AN ABSTRACT OF THE DISSERTATION OF

Sruti Srinivasa Ragavan for the degree of Doctor of Philosophy in Computer Science presented on August 29, 2019.

Title: Variations Foraging.

Abstract approved:

_____

Margaret Burnett

Information Foraging Theory (IFT) has successfully explained how people seek information in various domains, in turn, informing the design of several tools and information-intensive environments. However, prior research has not explored foraging in the presence of several, very similar variants of the same artifact. Such variants are commonplace in several creative, exploratory tasks such as graphic design, writing and programming.

In this thesis, we evaluate whether and how IFT applies to variants. Using empirical studies and computational models that predict programmers' information foraging among variants, this thesis provides evidence for the applicability of IFT in variations situations and offers new insights for variations-support tools. Along the way, this thesis also demonstrates the benefits of computationally modeling: 1) the hierarchical organization of information environments, 2) variable costs of navigation in an information environment and 3) accounting for non-textual (graphical) information.

.

Variations Foraging

by
Sruti Srinivasa Ragavan

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 29, 2019
Commencement June 2020

Doctor of Philosophy dissertation of Sruti Srinivasa Ragavan presented on August 29, 2019

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Sruti Srinivasa Ragavan, Author

ACKNOWLEDGEMENTS

First and foremost, I'd like to thank my advisor, Dr. Margaret Burnett. She was not only a patient teacher and mentor, but is also one of the kindest bosses I've ever come across (and probably ever will). Thanks for everything, Dr. B. I'll miss you.

I'd also like to thank my committee members, Dr. Anita Sarma, Dr. Cindy Grimm, Dr. Eric Walkingshaw and Dr. David Kling for all their feedback. I'd especially like to thank Anita who has been actively involved in my research, and with whom coauthoring papers has been a lot of fun. Also thanks to Dr. Prasad Tadepalli, who kindly agreed to be an examiner for my Ph.D. preliminary examination.

Most of the research in this thesis was done in collaboration with other fellow grad students and postdocs. Thanks to David Piorkowski, Sandeep Kuttal, Bhargav Pandya, Souti Chattopadhyay and Charles Hill—I've learnt a lot from each one of them.

Thanks to my grad school buddies who made life in Corvallis a lot more fun. A special shout out to Michael Hilton, Amin Alipour, Beatrice Mossiniac, Mihai Codoban, Michael Slater, Behnam Saeedi, Jon Dodge and Santosh "Santy" Suresh.

Thanks to Andrew Anderson and the others in Dr. Burnett's group for accommodating me during the times I was away in Pittsburgh.

Thanks are also due to Nicole Thompson, Alyssa Pautsch, Calvin Hughes, Todd Schechter and the College of Engineering IT teams for all their support.

Finally, I'd like to thank my family and friends. Thanks to mom, dad and my sister for supporting me throughout. Thanks to the husband for making the long-distance situation work and for all the thoughtful help on the homefront. Thanks to my mother-in-law and grandmother-in-law for raising my feminist husband and for being very understanding of the two-body situation. And thanks to my dear friends Kamala, Sharmin, Lavanya, Gayathri and Harish for always being there.

CONTRIBUTION OF AUTHORS

A part of the contributions in Chapter 5 also appears in the M.S. Thesis authored by Bhargav Chandravadan Pandya. Bhargav and I jointly developed the PFIS-H algorithm, brainstorming together and pair programming on the implementation. Bhargav also led the modeling of changelogs in PFIS-H. Souti Chattopadhyay and I pair programmed on the modeling of output patches and the statistical tests reported in Chapter 5.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

Figure Page

LIST OF FIGURES (Continued)

LIST OF TABLES

DEDICATION

To the husband man,
whose countless thoughtful acts over these last three years
have allowed me to focus on school.

CHAPTER 1. INTRODUCTION

We live in what scientists dub "the information age" [27], an era marked by easy and cheap access to abundant information, such as via public libraries and internet. While such abundant and easy-to-access information has made information-centric tasks, such as research and news broadcasting, easier for people in some ways, it also brings with it a fair share of difficulties, perhaps the most important one being tedious information seeking.

The presence of too much information (or "information overload" [28]), the intermingling of valuable information with potentially irrelevant and redundant information (or "information pollution" [29]) and the fact that related information can be scattered across different sources [67], make information seeking difficult, costing both time and cognitive effort [8, 85].

As if these difficulties were not enough, information seeking can become even harder in some scenarios, such as when the information is in an unfamiliar language, or for visually-impaired people, or when there are multiple copies or *variants* of the same information. The latter is the subject of this dissertation.

In computer-supported creative tasks such as writing, graphic design or programming, people often work in an exploratory and incremental manner—trying out various options, copying and modifying a previous solution to create a new solution, and comparing alternatives. Along the way, they keep track of the solutions they explore, along with some intermediate steps, resulting in multiple variants of the same artifact (e.g., several draft manuscripts of an essay, different possible visual designs for a flyer, different UI options for a website). They then revisit those variants to reuse bits and pieces, to compare solutions or to backtrack when things go wrong [6, 7, 93, 99]. However, there can be too many variants and the variants can be very, very similar, thereby demanding additional cognitive efforts to discern the differences between them.

Researchers have recognized such potential difficulties in the presence of variants and have attempted to address them. Some researchers have conducted empirical investigations into the needs of people in various domains involving variants [6, 23,

48, 93], while others have, based on these empirical investigations, gone on to build variant-support tools, such as Pipes Plumber for Yahoo! Pipes [40], D.note for web designs [23], ParallelPies and SideViews for images [94, 95] and Yestercode for LabVIEW visual programs [24]. However, these prior studies and tool evaluations are a-theoretic.

The lack of good theories presents major limitations for us as tool builders. First, without the theoretical *explanations* for why a phenomenon occurs (and occurs the way it does), we as tool builders risk addressing the symptoms of a specific problem as they manifest in a specific situation, instead of addressing the underlying causes of the problem. Second, without the abstractions of a theory and its explanations for why a solution will work, our ability to systematically generalize and reuse solutions from one domain to other domains is limited. Finally, theories are valuable for tool design and evaluation: because a theory can explain why a phenomenon happens, it can also reason in the opposite direction to *predict* what will happen in a given situation; this predictive power can inform the design and evaluation of tools (e.g., how will a person use a tool in a given situation? will a tool solve a problem in a given situation or not, and why?) even before the tool is actually implemented. [35].

Therefore, in this dissertation, we seek the theoretical foundations for how people seek information among variants. Specifically, we evaluate the applicability of an existing theory, namely Information Foraging Theory, to people's information seeking among variants.

Information Foraging Theory (IFT) is a theory of people's information-seeking behavior: it posits that people seek information similar to how wild animals forage for their prey in the wild [69]. In the past, IFT has explained people's information seeking behaviors in document collections, web, software engineering and information visualizations. It has also informed the practical design and evaluation of various tools in these domains [76], including generic design principles (e.g., for web design), thus exemplifying the explanatory, predictive and the generalization capabilities of a good theory described in the previous paragraphs. Encouraged by these prior successes, in this dissertation, we explore the applicability of IFT in variations situations.

Thus, the central thesis of this dissertation is:

IFT can explain and predict people's information seeking in the presence of variants.

In investigating this thesis statement, our work contributes: 1) a theory of variations foraging, 2) empirically-evaluated IFT computational models that accounts for variants, 3) an empirically-evaluated IFT computational models that accounts for hierarchically-organized information and 4) an empirically-evaluated IFT computational model that accounts for non-textual information.

CHAPTER 2. BACKGROUND AND RELATED WORK

Related literature for variations foraging is in two areas, namely variations and information foraging theory.

2.1 Variations

We use the term variations to refer to the phenomenon when, for the same artifact, multiple related copies exist together; each individual copy is a variant. Prior work has recognized two kinds of variants in digital artifacts: 1) serial versions capture the state of an artifact at various points during its incremental development and 2) parallel alternatives capture various alternative solutions or implementations for a problem [93].

Prior studies (e.g., [32, 93]) have revealed that, across domains, people keep track of both serial revisions and parallel alternatives by manually saving copies of their files. However, such file-based provides limited affordances for comparing variants or for editing multiple variants at once. To address these limitations, researchers have proposed various solutions to support managing and working with both serial and parallel variants.

2.1.1 Supporting serial revisions

A long history of versioning digital artifacts is in the domain of software engineering, where developers incrementally develop software—adding new features, fixing bugs and enhancing its UI to be released as part of a newer version. In such incremental development, developers keep track their software release versions; this is commonly accomplished via a numbering scheme (e.g., Mac OSX 10.10) or a model/year combinations (Windows 2000) for naming the versions [88].

To enable such incremental development (e.g., to fix a bug in a particular release), developers also keep track of revisions to their source code. For this, they use a class of tools called version control systems (e.g., Git, SVN).

In a version control system, each individual developer can save or "commit" the latest changes to the project's code; thus, the version control system captures the entire development history of the project. Developers can access the version control

repository to retrieve and integrate each other's latest code, compare revisions or even revert back to a previous version.

Over the years, researchers and tool builders have built different paradigms of version control tools. The first version control system, namely Source Code Control System (SCCS), was primarily command-line based [81] and captured the changes to text in the source-code files. Over the years, other text-based version control tools (e.g., RVS [96], Git [17], Subversion [91], CVS [13], Mercurial [49]) have introduced GUI tools for viewing and committing changes (e.g., offer graphical interfaces). Other tools, such as Smart Differencer [86], capture differences in terms of abstract and concrete syntax trees, instead of plain text. Most of these modern version control tools also integrate with other software development tools, such as the developers' IDE and the project's bug repository.

However, all of these VCS tools suffer a major limitation, namely that developers have to manually decide when and what changes to commit. Such manual control is useful in some situations, such as when a programmer might not want to commit incomplete or buggy code. However, in other situations, manual commits mean that developers might forget to save their changes and might end up with no way of getting back to an earlier state of their program (e.g., to backtrack when things go wrong).

To address this gap, state-of-the-art programming environments (e.g., JetBrains IDEs, Eclipse) automatically keep track of the code revisions an individual programmer makes in the IDE, even before the programmer has committed the changes to version control. In particular, the IDE creates a new revision of the program at specific save points, namely every time a programmer saves, compiles or runs the program or the test suite. A programmer can access these local revisions and compare and revert to an earlier revision, just as with version control revisions. However, unlike version control history, the local history is "local" to the developer (e.g., does not have changes by other developers) and is localized in time (e.g., up to one week).

Another tool that keeps track of local history of a programmer is Azurite [100]. Azurite automatically records, at the keystroke level, the fine-grained edits made by a programmer in an IDE. The tool then presents the edit history of the programmer over

a timeline view. A programmer can navigate through the timeline, compare the program across different points on the timeline and selectively undo (or restore) specific code snippets. Although the primary intent of Azurite is to support backtracking (or selective undo), Azurite also serves as a visualization tool for the local history of programs.

In non-programming domains, people keep track of revisions to their artifacts in several ways. Sometimes, people use different file names (e.g., suffix 1,2,3 and so on to file names) to keep track of revisions to artifacts. Other times, they use version control systems: although version control tools were developed for versioning program code, people use them to keep track of revisions to non-code artifacts also (e.g., text documents, scientific experiment notes). Some people also use special-purpose versioning tools. For example, tools such as Kactus [36] and Versions [97] bring version control tools for designers to version their graphic designs. Finally, tools in various domains (e.g., Google Docs [19], Microsoft Office suite [52], MacOS Time Machine [92]) provide features that allow people to automatically keep track of revisions to artifacts.

Researchers have also built tools to keep track of the provenance of artifacts, as they get cloned and revised over several iterations. Kuttal et al. built Pipes Plumber, a versioning tool for Yahoo! Pipes mashup programming environment, that keeps track of the clone history as well as the post-cloning revisions of mashup programs [40]. Similarly, Jensen et al. built TaskTracer that keeps track of the provenance of files and folders created with the Microsoft Office suite [34]. Even other researchers have proposed that provenance information be persisted as part of file systems, to better support recall, search capabilities as well as to support versioning [53].

As mentioned earlier, most of these tools are a-theoretic and derive from empirical observations. Only recently, we have begun exploring the application of information foraging theory to the design of version control systems [79].

Whereas the above tools focus on keeping track of serial variants, where changes to one version leads to a new version in a linear manner, people sometimes end up with parallel alternatives when working in an exploratory (e.g., to try out different menu

placement options, to tune hyperparameters for a machine learning program). As we discuss next, several tools exist that focus on supporting such parallel alternatives also.

2.1.2 Supporting parallel alternatives

In the programming domain, version control tools provide a feature called branching, where a programmer could branch from a version in several parallel ways. This branching comes in handy when developers want to try out different alternative implementations. Similarly, software teams also use branches to manage multiple editions of their software such as in software product lines [12]: these include managing the source code for different customized versions for different clients, or for different editions (e.g., free community vs. paid commercial versions).

However, branches can be heavy-weight in exploratory situations, where a programmer wants to explore at a "fine-grained" level (e.g., one sorting algorithm vs. another, one font vs. another). First, every time a person wants to explore a new option (e.g., try a new font size), s/he has to create a new branch and then commit the changes for that branch before exploring a second option. Second, since branching is manual, a person might forget to create a branch every time s/he wants to explore an option. Third, a person can work with only one branch at a time; therefore, comparing multiple branches or editing multiple branches of code is not easy. Finally, both versioning and the branches in version control systems deal with the entire program. In contrast, explorations can be local (e.g., only for one method) and do not warrant a new branch for the entire program / artifact [6, 33, 51].

Researchers have worked to solve these gaps and proposed various tools and techniques over the last decade. However, these tools are mostly based on empirical evidence and not based on theory. For example, in scientific programming, scientists conduct different experimental trials, resulting in different variants of the same experiment. To keep track of these variants to their experiments, scientists need to keep track of the changes to code (e.g., configurations and algorithms), inputs and outputs, intermediate results and the infrastructure (e.g., library versions). To support these activities, Guo et al. built IncPy and Burrito [20, 21]. IncPy is a custom python interpreter that keeps track of the data and code execution for each experimental variant; in turn, this data facilitates reuse of intermediate results when a different

variant of the experiment is run in the future. Burrito automatically keeps track of all the files (e.g., code, input, output, input results) related to each experimental variant and allows a scientist to annotate each trial with additional information and insights. The scientist can then view, via a graphical interface, the progression of their experiments together with the relevant copies of their input, results and annotations. Guo et al. also built a bundler, namely CDE, that bundles specific variants of an experiment (including the code, input, output and infrastructure) to be shared with other scientists [21].

Just like in scientific experiments, data scientists working on machine learning programs also need to keep track of the different variants, namely to the algorithm, the parameters and hyperparameters and to the input and output files, before deciding which machine learning model to use for their application. However, studies have revealed that even expert programmers struggle to manage these variants [25, 32]. To address this problem, Kery et al. built Variolite to allow data scientists to experiment with their code within programming IDEs [32]. With Variolite, a programmer can select a code snippet (e.g., method) to create one or more variants for that specific snippet. (In contrast, Burrito creates a new variant for the entire program). A programmer can then plug in one variant instead of another, or even nest variants (i.e., create different variants for a smaller snippet within a variant for a larger snippet). Variolite also allows a program to label their variants, so that a programmer can easily find the variant at a later time.

In contrast to Variolite where a programmer has to manually create a variant, Mikami et al. built a micro versioning tool, where a programmer can simply edit the code snippet without stopping to create a new variant explicitly [51]. The tool automatically records the edits and lists them as alternatives for that code snippet. The tool also groups changes made to related, but disconnected, code fragments into one candidate alternative. (This is achieved via hierarchical variants in Variolite). Both Kery's Variolite and Mikami's micro versioning tool also allow programmers to compare variants and to backtrack specific changes.

