. Joseph Lawrance, Margaret Burnett, Rachel Bellamy, Christopher Bogart, and Calvin Swart. 2010. Reactive information foraging for evolving goals. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '10), 25-34.
27. Brad A. Myers, YoungSeok Yoon, and Joel Brandt. 2013. Creativity support in authoring and backtracking. In *Proc. Workshop on Evaluation Methods for Creativity Support Environments at CHI (CHI '13)*, 40-43.
28. Jakob Nielsen. 2003. Information foraging: Why Google makes people leave your site faster. *Jakob Nielsen's Alertbox*.
29. Nan Niu, Anas Mahmoud, and Gary Bradshaw. 2011. Information foraging as a foundation for code navigation (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11), 816-819.
30. Chris Parnin and Carsten Gorg. 2006. Building usage contexts during program comprehension. In *14th IEEE International Conference on Program Comprehension* (ICPC'06), 13-22.
31. Alexandre Perez and Rui Abreu. 2014. A diagnosis-based approach to software comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension* (ICPC'14).
32. David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E. John, Margaret Burnett, and Rachel Bellamy. 2011. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC'11), 109-116.
33. David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. 2012. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '12), 1471- 1480.
34. David Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. 2013. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '13), 3063-3072.
35. David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. 2015. To Fix or to Learn? How production bias affects developers' information foraging during debugging. In *IEEE International Conference on Software Maintenance and Evolution* (ICSME '15), 11-20
36. Peter Pirolli, and Stuart Card. 1995. Information foraging in information access environments. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '95), 51-58.
37. Peter Pirolli. 1997. Computational models of information scent-following in a very large browsable text collection. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (CHI '97), 3-10.
38. Peter Pirolli, Wai-Tat Fu. 2003. SNIF-ACT: A model of information foraging on the World Wide Web. *User modeling. Springer Berlin Heidelberg*, 45- 54.
39. Peter Pirolli, Wai-tat Fu, Ed Chi, and Ayman Farahat. 2005. Information scent and web navigation: Theory, models and automated usability evaluation. In *Proceedings of HCI International*.
40. Peter Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. 2007. Oxford University Press.
41. Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. Foraging among an overabundance of similar variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (CHI '16), 3509-3521.
42. Vineet Sinha, David Karger, and Rob Miller. 2006. Helping users manage context during interactive exploratory visualization of large codebases. In *Visual*

*Languages and Human-Centric Computing (VL/HCC'06)*, 187-194.

43. Jared M Spool, Christine Perfetti, David Brittan. 2004. Designing for the scent of information, *User Interface Engineering*.

44. Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, Mark Musen. 2002. In *Extended Abstracts on Human Factors in Computing Systems* (CHI EA '02), 520-521.

45. Michael Terry and Elizabeth D. Mynatt. 2002. Side views: Persistent, on-demand previews for open-ended tasks. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology* (UIST '02), 71-80.

46. Sruti Srinivasa Ragavan. PFIS-V. 2016. Retrieved on Dec 27, 2016 from http://web.engr.oregonstate.edu/~srinivas/pfis-v.html

47. YoungSeok Yoon and Brad A. Myers. 2014. A longitudinal study of programmers' backtracking. In Visual Languages and Human-Centric Computing (VL/HCC '14), 101- 108.

48. Natural Programming. Retrieved on August 21, 2016 from http://www.cs.cmu.edu/~NatProg/cryolite.html.