A surprising scenario involving parallel variants is in online programming courses. For programming assignments, different students might turn in different solutions, or

different implementations of the same solution, resulting in different variants of the same program. In grading such submissions, a TA might want to view a stack of similar solutions and provide similar feedback to them at once, even when there might be some differences (e.g., different variable names). To support such manipulations, Glassman et al. built Overcode [18] that uses a combination of static analyses and program traces to cluster similar program variants into stacks. A grader can use OverCode to look at a single representative submission for each stack of similar submissions and provide the same grade and/or feedback for the entire (without having to grade multiple identical solutions).

In non-programming domains, variants mostly occur in exploring alternatives. Studies in various domains, such as web design [58], UI design [54, 93], diagrams [47], art [93], image manipulation [93], programming [6, 84], and even writing [46], have revealed the need for tools to support exploration of new alternatives. Following empirical insights from these studies, researchers have attempted to support exploratory variants in several ways.

Two researchers, Terry et al. and Lunzer et al. have independently proposed generic techniques for supporting variants to be instantiated in creativity support tools. Both Terry et al.'s ParallelPaths [95] and Lunzer et al.'s subjunctive interfaces [46] allow users to create variants of an artifact, embed the variants together with the main artifact, facilitate easy comparison of multiple variants and allow users to manipulate multiple variants at the same time. However, there are two key differences between the two approaches. First, in ParallelPaths, a user first explores an option and saves the exploration as a variant only when s/he is interested in the outcome of the exploration; in contrast, in subjunctive interfaces, the user first creates a variant before exploration and then deletes the variant if the outcome is not satisfactory. Second, whereas subjunctive interfaces list all variants allowing users to view one variant at a time, ParallelPies (an instantiation of ParallelPaths) groups related variants, allowing users to view combinations of variants. For example, in Figure 2.1, the three segments of the image come from three different variants shown on the right; a user can move the marker to compare and view different mashups of the three variants.

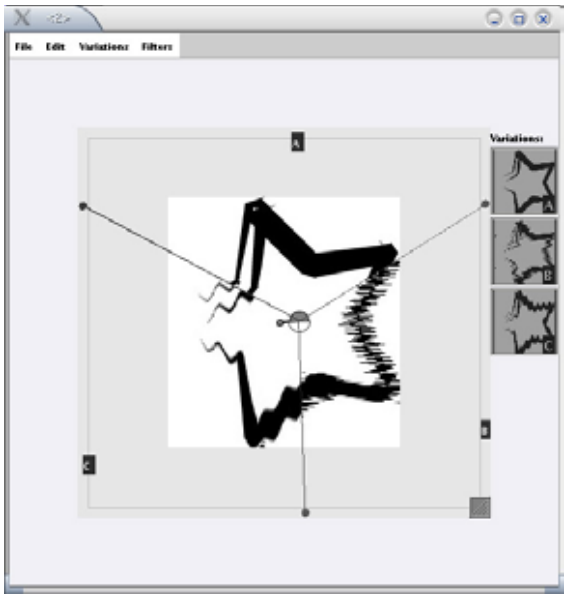

Figure 2.1. ParallelPies interface. ParallelPies groups related variants into a list (shown on the right). A user can create different mashups of these related variants by moving the marker shown at the center of the image. (Figure source: [95]).

Unlike ParallelPaths and subjunctive interfaces that are generic techniques to be instantiated in specific tools, researchers have also built domain-specific tools for exploring alternatives. Similar to the features in ParallelPies and subjunctive interfaces, Hartmann et al. built Juxtapose that allows UI designers to explore and manipulate multiple alternatives at once [22]. Terry et al. built SideViews, a tool that allows graphic designers to preview and compare the effects of applying different commands on an image [94]. Kumar et al.'s Bricolage focuses on exploring alternatives for websites: a web designer can provide as input two web pages and Bricolage creates a new collage by extracting the layout of one webpage to be applied to the other webpage [38].

Despite this long history of work on supporting variations (enumerated above and in the previous subsection), there is a lack of theoretical foundations for how people reason about and work with variants. This lack of foundational understanding limits our ability to "connect the dots" among variations-support tools across domains to develop a general understanding of what aspects in these successful experiments are

actually making the difference. See [83] for a case of how the results of such experiments can misattribute reasons behind a tool experiment's successes.

2.1.3 Choice calculus: a theoretical framework

Fortunately, in the area of supporting variations, researchers have made some progress in building theoretical foundations. *Choice calculus* is a formal language for specifying variations [98]. Based on the idea that each revision or alternative can be encoded by embedding variation points called "choices" within the program itself, choice calculus provides constructs and transformations to reconstruct a program's variant by making a series of selections for each choice. For example, suppose that a programmer creates two variants, each using a different sorting algorithm. In the choice calculus representation of the program, these two sorting algorithms are encoded as a choice in the "sorting" dimension. Choosing one of the two alternatives from that choice will result in one of the two variants.

The choice calculus approach has been applied to represent both revisions and branches, such as for multiple possible compilations of C programs (e.g., IFDEF macros) [45], to build a domain-specific language for querying in a variational information space [14] and even to manipulate multiple variants of images at once [87]. In essence, choice calculus provides a framework (and a programming language) for tool builders to represent and manipulate variants in variations support tools (under the hood).

Like choice calculus, this dissertation brings a theoretical foundation to variations. However, complementary to choice calculus that focuses on the internal representation and manipulation of variants, our theory informs tool builders about how people will reason about, navigate and seek information among variants, and how tools could better support these activities of their users. In particular, our theory can inform tool design in the following ways: 1) guide the design of useful and usable interfaces and interactions (e.g., presentation of variants, design of navigation affordances, supporting various information seeking strategies), 2) predict how a user will use a given variations-support tool in a given situation, 3) evaluate how well a tool supports its user's needs and 4) explain why existing variations-support tools are actually

successful (or not). Towards this end, we draw from Information Foraging Theory (IFT) to explain and predict how people forage for information among variants.

2.2 IFT: Constructs and Propositions

Information Foraging Theory (IFT) has its roots in evolutionary psychology and the biological sciences. Building on Miller's hypothesis that humans are "informavores" that have evolved to work with an abundance of information [50], Pirolli and Card turned to *food* foraging theories to derive a theory of human information seeking, eventually deriving IFT from the optimal food foraging theories [69, 75].

The optimal food foraging theories explains *how* predatory animals in the wild search for their prey [90]. Predators forage for their food in various ecological patches (e.g., grasslands, woods, treetops) by sniffing at various cues (e.g., hoofprints, fur) and following the trail of the strongest scent. By consuming the prey thus obtained, the predator will gain energy; but first, the predator has to expend some energy in hunting down and digesting that prey.

Thus, there is both value and cost associated with foraging a prey. According to the optimal food foraging theory, a predator will engage in foraging behaviors (e.g., deciding which prey to consume, which patch to forage in or which cues to follow) that will yield the maximum profitability or the maximum energy gain in return for the expended energy.

Information Foraging Theory (IFT), drawing on evolutionary psychology, posits that human information seeking has evolved in ways similar to that of their food foraging behaviors [75]. Thus, IFT posits that, just like predatory animals' food foraging behaviors, human information foragers will also optimize in scent-following behaviors so as to maximize the profitability of gaining information. In other words, an information forager will make foraging choices that they expect will maximize the gain in informational value for the cost he/she expends in gaining that information.

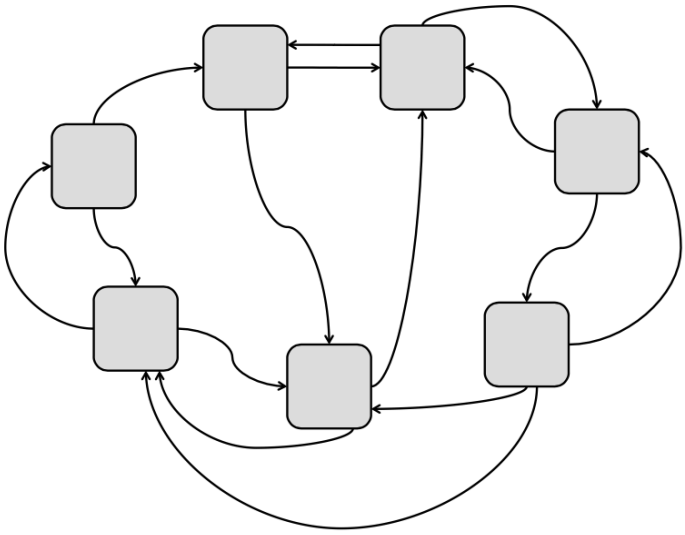

Figure 2.2 Information Topology. Each square represents an information patch. Each directed edge represents a link through which a predator can navigate from one patch to another [75].

More formally, according to IFT, a human predator forages for information prey in an information environment. The information in the environment occurs in a patchy manner, where each information patch is a container of information features (e.g., paragraph contains words, page contains icons). A patch can also be linked to other patches and a predator can traverse an *outgoing* link (e.g., click on a link, scroll) from a patch to navigate to another patch. The patches and the links together form a network called the topology. Associated with outgoing links in a patch are information features called *cues* (e.g., labels on hyperlinks). Cues act as signposts for what might be at the other end of the link. A predator sniffs at these cues and follows the scent trail to eventually locate their prey. Pirolli [75] visually classified these associations visually via Figure 2.2 and Figure 2.3.

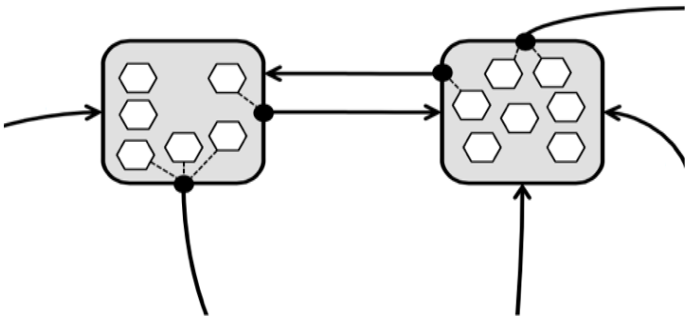

Figure 2.3. Information features and cues. Each patch (square) contains several information features (hexagon). Some of these information features are associated with outgoing links (arrows originating at circles) and serve as cues (circles) [75].

According to IFT, this scent following behavior is an optimization strategy. Since information-rich environments can contain too many patches and links and foraging in each and every patch can be practically impossible, a predator will try to optimize the information seeking by choosing those cues, patches or links that will maximize the information value to be gained for unit expended cost. According to Pirolli [75], this optimization is characterized by equation:

$$Predator's\ ideal\ choice\ =\ Max\ \left(\frac{V}{C}\right),$$

where, for a given foraging choice, V refers to the value of information to be obtained and C refers to the cost (e.g., number of clicks, time taken, cognitive effort required) of obtaining that information. Here, the cost includes both navigation costs as well as the cost of processing and understanding the information.

However, in most real-life foraging situations, the predator does not know the actual costs and values associated with the various patches or links; therefore, he/she often does not make the ideal choice. Instead, the predator makes navigation choices based on the values and costs that they perceive as being associated with a foraging choice. Pirolli [75] characterizes the actual foraging decisions of the predator as:

$$Predator's\ actual\ choice\ =\ Max(\frac{Perceived\ value}{Perceived\ cost}).$$

This imperfect perception of the value-to-cost ratio is called the scent, mathematically written as:

$$Scent = \frac{Perceived\ value}{Perceived\ cost}$$

$$Predator's\ actual\ choice = Max\ (scent).$$

2.3 IFT for Document Collections

In the formative work on IFT, Pirolli and colleagues first applied these constructs and propositions to explain how users of a "scatter-gather" interface foraged for information in a large document collection [69]. This body of work led to the abstract mathematical foundations of the information foraging theory.

To concretely understand people's foraging behavior in an environment, Pirolli and Card adopted the rational analysis technique from cognitive psychology [1]. In rational analysis, researchers assume that an agent (here, human) is rational and has evolved to optimally solve the specific problem under study (here, information foraging) in the environment. With this assumption in place, researchers can then study an agent's behavior in an environment to understand *how* an agent has evolved to solve that problem. Inspired by the success of rational analysis in psychologists' understanding of human memory and learning , Pirolli and Card adopted the technique to information seeking [75].

To perform a rational analysis of human information foraging, Pirolli and Card built computational cognitive models. They extended the ACT-R cognitive architecture [1] with a scent-computation module, thereby building an ACT-Scent architecture [75]. At its core, the ACT-Scent (and ACT-R) architecture predict a person's actions based a set of production rules that operate on a graph-based representation of the person's working memory. The various nodes in the graph represent "chunks" of memory, including the person's foraging goals, the information he/she knows and the what he/she currently perceives in the environment. For scent computation, the ACT-Scent architecture spreads "activation" energy to the different chunks of working memory, such that the activation on each chunk (patch) is a measure of the scent perceived by the forager from that patch.

Pirolli et al. instantiated an ACT-Scent model called ACT-IF to predict how a person will forage in a Scatter/Gather browser for document collections [70]. They also applied the ACT-IF model to foraging in a "Butterfly" document collection browser [71] and in a "hyperbolic" tree browser, to understand where users focused their visual attention [72, 73].

2.4 IFT for the Web

Following the initial success of IFT in the document collections domain, Pirolli and colleagues applied IFT to explain how web users browse the web. Similar to foraging in document collections, Pirolli and Fu employed an ACT-R based cognitive model called SNIF-ACT (Scent-based Navigation and Information Foraging in the ACT architecture) to gain an understanding of the psychology of web browsing, including what links a user will click on and when and why a user will leave a website [74].

Along the same lines, Chi et al. built two other predictive models, namely WUFIS and IUNIS, to establish the link between users' information needs and their foraging behaviors [10]. While WUFIS (Web User Flow by Information Scent) predicted which link a user will take based on scent, IUNIS (Inferring User Needs from Information Scent) predicted, based on the scent a user followed, a set of keywords defining his/her information needs. Both WUFIS and IUNIS were similar to SNIF-ACT in that they employed ACT-like graph representation and spreading activation to compute scent. However, unlike SNIF-ACT, WUFIS and IUNIS were not production-rule models; they worked based on lexical similarity algorithms.

Chi et al. also adopted WUFIS to evaluate the usability of websites and to discover web usability issues [9, 11]. This body of work eventually laid the foundations for modern web usability [89], including the recent work on mobile web usability by Ong et al. [59].

2.5 IFT for Software Engineering

In the domain of software engineering (SE), Ko et al. first speculated that IFT might provide solutions to some of the information-seeking problems in SE [37]. In the following year, Lawrance et al. revealed preliminary evidence that programmers' code

navigations suggested a scent-following behavior [41], subsequently operationalizing IFT for program code in a WUFIS-like computational model called PFIS (Programmer Flow by Information Scent) [42]. Their empirical results showed that programmers' navigations in IDEs were more scent-based than they were hypotheses-based [44]. (Earlier research on program debugging had revealed that programmers' navigations were mostly based on forming and evaluating hypotheses about the code.) Having thus established IFT as a theory for programmer navigations, Lawrance et al. then refined PFIS to PFIS2 and IFT to reactive IFT to account for the constantly evolving nature of programs as well as programmers' information goals even within the scope of a single task [43].

Building on Lawrance et al.'s foundational work, Piorkowski et al. further tested the applicability of IFT to programmers' foraging. They compared the accuracy of IFT-based PFIS2 against other heuristics-based models of programmer navigations and concluded that PFIS2, and hence IFT, was a more accurate predictor of program navigations, than the other heuristics-based models [63]. Encouraged by this result, they built a recommendation tool that presented PFIS2's predictions as recommendations to programmers [64].

Researchers have also explored other aspects of programmers' foraging in IDEs. Piorkowski et al. studied programmers' information diets [66], the effect of production bias [65] on their foraging behaviors and foraging differences between desktop and mobile IDEs [68]. Others, such as Niu et al., have explored IFT as a theory for designing navigation affordances in IDEs [56]. Perez et al. proposed an IFT-based toolkit, namely Pangolin, to aid developers' program comprehension [62].

Beyond its application to code navigation in IDEs, IFT has also informed other aspects of software engineering. Niu et al. applied IFT's optimality models to understand requirements gathering [57], thereby providing IFT-based insights for the design of requirements engineering tools. Kuttal et al. used an IFT's perspective to understand end-user programmers' debugging behavior [39]. Recently, IFT is also finding application in the design of Explainable AI: what information should an intelligent agent provide to its users (and how), so that the latter can understand the former's working and decisions [16, 51]?

Since software engineering is a collaborative activity, researchers have also applied social IFT, a variant of IFT dealing with groups and collaboration [76], to inform collaborative software engineering. For example, Bhowmik et al. applied the concept of "structural holes", a central concept in social IFT, to understand how analyst-stakeholder social linkages affected requirements gathering [4]. They also used social IFT's optimality models to guess the optimal team size for open-source projects [3].

Beyond specific tools and environments, researchers have distilled IFT's design insights into reusable principles and design patterns. Piorkowski et al. abstracted from various SE tools that there are fundamentally only four ways of improving SE tools according to IFT: improving actual costs and values and helping developers actually estimate those costs and values. Fleming et al. went further and distilled the elements of good design in several successful SE tools into a set of reusable SE tool design patterns [16]. Nabi et al. built on Fleming et al.'s work to build a community-based portal for curating IFT-based design patterns [55], to enable tool builders to leverage the theory's insights in a principled manner for building tools.

This thesis builds upon the existing theoretical foundations and computational models of IFT and applies them to the variations domain.

CHAPTER 3. DOES IFT APPLY TO VARIANTS?: FORMATIVE STUDY[1]

The aim of this research is to provide the theoretical foundations, in IFT's framework, for how people seek information among variants. But an elementary question arises in this pursuit: does IFT apply to variations and, if yes, is variations foraging any different from traditional foraging (and, therefore, requires a separate study)?

Therefore, as a first step, we conducted an exploratory study in the programming domain: 1) to frame programmers' information seeking among variants in terms of IFT and 2) to investigate whether programmers' variations foraging is any different from traditional foraging, such as in terms of the cues they attended to, or the foraging strategies they employed.

We chose the programming domain because: 1) IFT has been applied to programming in the past [41, 67, 79] and 2) variants and exploration are common in programming [6, 99].

3.1 Study Methodology

Our target population is people who engage in exploratory programming. A prior study on the subject found that novice (as well as expert) programmers opportunistically reuse code from various sources, including prior variants of a program [6]. We used the results from this prior study to guide our study design, such as for recruiting participants and in task design.

3.1.1. Participants

We recruited 8 novice programmers, namely undergraduate students in CS, and investigated their variations foraging behaviors during a reuse task. Table 3.1 summarizes the general demographics of our participants. Most of them had some

---

[1] Srinivasa Ragavan, S., Kuttal, S. K., Hill, C., Sarma, A., Piorkowski, D., & Burnett, M. (2016, May). Foraging among an overabundance of similar variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*(pp. 3509-3521). ACM.

experience with programming, but were relatively new to JavaScript, the language on which our study was based. The only exception was P2 who carried 6 years of JavaScript experience; however, he noted in his response that he had only occasionally programmed in JavaScript.

Participant label	Gender	Level	Age	Experience(years)		Prior web development experience?
				JavaScript	Programming	
P01	Male	Sophomore	20s	1	1.5	Yes
P02	Male	Freshman	Teens	6*	6	Yes
P03	Female	Junior	20s	0	9	No
P04	Male	Sophomore	20s	2	1	Yes
P05	Male	Freshman	Teens	0	3	No
P06	Male	Sophomore	Teens	0	5	Yes
P07	Male	Junior	20s	2	2	Yes
P08	Male	Junior	20s	1	5	Yes

Table 3.1. Study-1 participant demographics.

(*Participant P02 reported that he only occasionally programmed in JavaScript)

3.1.2 Tasks

Participants were presented with the following scenario. A small non-profit named Nourish Line hosted a JavaScript game called Hextris (similar to Tetris) [30] on their website. Since the company had very few full-time employees, several volunteer programmers had worked on Hextris over the years. Recently, visitors to the site had suggested some changes to the game and we asked participants to implement those changes for Nourish Line.

Specifically, we asked participants to make the following three changes to the latest version of the game (shown in Figure 3.1 (a)):

1) move the game's score indicator from the center of the hexagon to a location above the hexagon "like it was before",
2) move the bonus score multiplier to a location above the hexagon "like it was before" and

3) change the text color of the score and multiplier to black so it could be seen when placed above the hexagon.

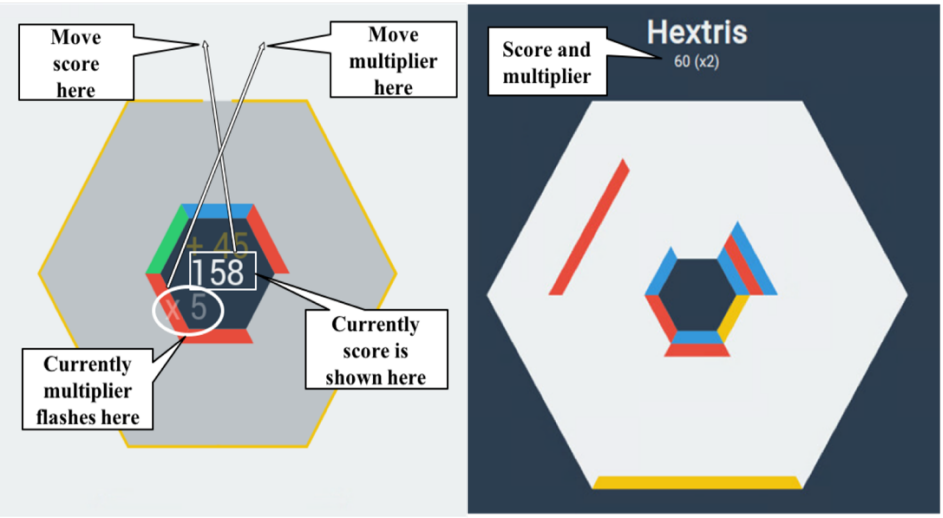

Figure 3.1. Hextris Game. Participants were asked to change the latest variant of the program (a), to move the score and multiplier above the hexagon "like it was before" (b). The "before" variant was not shown to participants.

Figure 3.1(b) shows how the game looked earlier: notice that the multiplier was displayed within parentheses next to the score. However, we did not directly provide this earlier version to participants. Instead, we used the phrase "like it was before" to suggest that useful portions of a solution might be available for reuse in earlier version(s) of the game. As mentioned earlier, the choice of our task, namely reuse from prior variants, is realistic and follows Brandt et al.'s observations in exploratory programming [6].

3.1.3 Presentation of Variants

To perform their reuse task, participants worked in Cloud9, a web-based IDE for Javascript development; Figure 3.2 shows the Cloud9 environment. As the figure shows, we provided participants with 700+ variants of the Hextris game.

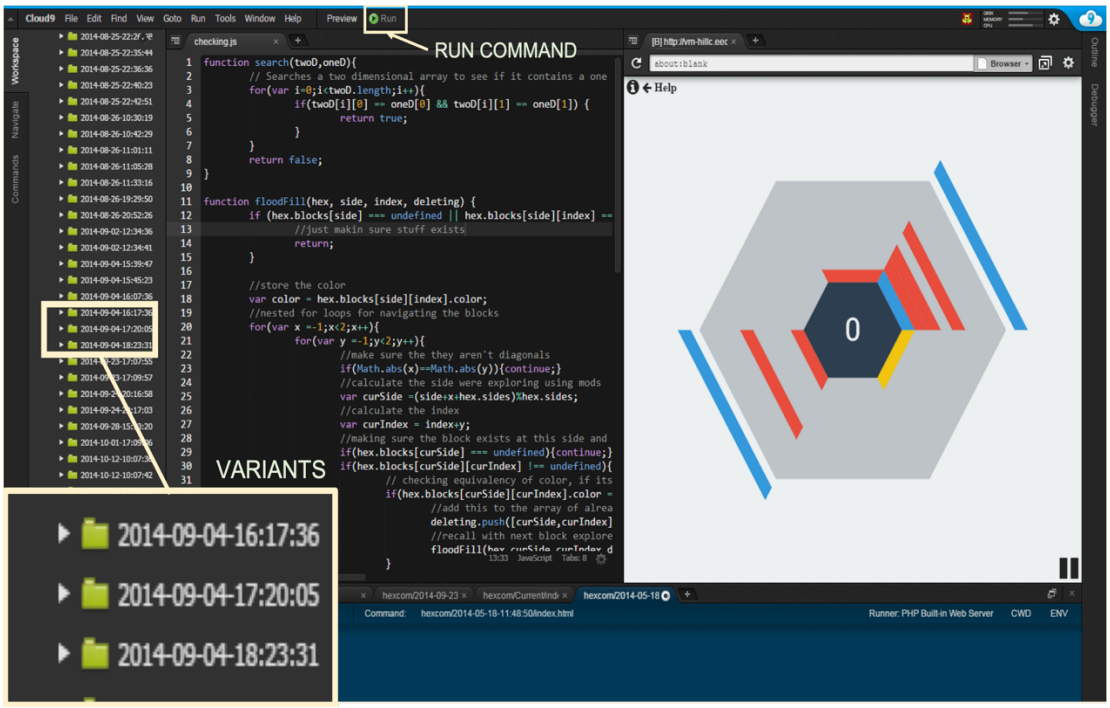

Figure 3.2. The Cloud9 environment. Variants were presented in chronological order and labeled with commit timestamps.

We obtained the variants from the game's public GitHub repository [31]. For each commit of the 700+ commits in the repository, we created a variant as follows: we extracted the entire repository tree, or the copy of the program at that version, into a folder; thus, each of these folders was a variant. In addition to the program code, each variant folder also contained a changelog.txt file that contained the corresponding commit's metadata (viz., commit timestamp, author name, commit message, commit ID).

These variants were then presented to participants in the Cloud9 environment. As Figure 3.2 shows, each commit was labeled with the commit timestamp. Only the latest variant was labeled "Current". (Recall that participants had to make changes to this latest variant). The variants were presented in a chronological order.

Since each program was an entire copy of the program, participants were able to run the entire program for each variant. For this, programmers had to run the index.html file within each variant, either by right clicking and choosing the run command, or by clicking on the run icon on the command bar (Figure 3.2).

3.1.4 Study procedure

At the beginning of each study session, participants signed an informed consent form and filled in a general demographic questionnaire. (Table 3.1 summarizes the demographics). We then provided participants a short tutorial on think-aloud protocol, followed by a brief introduction to the study environment. For participants with no prior Javascript experience, we also provided a short tutorial on the basics of HTML, CSS and JavaScript.

After the initial tutorials and task description, we introduced participants to the Nourish Line scenario and the tasks mentioned earlier (Section 3.1. 2). Each then participant spent 50 minutes working on their programming tasks while also thinking aloud. We collected the audio and video of the participant, captured their screen as well as logged their IDE actions. Two researchers observed the participant from another screen, annotating actions for in-depth data collection.

Following the programming session and a short break, we conducted a retrospective interview. We played the screen-capture video for the participant and drilled down into the annotated actions. We stopped the video at the annotations and asked the participant additional questions about their foraging. The interview questions, listed in Figure 3.3, were inspired by IFT research on debugging [66]. We also captured the audio, video and the screen for the retrospective interviews.

At the end of the study session, each participant was compensated with $20.

Explain:
    You chose to [do/go to] (Variant / location)
Ask:
    What did you expect to (see/find) when you went to _____?
    What did you see as your other possible choices?
    Why did you choose to navigate to _____ as opposed to (other choices)?

Figure 3.3. Interview questions. During the retrospective interview, we played back the video of the participant's programming task and asked these questions about their foraging decisions.

3.2 Qualitative analysis

For our analysis, we used qualitative methods. First, we transcribed participants' think-aloud verbalizations and on-screen actions and then segmented the transcripts into 30-second segments. We then coded the segments, allowing multiple codes per segment.

To ensure rigor, two researchers independently coded 20% of the transcripts, until we reached at least 80% agreement on the 20% data. In our study, we obtained an inter-rater agreement of 85% on 20% of the data. Once we reached agreement, the two researchers split up the coding for the remaining data. We used Jaccard coefficient as the measure of inter-rater agreement.

We coded the data according to the cue types participants used, the type of operations they performed, and their navigation behavior. We drew the base code set from previous IFT research [40, 66] and added new codes to capture the new phenomena that we observed in variations foraging. Table 3.2 lists the entire code set; the shaded rows indicated newly added codes.

CODE	DESCRIPTION
CUE TYPES	
Create Time, Update Time	Timestamp cues marking latest, first or intermediate variants, and navigation to corresponding variants.
Previous File, Previous Method	Reuse of information features (file and method names) from one variant as cues in another variant.
Output	Cues based on how output looks or running a preview
Domain	Game-related words, e.g., "score", "block"
Source	Source code-based cues, e.g., function / variable name.
Error, Correct	Cues based on error/correctness of patch/prey
File Name, File Type	Filename-based and file type cues
Document	Documentation cues: change logs, readme files, tooltips, etc.
Comment	Source code comments
Search	Search inside IDE or the internet
Debug, Inspect	Debugger or "element inspect" feature in browser

OPERATIONS	
Edit	Edits made to source code, to verify the prey using output-based cues, or to implement the task.
Reuse	Explicit reuse of source code , i.e., copy and paste
Compare	Compare two variants
NAVIGATIONS	
Between Variant Navigation	Between-variant navigation was coded along with the cues that guided these navigations.

Table 3.2. Analysis codeset . We derived our codeset based on prior work [40, 66]. The new codes we added are highlighted.

3.3 Results: Foraging activities

We frame our results on participants' variations foraging behaviors around a modified version of Rosson and Carroll's reuse model [82]. According to Rosson and Carroll, programmers reuse code from a "usage context" to accomplish a task in their "current context" in three stages:

1. finding a usage context,
2. evaluating the usage context and
3. debugging the usage context.

This model assumes that: 1) the current context is known to the programmer and 2) that the current and the usage contexts are within the same variant of the program.

However, in our study, the current and the usage contexts were in different variants and participants had to find both these contexts as part of their task. Therefore, we extended Rosson and Carroll's model to comprise the following three stages:

1. finding and evaluating a current context,
2. finding and reusing a usage context, and
3. integrating the current and usage contexts.

Note that this modified reuse model, visually illustrated in Figure 3.4, is essentially a more generic version of the original Rosson and Carroll's model.

From an IFT's standpoint, the term context, in both the original and the modified reuse models, refers to information patches: a programmer reuses code from one or

more "source" patches (usage context) in one or more "destination" patches (current context).

Also, in our study, the specific source and destination patches—namely, the exact locations where the relevant source and destination code is located—is in turn present inside of variants[2]. Therefore, as Figure 3.4 shows, in order to forage for a usage/current context, participants had to find and evaluate the appropriate source/destination variant, and then forage for the smaller source/destination patches (i.e., method or lines) within that variant.

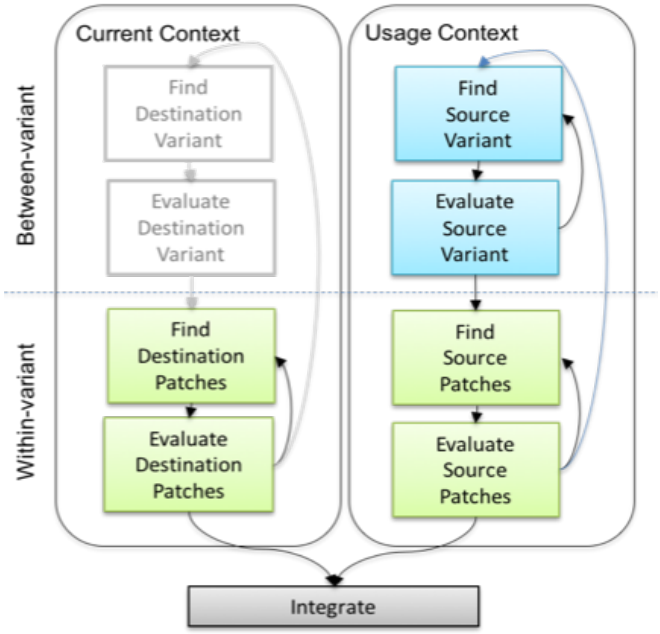

Figure 3.4. Modified reuse model. Participants were provided with the destination variant (greyed out). They interleaved finding and evaluating the prey in both between-variant (blue) and within-variant (green) foraging.

Note that we do not intend, with our modified reuse model, to suggest that programmers followed any particular order in foraging for their source and destination contexts. In fact, Figure 3.5 shows that Participant P06 foraged for the source variant

---

[2] Technically, in IFT, variants are also information "patches": however, for the purposes of disambiguation, we use the term variant to refer to the variant and the term patch to refer to smaller patches (e.g., methods, files) within the variant.

and the source patches (usage context) before he foraged for the destination patches (current context). Thus our results describe *sets*—not sequences—of behaviors. In the rest of this chapter, we will describe the foraging behaviors of participants in each of the reuse stages.

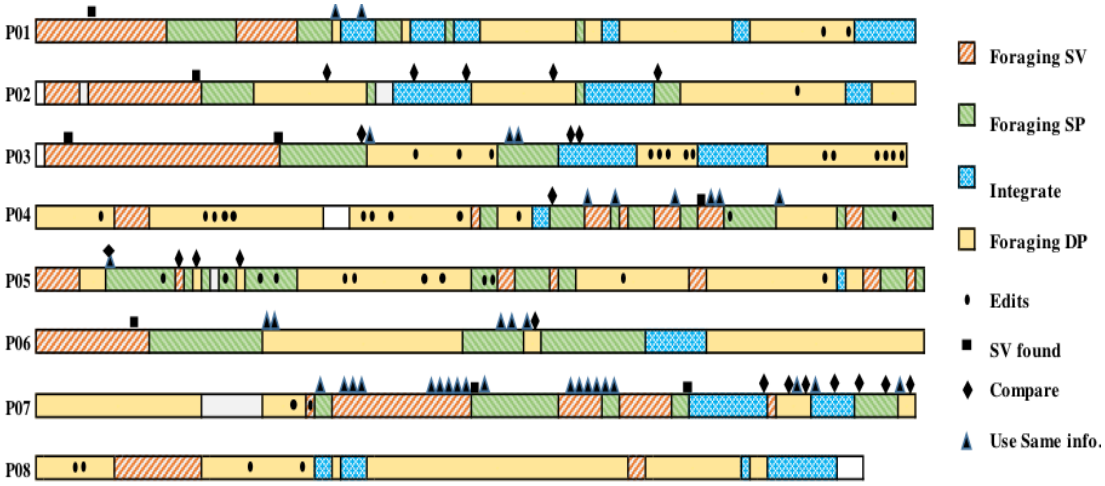

Figure 3.5. Foraging timeline. Participants foraged the Source Variant (SV), Source Patches (SP) and Destination Patches (DP) in different orders. They used information features (shown as triangles) from one variant to forage in other variants and performed comparison during both source-patch foraging and destination-patch foraging.

3.4 Results: Stage 1. Finding and evaluating the current (destination) context.

While foraging for the destination context, namely where the fix needed to be made, participants' foraging behaviors were unsurprising.

1. *Destination variant: find & evaluate.* Since participants in our study were asked to make changes to the latest variant labeled "Current", the source variant was already provided to them. Thus, participants did not have to find and evaluate the destination variant in our study. (These activities are therefore greyed out in Figure 3.4).
2. *Destination patches: find.* In order to find the task-relevant "destination" patches within the *Current* variant, participants engaged in reading code, searching and navigating links. These behaviors are well-described in prior IFT literature on debugging and maintenance in a single program variant [42, 66].

3. *Destination patches: evaluate*. Finally, once participants found the destination patches, they wanted to ensure whether the code they had found actually did what they thought it did. For this, participants edited the code and confirmed whether the output matched their expectations. For example, as the dots in yellow segments show in Figure 3.5, all participants altered the *x* and *y* parameters in the *renderText*() method in order to evaluate whether it really altered the score position. These behaviors conform with prior findings on IFT and hypotheses testing in debugging [44].

3.5 Results: Stage 2. Finding and evaluating usage (source) context.

Once participants foraged for their destination context, they then moved on to forage for their source context. However, unlike the destination context foraging that was largely unsurprising, participants' foraging for the source context revealed new nuances for variations foraging.

3.5.1 Source variant: Find

Participants began their source context foraging by looking for a suitable source variant—one that contained the source and multiplier above the hexagon as in Figure 3.1(b)—among the several available variants.

In IFT, when a predator forages to find a suitable patch among several available patches, it is called between-patch foraging [69]; analogously, when participants had to find a variant among several available variants, we call it *between-variant foraging*. However, unlike in traditional between-patch foraging where the patches (and hence the cues and scent) are mostly unique, the variants in between-variant foraging (and hence the cues and the scent) can all be very, very similar.

As part of their between-variant foraging, participants had to decide which variant to navigate to next; for this, our study environment provided only one kind of cues, namely the timestamps that were specified as the variants' folder names. Participants leveraged these timestamp cues to in the following ways.

1. Sometimes they used the timestamps to directly navigate to a variant they had already seen. For example, P01 said, *"So I'll just remember that on 2014-05-20, [it] had the right interface"* and later navigated directly to that variant.

2. Other times, participants navigated to a variant based on its position in the chronological list of variants; for example, 5 out of 8 participants scrolled to the top and navigated to the first variant to see how the game looked at the beginning.
3. Participants also used the distance between chronological variants as a measure of how similar (or different) variants might be. Since consecutive variants were very similar, participants often skipped ahead, using the used the distance between variants to guess how many variants to skip (or, equivalently, how far to scroll). This way, participants were able to avoid potentially irrelevant variants and to quickly land in a potentially valuable variant. As Figure 3.6 shows, such skip-ahead between-variant navigations followed one of three patterns:

1. *Unidirectional:* Four out of eight participants (P01, P02, P03, P04) foraged in a single direction. They either foraged from the oldest to the most recent variant or vice-versa (see Figure 9 (a)). P03 explained: "*... jumping down more ... like a sorting algorithm, checking further and further until there was a change.*"
2. *Bidirectional:* Four out of eight participants (P03, P06, P07, P08) changed directions while foraging between candidate source variants. Initially, they started from either the oldest or the most recent variant and foraged along one direction. However, when they found that they had gone too far in one direction, they reversed course and continued in the other direction(Figure 9 (b)).
3. *Systematic narrowing:* Two of the eight participants (P02, P05) started from a variant in the middle and systematically narrowed down the search space using an approach similar to a binary search. Participant P02 said: "*... just split the list in half and then ... do a binary search on it.*"

Figure 3.6. Between-variant navigation patterns. (a) Participant P01 was a unidirectional forager. (b) Participant P06 was a bidirectional forager. (c) Participant P05 was a systematic narrowing forager. Note that they all skipped several variants at a time, as they navigated.

Finally, in addition to the timestamp cues that were available in the environment, participants also left cues for themselves during their foraging. As participants navigated across variants, they left potentially valuable variant folders expanded to be able to revisit them later (or, equivalently, they closed off low-value variant folders). Later, when they had to navigate to a previously-visited variant, they directly navigated to the expanded folder(s), instead of searching for the it from scratch.

3.5.2 Source variant: Evaluate

Once participants navigated to a variant, they evaluated whether it was appropriate for reuse. Specifically, they evaluated whether the variant contained the score and multiplier above the hexagon. For this, participants used different types of cues that were present *within* the variant. (In contrast, the timestamp cues were present as the label for a variant).

Participant	Cue Types			
	Output-based	Changelog-based	Source code-based	Filename-based
P01	29	12		
P02	15	8		
P03	44		3	
P04	6	15	2	
P05	9		2	
P06	22		1	
P07	12		22	3
P08	14		2	
Total occurrences	151 (68.3%)	35 (15.8%)	32 (14.5%)	3 (1.4%)

Table 3.3. Source variant evaluation: cue types. Participants used some cue types more frequently than others to evaluate variants.

1. Output-based: When foraging in output patches, participants attended to information features, such as the position of score or whether the output runs or not. These information features functioned as cues (signposts to—or away from—the prey) because, depending on what they saw, participants decided whether to navigate into the variant's source code, or to navigate away from that variant. These output-based cues were the most popular of all cue types; as Table 3.3 shows, 68.3% of all cue-type codes in our coded transcripts were related to output.

We attribute the popularity of this cue type to its low cost. First, the cost of bringing up these patches was low: participants had to right click on the index.html file and choose “Run” or selecting the index.html file and click on the “Run” button in the command bar. Second, the cost of processing these patches appeared to be low: just a quick visual inspection of the output revealed whether or not the score and multiple appeared above the hexagon, or whether the game was broken.

2. Changelog-based: Another kind of low-cost cues were the words in the variants' changelogs, present in the changes.txt file within each variant. Participants frequently used these cue types to identify whether the variant contained the required information features; for example,

P1: "I expected to see something along the lines of 'changing the position of score'" .

However, change logs were often unhelpful because: 1) they were non-descriptive (e.g. one log contained only the text "asdf") and 2) they contained information only about what had *changed*—and not what was present—in a variant. As a result, participants abandoned changelogs and fell back to other cue types;

P3: "their document isn't that good for people changing things."

3. Source code-based: Most participants (5 out of 7) also evaluated a variant based on the source code in the variant. Sometimes, participants read the code in variants to understand what the code did. At other times, participants used the similarities or differences across variants to judge what a variant contained.

P07: "this still looks like the center to me" (emphasis added).

Some participants also perceived errors in source code (e.g., squiggly red lines indicating compilation errors) as a negative scent and steered away from that variant to other variants.

4. Filename-based: Finally, one participant (P07) also used file names as cues for evaluating variants: if a variant did not contain certain file names, he immediately rejected the variant.

P07: "... at some point, [filename] did not even exist"

Based on the different types of cues described above, participants made one of the following three foraging decisions:

1. they concluded that the variant was inappropriate for reuse (e.g., the game was broken, did not contain score) and went on to find another variant, or
2. they found that the variant contained the score above the hexagon, but continued to find another variant that contains easier-to-integrate code, or

3. they concluded that the variant was appropriate for reuse, proceeding to find and reuse code from that source variant.

3.5.3 Source patch within source variant: Find

Once participants found a source variant, they then foraged for the task-relevant (source) patches within that variant. Thus, participants looked for the code displayed the score and the multiplier. Although this foraging goal, namely finding the score and multiplier-related code, is similar to that in destination-patch foraging (Section 3.5.1), participants adopted very different strategies in both these activities.

Earlier, when participants foraged within the destination variant, they did not know where the task-relevant patches might be. They had to rely solely on cues in the environment to hunt down the destination patches. In contrast, while foraging in the source variant which was *similar* to destination variant, participants had already formed expectations about where the source patches might be located. These expectations provided a starting point for participants' foraging within the source variant.

More specifically, because the variants were similar to each other, participants expected that the source and the destination patches will contain (at least some) common information features, and that the information features found in the destination patches in the destination variant will also lead them to the source patches in the source variant. For example, P03 found calls to renderText (in the view.js file) in the destination patches within the destination variant. Later, when she foraged in the source variant, she searched for exactly the same features, hoping that the task-relevant patches in the source variant also will contain the same information features (and will hence lead her to the source patches): "*renderText is still something I can look for*". As the triangles above the rows in Figure 3.5, of the seven participants who foraged for source patches (green segments), six participants used such similarity-based strategies for their foraging.

However, participants did not always succeed with such a similarity-based strategy. Sometimes, the program had changed over time between the source and the destination variant. As a result, participants did not find the same patches or information features that they had found earlier in the destination variant. In such cases, participants proceeded in one of the following two ways.

Sometimes, participants continued to forage based on similarities, hoping to find other information features that might still be similar among the variants and therefore lead them to the source patches. For example, when Participant P03 did not find the *renderText()* method in the source variant, she said:

> *"what are some other key phrases I can look for... I guess go back and check for score."*

In Figure 3.5, row P03, the two consecutive triangles denote how she looked for *renderText* and then immediately looked for *score*.

Other times, participants abandoned the similarity-based approach and began foraging within the source variant, just like they had foraged in their destination variant (e.g., by reading source code words and following scent). For example, when P03 tried and failed to forage for her prey using similarities for two consecutive times (as indicated by two consecutive triangles in Figure 3.5, row P03), she started reading the source code within that variant, as the subsequent absence of the triangles in the figure shows.

3.5.4 Source patch within a source variant: Evaluate

After finding their source patches, participants also evaluated them, based on the following two criteria.

First, participants evaluated whether the code did what they thought it did (e.g., if a line of code actually altered the score position). For this, they edited the code and saw whether the resulting output met their expectations, just like they did when they evaluated the destination patches. Two participants (P04, P05) performed this kind of evaluation as Figure 3.5 shows: in rows P04 and P05, the dots in the green bars represent edits made during source patch foraging. If participants concluded that they were not on the appropriate source patches, they undid their changes and continued foraging for the correct ones within the same variant.

The second kind of evaluation relates to integrating the source and destination patches: how easy can these source patches be integrated with the destination patches? If the source and destination patches were similar, participants expected that the code integration would be easy; otherwise, if the source and destination patches were

different, the integration might be hard and so participants went on to forage for another variant where the code might be easier to integrate. All seven participants who foraged for source patches within a source variant engaged in this evaluation activity. (The exception, P08, could not successfully find the source patches and hence did not evaluate them. Instead, he re-implemented, instead of reused, the code to complete the task.)

Although in finding and evaluating the source context, participants mostly used a drill-down approach—they first found and evaluate the source variant using non-code cues and then drilled down to find and evaluate the source code patches—two participants (P04 and P07) foraged for the source patches as part of finding and evaluating the source variant. These participants relied on cues in source-code patches, comparing them with destination patches, for evaluating their source variant. Thus, they interleaved their source-patch foraging with their destination patch foraging. The triangles above the orange segments in Figure 3.5: rows P04 and P07 indicate these instances.

3.5.5. A Foraging Strategy: Story-guided foraging

Crosscutting the foraging activities described above, participants engaged in what we call *story-guided foraging*. As participants went about foraging between and within variants, they built stories of how the game had evolved. Specifically, based on the information features they had collected from the various variants (and patches) they had foraged in, participants built two kinds of stories.

Some participants, such as P08, built stories of how the game evolved: "*seems that the game had the zero in the middle and then before that never worked. That could've gotten broken at some point though*" (P08). Other participants, such as P07, built stories about how the code –instead of just the game– evolved: *"At some point, this [method] was [re]factored into its own function, then it was [re]factored back out of its own function."*

Indeed, participants used both the kinds of stories to guide their foraging. However, neither of these kinds of stories were necessarily complete, or even correct. Participants started off by creating an incomplete outline of the story and then refined them as they visited more variants and gathered more information features. As an illustration of how

these stories were built and refined, let us look at the verbalizations of P07 in the retrospective interviews. Initially, he built the following story, based on information features in a few variants:

> *"...early in development there's no score label. At some point the original score label is introduced. And then, after that, the 2nd score label's introduced".*

As he processed more information features from the output of more variants, he refined his story:

> *"... after dealing with this for a while, there might have been like no score label, then the original score label, then no score label, and then the second score label for a while".*

He then used his story to guide his foraging:

> *"... that's basically why, when I hit this version that had no score label, I just decided to start searching in a more recent direction".*

3.6 Results: Stage 3. Integrating the variants

Once participants found and evaluated the current and usage context, they proceeded to the third stage of reuse, namely *integrating the variants,* to complete their task. They integrated the code in one of two ways.

1. *Copy and paste:* When two variants were similar, participants attempted to copy and paste the code from the source variant into the destination variant, and made minimal modifications to match the task requirement. Only one participant (P06) was able to find such a similar patch for reuse, and only for one of the tasks. In other cases, when the two variants were dissimilar, participants copied and pasted code from the source patches into destination patches, and then fixed all the dependencies and errors. Three out of eight participants (P01, P07, P06) followed this strategy.
2. *Re-implement:* Two participants (P07, P03) implemented the task from scratch (without reuse) when they found source and destination variants to be dissimilar. Further, one participant (P08) could not locate the right source

variant; therefore, he directly implemented the fix only based on the (textual) task descriptions.

The two integration strategies described above correspond to "cut-and-stanch-the-bleeding" and "analyze-then-act" strategies described in the reuse literature [26]. Participants chose their reuse strategy depending on whichever they perceived to be low cost.

3.7 Discussion

3.7.1 Threats to validity

As with every study, the results presented so far have to be interpreted keeping a few threats to validity in mind. First, in our study, we presented each variant as a folder containing the entire copy of the program. This "vanilla setup" has the disadvantage of not considering state-of-the-art presentation devices and navigation affordances for variants (e.g., in version control tools). However, this choice of presentation allows us to build the theory from scratch, eliminating the effects due to affordances in existing variations-support tools (e.g., version control tools).

Second, our participants were not gender-balanced. Given that prior studies (e.g., [2]) have revealed that problem solving strategies cluster by gender, it is possible that some foraging behaviors relating to women's problem-solving strategies were not revealed in our study.

Third, although we chose a real program from GitHub, and designed our tasks based on prior studies, the program and tasks used in our study might not be representative of all tasks involving variants in the real world. Similarly, our results might also not generalize to other programmer populations who might engage in variations foraging with other motivations than reuse, or adopt different foraging strategies (e.g., programmers with prior knowledge of the codebase, expert programmers). Such limitations in generalizability can only be addressed through further empirical studies.

3.7.2 Generalization of our results

Fortunately for us, Martos et al. built upon our work and conducted another study investigating cues and strategies in variations foraging [48]. Their study also involved

reuse from variants. However, they conducted their study with end-user programmers. They also used two different programming environments, namely AppInventor and MATLAB, deriving their variants from crowdsourced online repositories, namely AppInventor Gallery and MATLAB File Exchange respectively. These differences in population and study environment provide some external generalization for our findings

Indeed, several of our results also generalized to Martos et al.'s study. Specifically, participants in both studies used similar cues and strategies, such as timestamps and comparison-based strategies, thus lending some external validity to our findings.

However, their participants also adopted new cues and strategies (e.g., program descriptions, crowd ratings) based on the affordances and cues available in their study environments (and not available in ours). These new cues and foraging behaviors expand our results on variations foraging to online, crowdsourced environments.

3.7.2 Open questions

Our study also raises new questions in information foraging, from the perspectives of information consumers as well as information producers.

Starting with the consumer (forager) side, traditionally, IFT has dealt with information environments with largely dissimilar patches (e.g., different web pages in a website, different methods within a program). However, in the variations domain, the information environment consists of very similar and even identical patches across variants; consequently, the information features and hence the cues and scent might also be similar across variants, making it difficult for foragers to follow scent and to forage with the strategies described in prior studies in IFT (e.g., [44, 66]).

Indeed, participants in our study demonstrated two new foraging strategies. First, they heavily relied on the comparison operation: 1) to find what was different among very similar variants, 2) to find a variant similar to a given variant in a certain way, 3) to find patches with information features identical to those seen in other variants. Second, they foraged across variants by generating temporal stories about how the program evolved over time and then using those stories to guide their foraging (5 out

of 8 participants). These foraging behaviors may be uniquely important to variations foraging.

These new behaviors call for further research into variations foraging. First, there is no construct in IFT that can be instantiated as a story--stories are not cues, not patches, not prey, and so on. Thus, an open research question is whether new IFT construct(s) are needed to capture this phenomenon. Second, comparison is important in variations foraging, but current IFT computational models do not account for an explicit comparison operation: they mostly consider within-patch foraging, between-patch foraging and enrichment as operations that human foragers perform. This calls for enhancements to IFT computational models.

Whereas the above two open problems concern how foragers (consumers) forage among available variants, two additional questions arise on the producer side: 1) *what makes a good variant?* 2) what makes a good cue (e.g., changelogs, descriptions) for variants? These producer-side questions are important because the patches and cues created by the producers are eventually where and how consumers will forage later.

Currently, the ways programmers create variants (including in our Hextris program) are rather arbitrary and depend on the individual programmer: if a producer chooses not to save a variant (e.g., broken code), the consumer might not find the prey at a later time [80]. On the other hand, if the producer created too many variants, consumers might find it harder to forage for information. This relationship between the producers' information creation and the consumers' information foraging calls for research into the producer side of information foraging.

Here, two avenues are particularly ripe for research. First, IFT researchers need to begin looking at foraging from the perspective of the producer: how do producers of variants think about the scent they are leaving for the variants?, what signposts do they think they are leaving for the future consumers?, how do they think of the different potentially-conflicting needs (e.g., every change is a new variant vs. multiple related changes makes a variant) of future consumers?, do they think about future consumers' navigations between patches, or only within patches, or not at all? Second, as tool

builders, we need to consider how well variations-support tools support producers' creation of patches and cues so that they meet the foraging needs of future foragers.

3.8 Conclusion: Does IFT apply to variants?

Finally, revisiting the fundamental question we set out to answer in this chapter, *does IFT apply to variants?* Our results suggest *yes*. First, we were able to frame our explanations of programmers' foraging behaviors in IFT's constructs of patches, prey, cues and scent. Second, the foraging behaviors we observed are explained and predicted by IFT's cost-value proposition. For example, participants adopted tactics such as looking for identical information features in within-variant foraging, or skipping variants during between-variant foraging or looking for cues that highlighted differences, so as to minimize the costs of their foraging.

However, these are only preliminary evidences and are not conclusive, or even sufficient. First, there is the story-guided foraging strategy that participants adopted: since we do not know what IFT constructs these stories should be instantiated as, we need to investigate how important these stories are for explaining and predicting participants' variations foraging. Second, our evidence favoring IFT as a theoretical framework for variants is based on our interpretation of participants' foraging behavior; to build solid theoretical foundations, we need to validate whether these interpretations are correct.

Therefore, in the next two chapters, we will concretely operationalize IFT for variants in computational models and then evaluate how well our interpretations of variations foraging, encoded in those models, can predict variations foraging.

CHAPTER 4: PFIS-V: MODELING PROGRAMMERS' VARIATIONS FORAGING IN SOURCE CODE[3]

In the previous chapter, the qualitative results from our formative study suggested that IFT applies to variations. In particular, we were able to describe many of participants' variations foraging behaviors using IFT's constructs and propositions (e.g., cues, patches, costs, values, scent). However, qualitative results mostly show the existence of phenomena and not their generality. To fill this gap, we built computational models that concretely operationalize IFT's constructs and propositions for an environment containing variants, and predict how, according to IFT, a programmer will forage in that environment.

IFT computational models are tools for researchers to test their hypotheses about how people will forage in a given situation. Researchers can operationalize IFT—the patches, the cues, the links—for an environment in an IFT computational model. They can then encode in the model their hypotheses about how a person will forage in that environment. If the model can accurately predict a person's navigation in the environment, it adds to evidence of the validity of the researchers' hypotheses. Otherwise, if the model fails to accurately predict the navigations, the researchers need to refine their hypotheses (and hence understanding) about people's foraging.

In our research, we framed our hypotheses about variations foraging based on our qualitative observations described in the previous chapter. We encoded these hypotheses in two computational models, namely PFIS-V and PFIS-H, and then evaluated the models, as a way of evaluating our hypotheses and to solidify our understanding of variations foraging. In this chapter, we will present PFIS-V and its

---

[3] Ragavan, S. S., Pandya, B., Piorkowski, D., Hill, C., Kuttal, S. K., Sarma, A., & Burnett, M. (2017, May). PFIS-V: modeling foraging behavior in the presence of variants. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (pp. 6232-6244). ACM.

empirical evaluations. In the next chapter, we will discuss our second model, namely PFIS-H, that addresses key limitations in PFIS-V.

4.1 The PFIS-V Computational Model

PFIS-V, short for PFIS for Variants, belongs to the PFIS family of IFT computational models. In turn, PFIS stands Programmer Flow by Information Scent. As the name suggests, the PFIS models predict a programmer's navigations based on the scent s/he will perceive in the programming IDE. Whereas earlier PFIS models predicted how a programmer will navigate within a single variant of a program, PFIS-V predicts how a programmer will navigate among multiple program variants.

PFIS-V builds upon PFIS3, the latest PFIS model for single variant situations [63]. It makes two key extensions to PFIS3 to account for variations foraging: 1) it accounts for multiple variants in the information environment and 2) it accounts for how the programmer will reason about, and navigate among, those variants. These extensions are encoded in the model's data model and predictive algorithm respectively.

4.1.1 PFIS-V Data model: Accounting for programmers' mental model of variants

The data model in PFIS-V (or any other IFT computational model) is a representation of the information environment--the patches, links and cues--that the programmer has seen (and thus knows about) so far. It represents the information environment as a graph $G = (V_P \cup V_W, E_L \cup E_W)$, where

- V_P = set of all patches (methods) the programmer knows about so far,
- V_W = set of words in the patches (excludes programming language reserved words such as "return" and common English language words such as "the"),
- E_W = set of "patch contains word (cue)" relationships, and
- E_L = set of links, or one-click navigation affordances, between patches (e.g., adjacency, method invocation links).

As an illustration of the PFIS-V data model, consider the program snippet in Figure 4.1 (a): the corresponding data model graph is given in Figure 4.1 (b). Notice that for each patch (e.g., sum method) in the program snippet, there is a patch node (blue

ellipse) in the graph. Similarly, each word in the program corresponds to a word node (red rectangle) in the graph.

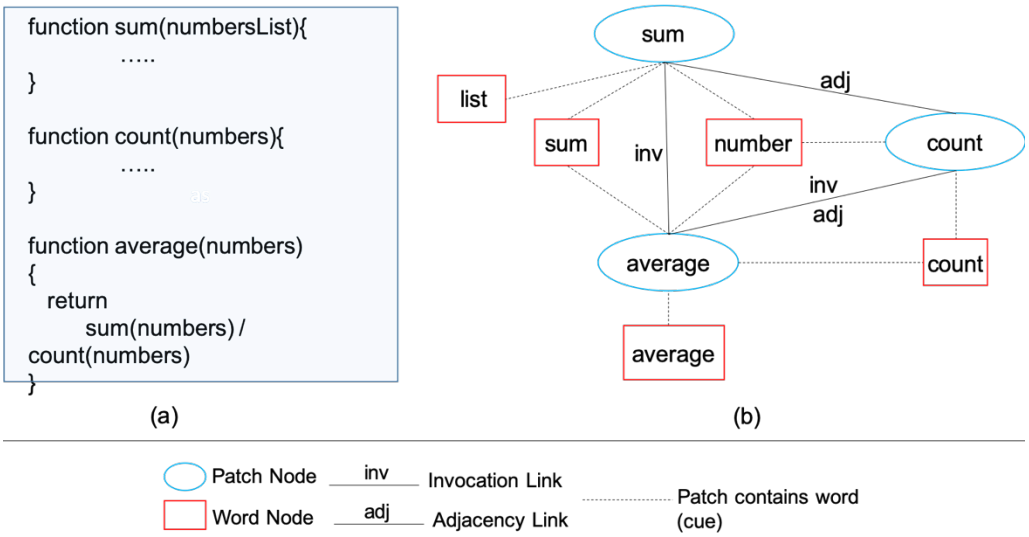

Figure 4.1. PFIS3 data model (a=single variant program snippet, b=corresponding PFIS3 data model). Each patch and word in (a) are represented as nodes in (b). The links between patches and "patch contain word" relationships in (a) are represented as edges in (b).

Now consider the method *average* (a patch) in the program snippet. It contains words like average (its name), numbers (parameter), sum and count (contents and calls to other methods). Thus, the data model contains "patch contains word" links (---) between the *average* patch node and the *average*, *number*, sum, count word nodes.

Note that words in the program are first split (e.g., "numbersList" = "numbers" + "list") and trimmed (e.g., "numbers" to "number") and then added as word nodes to the graph. Thus, for each word in the program (including all its grammatical forms), the PFIS-V graph contains a unique word node.

Moving on to the links between patches, again consider method average in Figure 4.1(a). A programmer can go from this method to other methods, such as sum or count, by clicking on these method invocations (links). Similarly, s/he can navigate from the method count to the methods sum and average by scrolling. These navigation

affordances (or links) are modeled as links between the average method and the sum, count and average methods respectively (The labels "*inv"* and "*adj"* on the link edges represent method invocation and adjacency respectively.)

To represent multiple variants, PFIS-V extends PFIS3's single-variant data model in four different ways, resulting in four different configurations. Each of these configurations represent different assumptions about programmers' mental models of the variational information space.

Before we enter the discussion about these representations, let us consider an example. Consider a programmer, Jane, starting with Variant 1 of a program (Figure 4.2 (a)). The first variant of program consists of four method patches A, P, Q, R. (We have excluded the word nodes from the graph for ease of illustration.)

Jane modifies the method R and saves it as Variant 2. We indicate that R has changed between Variants 1 and 2, by naming it R' in Variant 2. In a similar way, Jane again changes method R (resulting in R''), adds a new method S to result in Variant 3. Finally, she modifies the new method S (resulting in S') and removes method A, resulting in Variant 4. We will now discuss how these variants are represented in PFIS-V's four configurations.

Configuration #1: Variant-unaware data model

The *variant-unaware* data model, illustrated in Figure 4.2(a), is the simplest of the four data model configurations. Here, each patch in each variant is represented as a unique patch node, irrespective of whether two patches have the same names (e.g., R and R'), or are somewhat similar, or even identical across variants (e.g., P). For each word in the environment (across variants), there is a single word node, irrespective of how many variants or methods the word occurs in.

This configuration accurately reflects the navigation affordances in our study environment (Figure 3.2, page 22), as well as in most other IDEs. For example, a programmer could not navigate from a method in one variant to a method in another variant, via an IDE links; this data model contains no such navigation links going across variants. Therefore, we also use this configuration as a baseline for our evaluation of the other data model configurations.

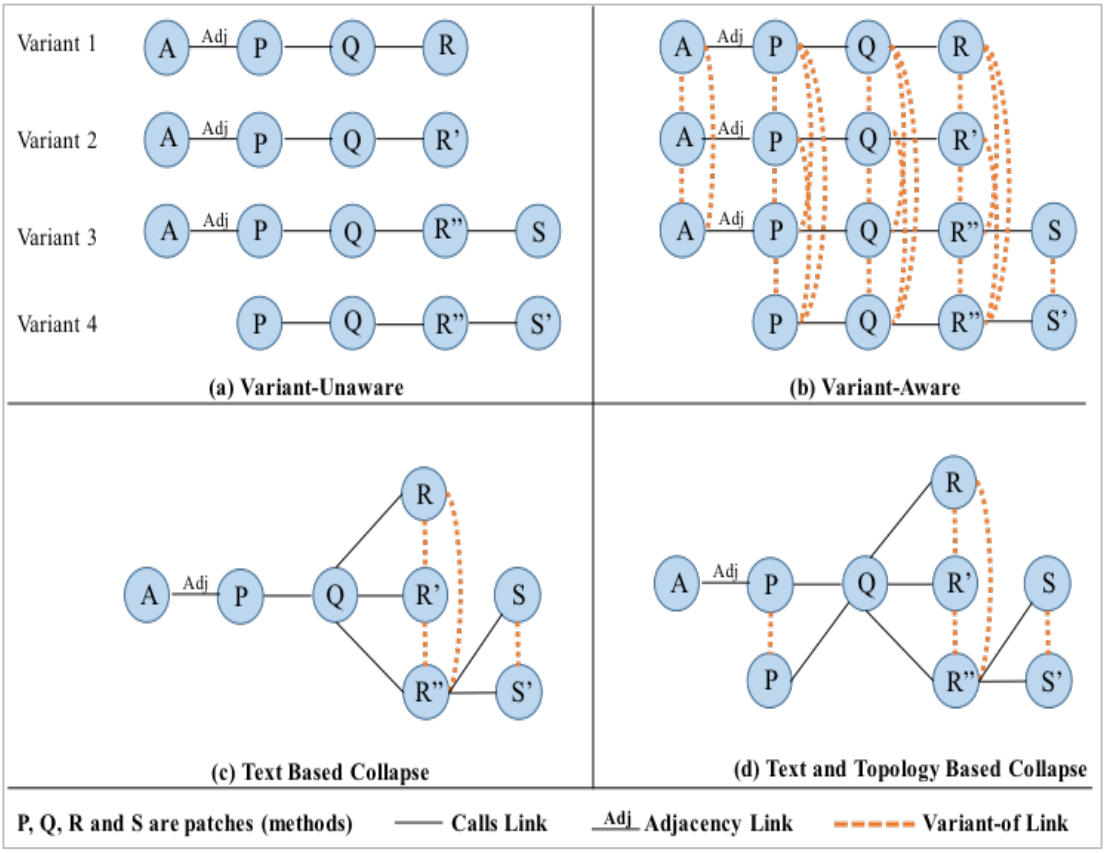

Figure 4.2. Four PFIS-V data model configurations. (a) The variant-unaware configuration accurately reflects the navigation affordances in our programming environment and does not capture any similarity between variants. (b) To this, we introduced variant-awareness by adding "variant-of" edges (dotted lines) between similar patches in different variants. Further, we collapsed identical patches across variants, calling them equivalent; (c) text-based equivalence is based on identical text and (d) text-and-topology-based equivalence is based on identical text as well as same neighbors.

However, contrary to what this data model captures, a programmer like Jane might expect methods with the same name (but in different variants) to be identical, or at least similar in some way. For example, if she sees that the method R computes the rate of interest in one variant, she might expect the method R (dubbed here as R' and R") to do the same thing in other variants, even if the method body has changed. Whereas the variant-unaware configuration does not capture this similarity property of variants (and hence gets the name variant-unaware), the variant-aware configuration captures them.

Configuration #2: Variant-aware data model

The variant-aware data model considers one possible mental model of similarities between patches. In this data model configuration, patches having the same fully qualified names (i.e., package, file and method names) but in different variants, are considered to similar and are linked via "variant-of" links. For example, in Figure 4.2 (b), all patches with the same name (e.g., R, R' and R" or all Ps) are variants of each other; therefore, they are linked by dotted lines representing "variant-of" links.

Note that, unlike other links, such as adjacency or method invocation links, the "variant-of" links are not physical navigation affordances in the environment; instead, they are conceptual links that capture the relationship between patches that may exist in a programmers' head.

Configurations #3 & #4: Variant-and-equivalence aware data models

The next two configurations are the variant-and-equivalence aware data models. As the name suggests, these data models are both variant-aware and equivalence-aware. The term variant-aware is exactly the same as in the previous configuration: patches that have same fully qualified names are similar across variants and are connected via "variant-of" links. For example, see the dotted edges between R, R' and R'' in Figure 4.2(c) and (d).

In addition, these data models also add the notion of "equivalence" between certain patches. Equivalence means that, from the perspective of a forager, it does not matter which of two equivalent patches s/he forages in. For example, in Figure 4.2(b) all the methods named P are identical across variants: they are equivalent for foraging purposes because they all provide exactly the same information, cues and scent to the predator. To model this perspective, in the variant-and-equivalence-aware data models (Configurations #3 and #4), we replace multiple equivalent patches with only a single representative patch. The difference then, between these two configurations is in how equivalence is computed.

In Configuration #3, *text-based equivalence* models that programmers considered two patches to be equivalent if they had identical textual contents. For example, in

Figure 4.2 (a) and (b), all the methods P have exactly the same content across variants—there are no there are no P' and P''. Therefore, in Figure 4.2(c), all methods P are considered equivalent and collapsed into a single node.

In Configuration #4, *text-and-topology-based equivalence* models programmers differentiating between patches that have exactly the same content, but different topologies across variants (e.g., whether the method moved across variants). For example, in Figure 4.2(d), P is adjacent to A in Variants 1,2 and 3, but not in Variant 4. The intuition here is that the similarities (or differences) in topology might be important to variations foraging, just as the topological layout of methods in a program is important to programmers' foraging within a single program variant [63].

Later, in Section 4.2, we will empirically compare these data model configurations to see which of them is the most predictive of programmers' variations foraging behaviors (and hence closely resembles programmers' mental models of variants).

4.1.2 PFIS-V algorithm: Predicting programmer navigations based on their mental models

In a computational model like PFIS-V, the data model is only one part: it only models the information that the programmer knows about so far. However, when foraging based on that information, a programmer has several possible navigation options: s/he could scroll up or down from the current method to an adjacent method, s/he could go back to the previous method that s/he had already been to, or follow the link to a method called from another method or go to a method containing some words in the method name or the body, forage within the same variant or go to another variant. The predictions for which of these navigations a programmer will make is made by the predictive algorithm of the model.

The PFIS-V algorithm (drawn in part from the PFIS3 algorithm for single-variant foraging) takes as inputs: 1) the data model graph, representing the patches, cues and links the programmer knows about so far and 2) the list of navigations the programmer has made so far. Based on these inputs, the algorithm computes how likely it is for a programmer to make each of the available navigation choices based on the scent s/he will perceive. PFIS-V computes this scent in two stages as follows.

Stage #1: Activation. First, the algorithm initializes the weights for the patch nodes the programmer has navigated to so far (i.e., the programmers' navigation history). Specifically, if the algorithm is predicting navigation H_{K+1}, and H_1, H_2… H_K represent the patches the programmer has navigated to so far, then the activation step assigns the initial weight:

$$H_i = \begin{cases} 1.0 & for\ i = K \\ \alpha H_{i+1} & for\ i = 1,2..K01 \end{cases}$$

Here, α is the decay factor. We used α=0.9, preserving the value from WUFIS, the web-foraging model [10], on which PFIS is based. Thus, the most recent patch, namely the patch the programmer is currently in, receives an activation 1.0, the previous patch gets 0.9, and the earlier patches get 0.81, 0,729 and so on. All the other patch and word nodes in the graph are initialized with weight 0.0.

From a code navigation perspective, the activation step operationalizes "recency", which prior studies (e.g., [5, 60, 63]) have revealed to be an important predictor of programmer navigations: a programmer often revisits patches that s/he has recently foraged in.

Stage #2: Spreading the activation. In the second stage, the algorithm spreads the initial activation weights from the activated patches to other related patches, decaying the weight by a factor of β=0.85. (We preserved the value of β from the WUFIS model [10].) This spreading proceeds in parallel along two distinct paths.

In the first path, the algorithm spreads the initial activation from patch nodes to word nodes and, in turn, from word nodes to other patch nodes via "patch contains word" edges. Thus, the algorithm spreads weights to patches that are lexically similar to the current patch, thereby capturing the scent programmers perceive from words in source code. Since, in one instance, a programmer will attend to only one cue, or follow only one link, among the several available choices, the PFIS-V algorithm accounts for the probability that a programmer will attend to a certain cue. Therefore, the spreading from patch P to patch Q, via word W is given by:

$$W = W + P * 0.85 * \frac{1}{no.of\ neighbors\ of\ P\ in\ the\ graph},$$

$$Q = Q + W * 1 * \frac{1}{no.of\ \ neighbors\ of\ W\ in\ the\ graph}$$

Here, β=0.85 indicates a decay factor and is an indicator of how much weight the PFIS-V algorithm assigns to the lexical-similarity factor while making predictions. Note that the decay is applied only to the first step, otherwise, the algorithm would have double decayed the weights.

Simultaneous to spreading weights along the "patch contains word" edges, the algorithm also spreads activation from patch nodes to other patch nodes along topological links, namely adjacency and method invocation edges. This spreading models that a programmer could follow one of these several available links to navigate to the next patch. PFIS-V also accounts for the probability that a programmer will follow a link, while spreading activation along links. Thus, in spreading weight from patch P to patch Q, the PFIS-V algorithm assigns:

$$Q = Q + P * 0.85 * \frac{1}{no.of\ neighbors\ of\ P\ in\ the\ graph}.$$

Here again, β=0.85 is the decay factor; note that the weights are equal for the topological links and word-similarity links[4].

At the end of these spreading steps, the resultant weight on each patch node is a measure of the likelihood that the programmer will navigate to that patch; in other words, it is a measure of scent: the programmer will follow the path that provides the strongest scent.

The algorithm then ranks the patch nodes based on the weights (highest weight gets lowest rank and vice versa, where rank=1 is the algorithm's top prediction). If several patch nodes have exactly the same weight, then the algorithm resolves the tie by

---

[4] In the version of PFIS-V presented in [77], the algorithm double decayed weights while spreading along "patch contains word" edges, thereby biasing the model towards navigations via topological links than via word similarity. The earlier version also did not: 1) spread activation in parallel resulting in multiplicative effects between the two spreading steps and 2) consider similarity or equivalence between the top-of-file declarations across variants.

assigning an average rank. For example, if k patches are tied at rank r, then the algorithm assigns a rank:

$$R = r + \frac{k+1}{2}.$$

The algorithm then "rates itself" by returning the rank of the patch to which the participant actually navigated to. This rank is an indicator of the accuracy of the prediction: if the algorithm assigned rank 1, then the actual navigation was also the algorithm's top prediction. On the other hand, if the algorithm returned a high rank (lower scent), then the algorithm mispredicted the navigation. As we shall see later, we will use this accuracy measure to evaluate the model, and hence gather evidence towards (or against) the hypotheses encoded in the model.

The spreading activation steps and ranking mechanisms described so far are common to both PFIS3 and PFIS-V algorithms. The difference then, between the two algorithms, is PFIS-V implements the following variations-specific extension.

Variations-specific extensions:

Recall that programmers capitalized on the similarities between variants during their variations foraging. The data model configurations of PFIS-V, described in Section 4.1.1, captured *what* similarities programmers perceive, and formed a part of their mental models. The PFIS-V algorithm operates on these data models and predicts *how* a programmers will use those similarities (in their mental models) to make navigation decisions, such as cost-value estimations. Specifically, it implements the following two extensions (highlighted in blue in Figure 4.3).

First, PFIS-V accounts for the fact that programmers might navigate between similar patches (patches with same file, folder and method names) in multiple variants, even when a direct link between those patches is absent. For example, some participants in our study navigated to the view.js file, drawScoreBoard() method in one variant after the other. To model this behavior, the PFIS-V algorithm spreads activation along the "variant-of" edges while spreading activation via topological links.

Second, as the following paragraphs detail, PFIS-V models that a programmer will know about the existence of a patch in one variant and will be able to estimate its costs and values, based on what s/he has seen in other variants.

Definitions:

1: Patch Set P: set of all patches the programmer has seen so far.

2: Word Set W: set of all words in all the patches in P

3: Graph G = (N_P U N_W, E_P U 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.

4: Navigation history H: sequence of patches to which the programmer has navigated so far.

Activate (G, H[1..k]):

5: **for** each node N in G:
set Weight (N) ← 0.0
6: **for** i ← k **down to** 1:
set Weight(H[i]) ← 1.0
7: **for** i ← k-1 **down to** 1:
8: **set** Weight(H[i]) = α * H[i+1]; α = 0.9

Spread (P, Q, decay, no. of neighbors)

9: Probability = 1.0 / no. of neighbors
10: Weight (Q) = Weight (Q) + Weight(p) * decay * probability

Steps to predict the (k+1)th patch in H:

11: **if** programmer has not seen exact patch earlier:
12: **if** programmer has seen a similar patch *s* in another variant:
13: Approximate the content of (k+1)th patch to the contents of *s*.
14: Else, return "unknown"
15: Activate(G, H[1..k])
16: For each patch node P in graph:
17: For each neighbor N of P:
18: Spread (P, N, 1.0, count(N))
19: For each node P in graph:
20: For each neighbor N of P (along all edges):
21: Spread (P, N, β=0.85, count(N))
22: Rank the patch nodes in the decreasing order of activation.
23: If t patch nodes are tied at rank r,
24: Assign rank = [r + (t-1)/2] to all t patch nodes.
25: Return the rank for the node representing the $(k+1)^{th}$ patch.

Figure 4.3 PFIS-V's algorithm (Lines in blue are additions from the original PFIS3 algorithm.).

Traditionally, IFT computational models such as PFIS-V predict navigations to only those patches that a programmer already "knows about" and can estimate the costs and values for. For example, if a programmer has navigated to a method average(), then s/he knows about the exact costs and values for that patch. Also, if average() calls sum(), then the programmer knows about sum() also. In fact, the programmer might be able to guess the cost and value for sum() based on cues, such as words in the method name, even though s/he might not know its exact content. Thus, PFIS-V makes a prediction when a participant navigates to a known patch; *known patch = seen the exact patch, or its name.* For all other navigations to patches that the programmer does not already know about (e.g., by opening a file from the package explorer at random), the model makes no prediction; we call these navigations "unknowns".

However, this notion of known vs. unknown patches changes subtly in variations foraging, where a programmer might know about a patch without seeing the exact patch or its name but based on what s/he has seen in other similar variants. For example, if a programmer has already seen the *average()* method in *Variant 1,* s/he might expect to find *average()* in *Variant 2* also. Similarly, if *average()* computes the average mileage in *Variant 1*, the programmer might expect *average()* to compute average mileage (i.e., contain similar values and costs) in *Variant 2* also.

Therefore, PFIS-V predicts navigations to patch when: 1) the programmer has seen exactly the same patch or its name, OR 2) the programmer has seen a similar patch in another variant. In the latter case, PFIS-V approximates the contents of the patch to be the same as that in the last-seen similar patch in another variant—the idea being that the programmer will expect the contents to be very similar to what they seen previously in other variants.

In the next section, we will use this PFIS-V algorithm and evaluate its predictions under each of the four data models.

4.2 PFIS-V evaluation

To evaluate PFIS-V, we used the model to predict the navigations made by 7 participants in the user study described in the previous chapter; we dropped the eighth participant, namely P05, due to incomplete data.

Recall that participants in the study foraged among 700+ variants of the Hextris game to complete a reuse task. In doing so, the 7 participants made a total of 665 navigations going from one method to another (between-method navigations). We used PFIS-V to predict these between-method navigations, in all four data model configurations. We used click-based navigations [63] where a navigation is defined as a change in cursor position[5].

For comparison purposes, we also predicted programmer navigations using PFIS3. Since both PFIS3 and PFIS-V algorithms work with all four data model configurations, we use an algorithm/data-model notation to disambiguate these combinations; for example, PFIS3/variant-unaware refers to the PFIS3 algorithm with the variant-unaware data model.

4.2.1 PFIS-V vs. PFIS3 algorithms

To compare the predictive ability of PFIS-V and PFIS3 algorithms, we used two measures, namely unknown rate and hit rate.

Unknown rates: how many navigations can PFIS-V predict?

Recall from Section 4.1.2 that models such as PFIS-V only predict navigations to patches that the programmer already knows about; they cannot predict a navigation to a patch they do not know exists. Thus, whenever a programmer navigates to an entirely new patch, the prediction is always a failure and we denote this as "Unknown". For example, an unknown rate of 60% would mean that the model failed to predict 60% of all programmer navigations; thus, lower unknown rates are better. Note that both PFIS-V (or PFIS3) predict exactly the same set of navigations, and hence yield the same unknown rates, in all data model configurations.

[5] Another operationalization of navigation, namely view-based navigation, also accounts for navigations via scrolling which are not accompanied by a change in the cursor position. [63] revealed that a model's predictive accuracy might vary depending on the choice of click-based vs. view-based navigation. However, we followed prior work in programmer recommendations and used click-based navigations [42, 64].

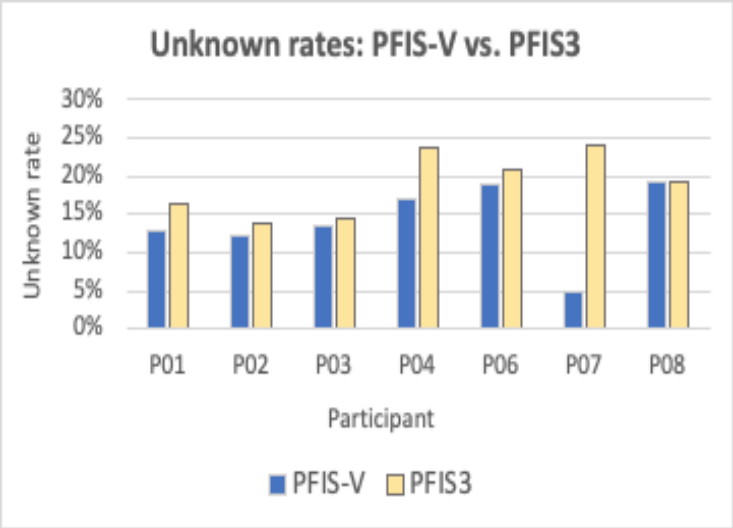

Figure 4.4. PFIS-V vs. PFIS3 Unknown rates. While predicting participants' between-method' navigations, PFIS-V had fewer unknown patches, and hence predicted more navigations, than PFIS3 did. (Note that lower unknown rate is better.)

Figure 4.4 compares the unknown rates of PFIS-V (blue) and PFIS3 (yellow) while predicting individual participants' navigations[6]. We see that, for 7 out of 8 participants, PFIS-V's unknown rates were lower than that of PFIS3 indicating that fewer navigations were unknown to PFIS-V than to PFIS3. The one exception is P08, where PFIS-V and PFIS3 yielded similar unknown rates: this is because, P08 foraged in methods in only one variant, namely the destination variant, and PFIS-V behaves very similar to PFIS3 in single-variant situations. This ability of PFIS-V to predict the same, or more, navigations than PFIS3 provides the first evidence towards the efficacy of PFIS-V in predicting participants' variations foraging.

---

[6] In this thesis, we use the latest versions of PFIS3 and PFIS-V that implement the following changes (since [78]). First, similarity and equivalence are also computed for top-of-file declarations (modeled as method patches). Second, the spreading of activation proceeds in parallel for all nodes. Third, the algorithm does not decay weights while spreading from word to patch nodes to prevent double decay (patch-to-word, word-to-patch) while spreading along "patch contains word" edges; this way, lexical similarity and topology factors are assigned equal weightages for making predictions.

However, unknown rates are a measure of how often a model entirely fails to make a prediction, and do not provide any insight into how accurate those predictions were. Our next measure, namely hit rates, evaluates the accuracy of the predictions.

Hit rates: how accurate are PFIS-V's predictions?

As described earlier (Figure 4.3), models such as PFIS-V make multiple predictions for each navigation, and rank them (its top choice=1, its next choice=2, …). If the algorithm assigns a low rank to a patch where the programmer actually navigated to, then the prediction is a "hit"; otherwise it is a "miss". H*it rate(threshold=K)* refers to the percentage of all actual navigations that a model predicted within its top K ranks ("hits"). For example, *hit rate(threshold=10)* = 90% would mean that a model predicted 90% of participants' navigations within its top 10 ranks. Thus, higher hit rates, especially at lower thresholds, are better.

Following prior work (e.g., [42, 64]) on predicting and recommending programmer navigations, we used hit rate (threshold=10) as our default measure of accuracy. In the rest of this thesis, the terms "accuracy" and "hit rate" generally refer to *hit rate (threshold=10)*, unless specified otherwise.

Figure 4.5 compares the average hit rates from PFIS-V and PFIS3 for various rank thresholds; solid lines represent PFIS-V, dotted lines represent PFIS3, and the four colors correspond to the four data model configurations. As the graph indicates, on an average, PFIS-V yielded higher hit rates--and therefore was more accurate--than PFIS3 in all data model configurations.

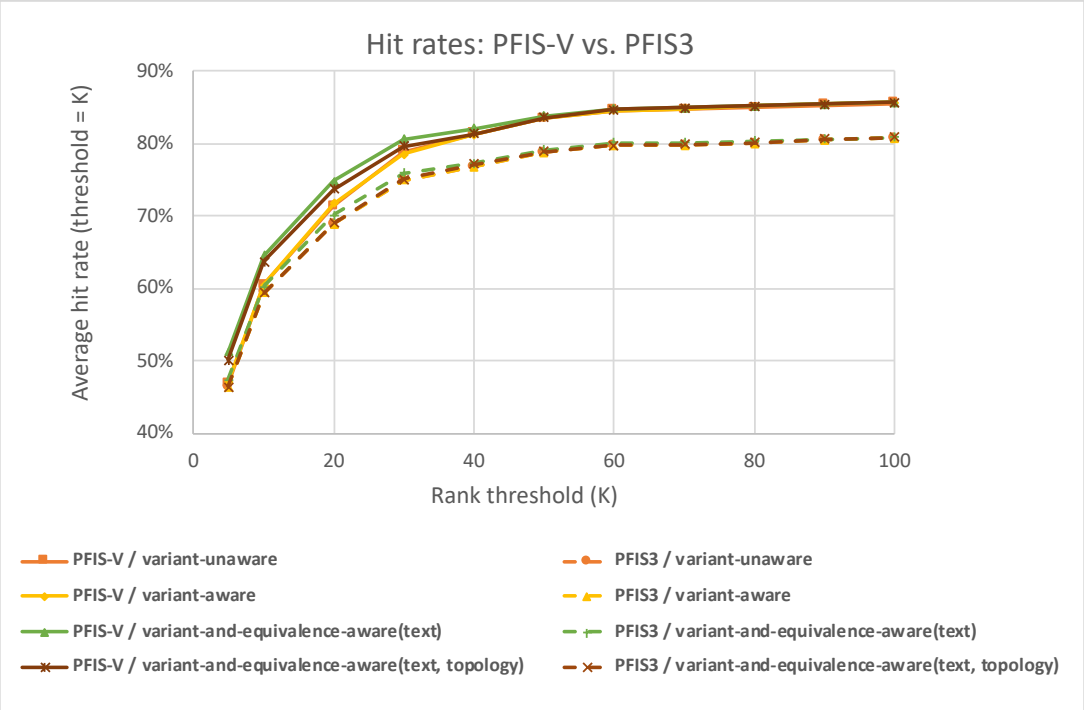

Figure 4.5. PFIS-V vs. PFIS3 Hit rates. On an average, PFIS-V yielded higher hit rates than PFIS3 across all data model configurations. (Higher hit rates are better.)

Drilling down to individual participants, PFIS-V was more accurate than PFIS3 while predicting individual participants' navigations also. As Table 4.1, columns d-g, show, for 5 out 7 participants, PFIS-V (black) was more accurate than PFIS3 (orange); the improvements were as high as 19.23% for P07 (columns f, g). For the remaining two participants (P03, P08), PFIS-V matched to PFIS3's accuracy. These improvements in hit rates indicate that PFIS-V was a more accurate predictor of participants' variations foraging than PFIS3.

Participant (a)	No. of variants (b)	No. of between-method navigations (total=665) (c)	Per participant hit rate (rank threshold = 10) black = PFIS-V; orange = PFIS3			
			variant-unaware (d)	variant-aware (e)	variant-and-equivalence-aware(text) (f)	variant-and-equivalence-aware(text, topology) (g)
GROUP-1						
P04	5	89	65.17%	65.17%	69.66%	69.66%
			64.04%	64.04%	64.04%	64.04%
P07	21	104	49.04%	49.04%	73.08%	67.31%
			48.08%	48.08%	53.85%	48.08%
GROUP-2						
P01	3*	55	83.64%	83.64%	83.64%	83.64%
			80.00%	80.00%	80.00%	80.00%
P02	2	116	70.69%	70.69%	70.69%	70.69%
			69.83%	69.83%	69.83%	69.83%
P03	2	132	51.52%	51.52%	51.52%	51.52%
			51.52%	51.52%	51.52%	51.52%
P06	2	101	47.52%	47.52%	47.52%	47.52%
			46.53%	46.53%	46.53%	46.53%
P08	2	68	55.88%	55.88%	55.88%	55.88%
			55.88%	55.88%	55.88%	55.88%
* P01 made an additional navigation a third (potential source) variant.						

Table 4.1. PFIS-V vs. PFIS3: Per-participant hit rates. Black = PFIS-V; orange=PFIS3. For all data models and participants, PFIS-V was similar or more accurate than PFIS3. In particular, the PFIS-V/variant-and-equivalence-aware(text) model yielded higher average accuracies, particularly benefiting Group-1 participants (P04, P07). For all other participants, equivalence-awareness did not bring any additional improvements over the PFIS-V/variant-aware model.

The fact that PFIS-V not only predicted more navigations than PFIS3 (lower unknown rates) but also did so with higher accuracy (higher hit rates) provide evidence supporting the assumptions encoded in the PFIS-V algorithm. Recall that PFIS-V made models that, even though a programmer has never seen a patch earlier, s/he might capitalize on the similarities between variants to infer the existence of the patch, and to estimate its costs and values (i.e., based on what s/he has seen in other similar variants).

4.2.2 Data model configurations: which one is closer to programmers' mental models?

Another set of assumptions encoded in PFIS-V are in its four data model configurations. They implement different assumptions about participants' mental models of the variational information space: 1) patches with the same name and in different variants are somewhat similar in terms of their costs and values (variant-of links), 2) patches containing identical content are equivalent and it does not matter which one of those identical patches a predator forages in (collapsed patches) and 3) in comparing variants, participants not only attended to lexical similarities and differences but also compared the topology of patches across variants (text-based vs. text-and-topology-based equivalence). To evaluate which of these assumptions closely represent participants' mental models, we compare PFIS-V's accuracy across the four data model configurations.

In Figure 4.5, compare the four solid lines. First, introducing the notions of variant-awareness (i.e., variant-of links) resulted in slightly higher hit rates for the variant-aware configuration than the variant-unaware configuration, suggesting that participants did navigate among similar patches in different variants.

Second, comparing the variant-aware and the variant-and-equivalence-aware models, we see that modeling equivalence between patches (i.e., collapsing identical patches) resulted in higher hit rates. This result suggests that participants' mental models of variants included whether patches were identical (or not) and that they considered identical patches equivalent.

Third, among the two variant-and-equivalence-aware models, text-based equivalence resulted in higher predictive accuracy than the text-and-topology-based equivalence, suggesting that participants' comparisons of source-code patches were generally based on textual content (e.g., do these methods contain the same text?) than on the source topology (e.g., has the method moved?)[7].

---

[7] In the earlier version of PFIS-V [78], the variant-and-equivalence-aware(text, topology) model yielded slightly higher hit rates than the variant-and-equivalence-aware(text) model. This is because, in the earlier version: 1) the spreading decay was higher for word-similarity than for topological relationships and 2) the spreading of

4.2.3 Two groups: different between-variant foraging behaviors

However, as Table 4.1 shows, there existed two distinct groups of participants. For Group-1 participants (P04, P07), PFIS-V's accuracy improved while progressing from the variant-unaware to variant-aware to variant-and-equivalent-aware models, as the four distinct lines in Figure 4.6(a)) show.

In contrast, for Group-2 participants (P01, P02, P03, P06, P08), the four hit rate lines overlap in Figure 4.6(b), suggesting that the PFIS-V hit rates were very similar across all data models. Table 4.1 (rows in black, columns d-f) also reveals these differences between the two groups.

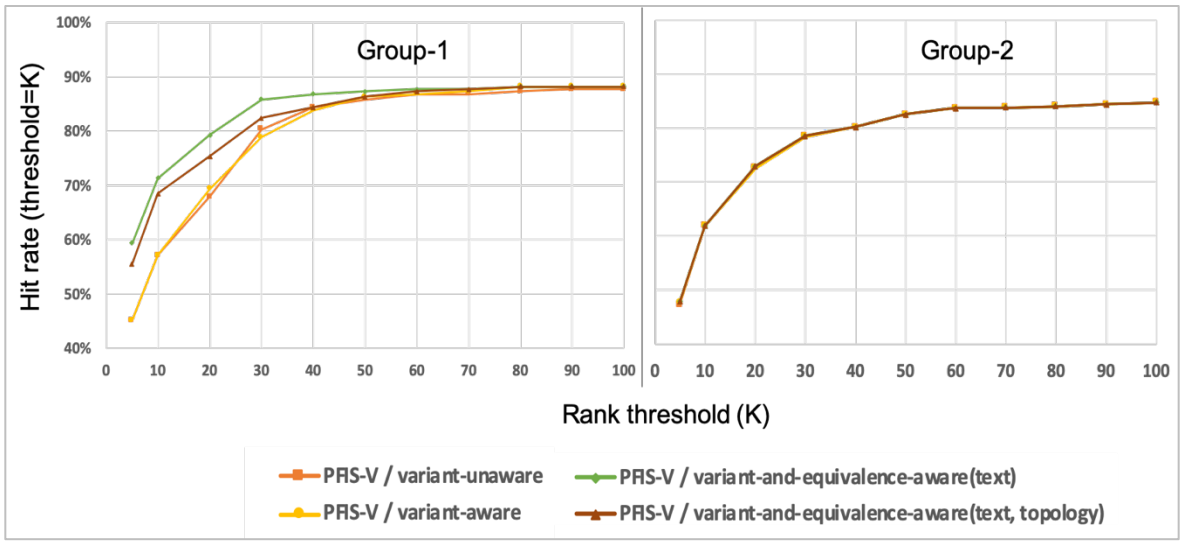

Figure 4.6. PFIS-V improvements: two groups of participants. For the Group-1 participants (5 out of 7), the average PFIS-V hit rates across all data models were very similar; however, for Group-2 participants, the variant-and-equivalence-aware models were more accurate than the other two models.

In order to reason about this dichotomy, let us revisit the assumptions the variant-aware and the variant-and-equivalence-aware data models make: 1) programmers will navigate to similar patches in similar locations across variants (variant-aware) and 2) programmers will attend to similarities and differences in patches across variants

---

activations proceeded in a non-parallel fashion, leading to multiplicative effects between textual and topological relationships.

(equivalence-aware). Ideally, the more variants a programmer forages in and the more similar (or identical) patches s/he encounters, the more closely the variant-aware or the variant-and-equivalence-aware models will reflect programmers' foraging (higher accuracy).

This is exactly what happened with Group-1 participants. In Table 4.2, graphs for Group-1 participants had more "variant-of" edges and collapsed nodes than Group-2 participants. This is because, while foraging for an appropriate source variant (between-variant foraging), Group-1 participants navigated to methods in several variants. Specifically, they 1) navigated to similar locations in different variants and 2) attended to the similarities and differences in source-code by way of processing cues, just like the "variant-of" links and the collapsed nodes model. Therefore, the variant-and-equivalence-aware models resulted in higher PFIS-V accuracy for Group-1 participants. (Notice that even among Group-1 participants, P07 visited way more variants than P04 and benefiting heavily from the "variant-of" edges and collapsed nodes).

Participant	Variant-unaware			Variant-aware			Variant-and-equivalence-aware					
							Text-based equivalence			Text-and-topology-based-equivalence		
	Methods	Edges	Variant-of edges	Methods	Edges	Variant-of edges	Methods	Edges	Variant-of edges	Methods	Edges	Variant-of edges
GROUP-1												
P04	275	623	0	275	719	96	250	656	63	262	660	67
P07	327	822	0	327	2458	1636	219	1150	596	289	1933	1184
GROUP - 2												
P01	178	380	0	178	406	26	176	396	22	178	406	26
P02	192	415	0	192	437	22	191	435	20	192	437	22
P03	202	487	0	202	492	5	202	492	5	202	492	5
P06	222	540	0	222	557	17	222	557	17	222	557	17
P08	164	370	0	164	370	0	164	370	0	164	370	0

Table 4.2. Two groups of participants: PFIS-V data model graphs. Group-1 participants foraged in source-code patches in several variants and the variant-aware and variant-and-equivalence aware models closely resembled these navigations. In contrast, for Group-2 participants who foraged in only two variants, the four data model graphs (and hence accuracy) were very similar.

In contrast, for Group-2 participants, there were fewer "variant-of" edges and no collapsed nodes in Table 4.2 (all four data models have the same no. of nodes). In other words, PFIS-V did not benefit from the variant-awareness and equivalence-awareness improvements. This is because, unlike Group-1 participants who relied on similarities and differences in methods for their between-variant foraging, Group-2 participants relied exclusively on words in *changelogs or cues in the game's output* to lead them to an appropriate source variant. As a result, they foraged in source-code patches in only two variants, namely the source and the destination variant, resulting in little predictive advantage from the variant-aware and variant-and-equivalence-aware models.

Then, how does PFIS-V model Group-2 participants' between-variant foraging in textual changelog and graphical output patches? Unfortunately, it does not! In fact, no prior IFT computational model, in programming as well as non-programming domains, has accounted for non-textual patches (e.g., graphical outputs). This reveals a gap in the state-of-the-art IFT computational modeling.

But before we go ahead to address this gap and to model changelogs and outputs, let us discuss the implications of our results.

4.3 Implications: Designing for variants

So far, PFIS-V's predictions of participants' navigations provided an apparatus for us, as researchers and tool builders, to test our hypotheses about variations foraging in an 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 versus text-and-topology-based similarity.

However, the benefits of models such as PFIS-V are not limited to such theoretical understanding. They can also be adopted to practical tool design, such as for tool builders to evaluate the gaps in their existing tools. For example, common version control tools such as Git employ equivalence in their interfaces; they highlight differences between variants and hide away what is unchanged between them. Indeed, in our evaluation of PFIS-V, the introduction of equivalence resulted in higher predictive accuracy. However, as our results also suggest, tools could provide more

navigation affordances for variants, such as navigation between similar patches across variants (e.g., easily forage through all versions of a method).

The higher accuracy of variant-aware data models reveals the importance of navigation affordances between similar patches across variants ("variant-of" links). Further, the comparison results of the four data models reveals that the variant-and-equivalence-aware(text) 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.

However, these improvements in accuracy from the variant-aware and the variant-and-equivalence-aware models were limited to when participants (e.g., Group-1) foraged among *source-code* patches across variants. In our study, the majority of participants fell into Group-2 and foraged exclusively in *non-code patche*s, namely textual changelogs and graphical outputs, for their between-variant foraging. Since PFIS-V does not account for non-code patches, it fails to accurately model Group-2 participants' variations foraging behaviors. In the next chapter, we will begin addressing this gap by accounting for participants' foraging in non-code patches.

4.4 Open problem: what about modeling non-code patches?

This limitation in PFIS-V, namely that it does not model non-code patches, is shared by almost all IFT-based predictive models--in programming as well as non-programming domains. They model foraging among only textual patches and do not account for non-textual patches, such as videos, audio, interactive content, or graphical content (e.g., graphical outputs).

Given the prevalence of information environments with heterogeneous patch types—documents contain text and images, web contains text, images, video, audio and interactive games—we believe that 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 foraging in variants and information foraging in general.

However, expressing non-textual patches in IFT computational models, parsing their information features and computing similarities and differences is a non-trivial

problem, let alone modeling programmers' foraging behavior heavily involving visual comparisons, as in our study. We consider this to be an important new research opportunity in the area of computationally modeling variations foraging. In the next chapter, we will begin addressing this gap.

CHAPTER 5: PFIS-H: MODELING PROGRAMMERS' VARIATIONS FORAGING IN NON-CODE PATCHES AND HIERARCHIES

During programming tasks, programmers, including our study participants, forage in different kinds of information, such as code, outputs, changelogs, design documents and software visualizations. Prior IFT research has predominantly focused on understanding how programmers forage in code but leaves gaps in our understanding of the other kinds of foraging. For example, how do we account for the way programmers connect the outputs they inspected to the code they inspect? How do change logs figure into their foraging through variants? And how do programmers choose which part of the IDE (e.g., code vs. changelog vs. output) to forage in?

In this chapter, we consider "hierarchical foraging" as a potential answer to these questions. The idea here is that some of programmers' foraging choices—such as the navigations to changelogs vs. outputs vs. code in our formative user study—take the hierarchical organization of the IDE (e.g., project contains packages contain classes contain methods) into account. Therefore, our new computational model, namely PFIS-H (or PFIS for Hierarchies), models hierarchical foraging to predict programmers' navigations in code and non-code patches. Further, to consider hierarchical foraging in a way that does not restrict people's foraging to strictly textual patches, we add treatment of non-textual patches (that prior IFT computational models have largely ignored).

5.1 PFIS-H data model

The data model in PFIS-H similar to that in PFIS-V: it is a graph representation of the information environment--the patches, cues, links--that the programmer has seen so far. Additionally, to account for hierarchical foraging and non-code patches, PFIS-H makes the following two extensions to its data model.

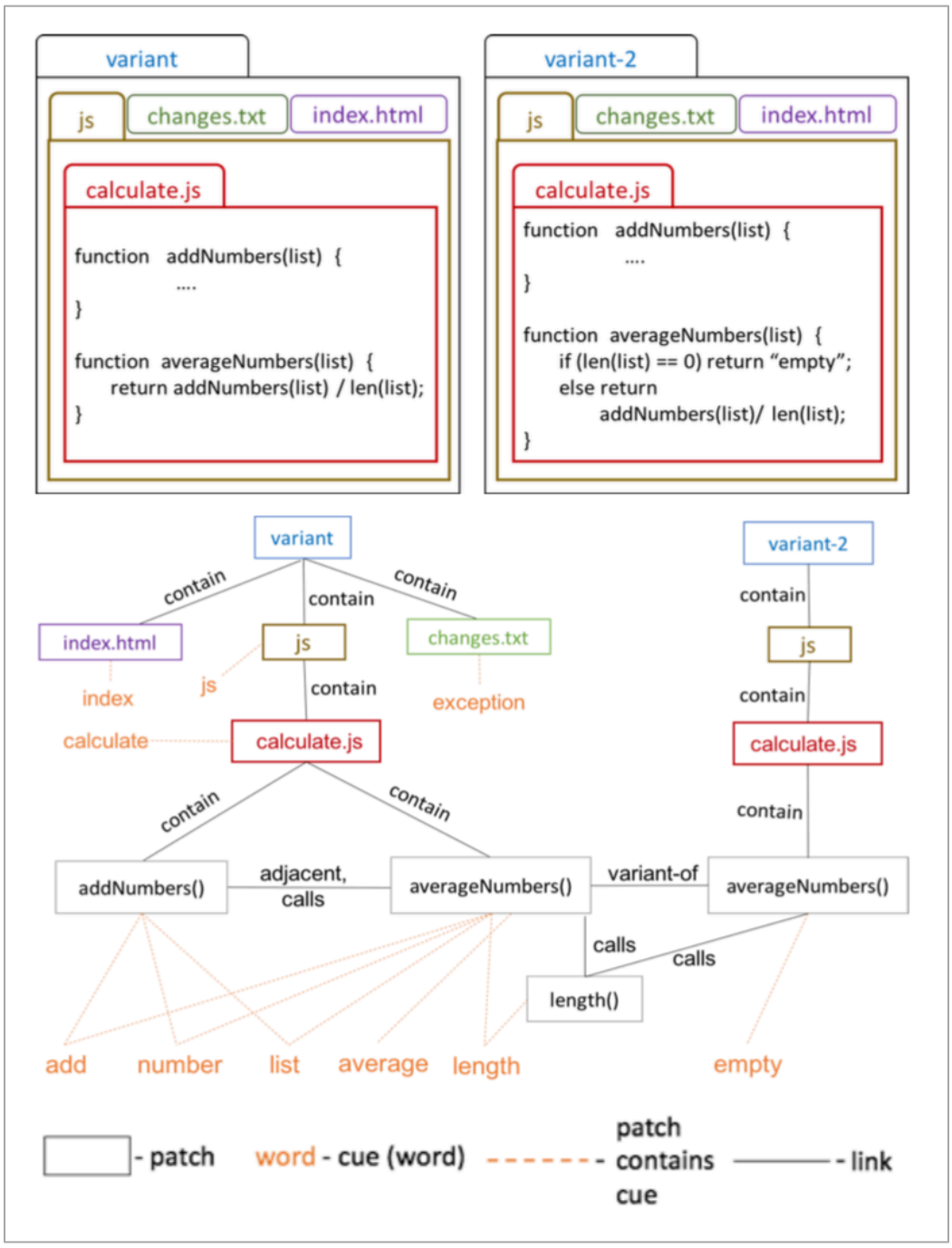

Figure 5.1. The PFIS-H data model. (top = example program, bottom = corresponding PFIS-H data model graph). The PFIS-H data model: 1) accounts for non-code patches (changes.txt=changelog, index.html=output) and 2) models "patch contains smaller patch" relationships via "contain" edges which form a hierarchy.

First, the PFIS-H data model includes the hierarchy of patches in the environment as follows. Whenever a patch contains one or more smaller patches, the PFIS-H data

model graph includes a "contains" edge between the two patch nodes[8]. For example, in Figure 5.1 (top), the variant contains a "js" folder. Correspondingly, the data model graph (bottom) includes a "contain" edge between the variant and "js" nodes. Similar "contains" edges indicate that the "js" folder contains calculate.js file, which in turn contains the *addNumbers()* method. Also note that the variant contains changes.txt (changelog) and index.html (output), as in our study environment.

Second, similar to the source-code patches that a programmer has seen, the PFIS-H data model also includes non-code patches that a programmer has seen in the study environment. Specifically, PFIS-H models textual changelogs and outputs, including non-textual (graphical) ones, as follows.

Modeling changelogs: Starting with changelogs, in one way, changelogs are similar to methods: they are both textual. Therefore, we represented changelog patches similar to how we modeled methods. For each changelog patch, we included a node in the graph. The words in changelogs were represented as word nodes and were linked to the changelog nodes to indicate "patch contains word" relationships.

However, changelogs also fundamentally differ from method patches in that changelog patches are all about what is different about a variant; in contrast, method patches can sometimes be identical across variants (and hence could be collapsed as equivalent). Thus, the notion of equivalence does not apply to changelog patches. However, PFIS-H still models that all changelogs patches across variants are similar in the sense that they all contain change information; therefore, all changelog patches are connected to each other via "variant-of" links to indicate their similarity.

Modeling outputs: Modeling a generic output patch is hard since outputs can come in different formats: some programs produce textual output, others might produce

---

[8] Earlier versions of PFIS including PFIS3 and PFIS-V, included "contains" edges in the data model graph, thereby modeling the patch hierarchy. However, they did not affect the scent computation in any meaningful way. For example, files did not receive any initial activation that it spread to other files or methods. Similarly, the algorithm spread activation from methods to the containing files, but the files did not, in turn, spread the activation back to other methods. Therefore, we did not discuss these containment relationships as part of the PFIS-V data model and algorithm.

audio, video or graphical images; even others might require user interaction to reveal parts of the output. As a first step towards modeling such diverse outputs we begin by modeling graphical outputs in PFIS-H. Note that, PFIS-H includes capabilities for modeling textual outputs also, just like it models textual changelogs and outputs; however, in this thesis, we deal only with graphical output patches that our study participants foraged in.

The PFIS-H data model represents output patches as nodes in the data model graph. In our study environment, each variant contained an output patch (Figure 5.2), resulting in multiple output nodes in the data model graph. Since all output patches in our study's project contained similar information (e.g., Figure 3.1 (left) and (right) contain the Hextris game interface), the output nodes for the project are connected to each other in the data model graph via "variant-of" links.

To model programmers' foraging in the graphical (non-textual) content in output patches, we explore the appropriateness of a captioning approach. The idea here is that, if we could replace graphical patches with equivalent textual content, then we could model graphical patches similar to other textual patches such as code and changelogs. Note that our aim here is not to explore automated captioning, but to investigate whether, given suitable captions, we could account for graphical patches to improve our computational models of programmers' foraging.

Therefore, we simply replaced the contents in the graphical output patches with descriptions of the task-relevant information features in the patch. We described only the task-relevant features because we hypothesized that a programmer will mostly attend to task-relevant features as cues while foraging in an output patch. For example, score was relevant to the task and so we captioned it, but social media links were not relevant to the task and we did not caption them. We also used the same vocabulary as in the game's domain and the task description; for example, the task description contained the phrase "move the score above the hexagon" and so we used the phrase "score is above the hexagon" rather than the equivalent "score is on top of the hexagon". (Also note the use of the domain-specific "score" rather than a more generic "number").

Using these captions, we included the output patches in the PFIS-H data model as follows. Each output patch translated to a patch node; each node in the caption translated to a word node; the output patch node and word nodes were connected via "patch contains word" links. Since several output patches were similar across variants, we also computed the equivalence of output patches, based on their textual captions: we considered patches with similar captions (e.g., "score is above the multiplier") as equivalent in the eyes of the forager, because they contained similar information value, and provided similar scent about what is in the variant.

5.2 PFIS-H algorithm: Modeling hierarchical foraging

What the PFIS-H algorithm adds to the PFIS-V algorithm is accounting for hierarchies, which it does in the following ways.

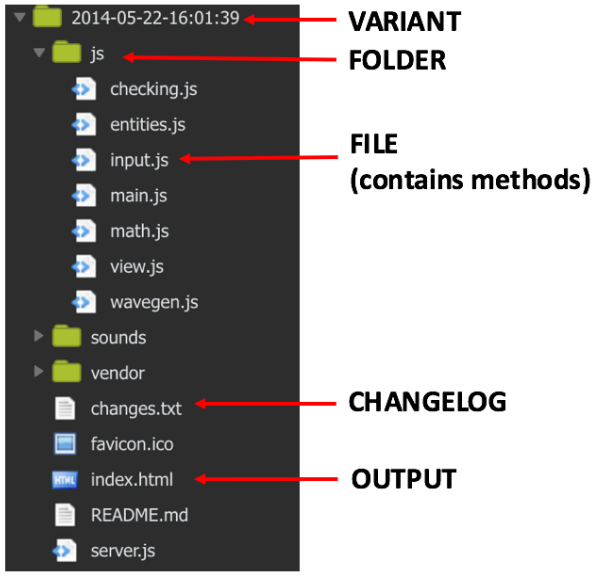

Figure 5.2. A variant's hierarchy. The information environment in our study was organized as patches: variants contained folders, folders contained files, files contained methods.

Extension #1: PFIS-H extends Pirolli's interpretations of the patch construct—that it might be easier for a programmer to forage within a patch than to navigate outside to another patch—to multiple levels of patches in the patch hierarchy. Since, in IFT,

variants are also patches, PFIS-H assumes that a predator is more likely to engage in within-variant foraging rather than to forage across variants—and further, that inside those variants, the predator is likely to forage within the same folder, and within those folders, more likely to forage within the same file and so on.

Thus, PFIS-H first activates the entire hierarchy of patches, namely the variant, the folder, the file and the method (or changelog or output), that the programmer is currently in. Then, in the spreading stage, PFIS-H spreads activation along the "contain" edges going top-down, level-wise. Thus, the variant spreads weights to the folders it contains, the folders then spread activation to the files they contain, and the files in turn spread to methods within it. This way, the algorithm spreads more weights to patches in the same hierarchy that the programmer is currently in (than to patches in other hierarchies).

Extension #2: PFIS-H accounts for additional navigation costs that programmers might perceive when navigating downwards in the hierarchy. For example, in Figure 5.2, some navigations such as scrolling to an adjacent method or clicking on a method call (link) cost 1, a navigation from one file's method to another file's method via the package explorer cost 2 (open file → scroll to method = 2 steps), a navigation to a method in another folder cost 3 (open folder → open file → scroll to method = 3 steps) and a navigation from one variant to another variant's folder's file's method costs 4 (expand variant → expand "js" folder → open file → scroll to method = 4 steps). PFIS-H accounts for participants' perceptions of such variable navigation costs and models participants' perceived costs to match the number of actions it took to make a navigation. (In contrast, PFIS-V approximated all navigation costs to 1, and hence did not explicitly model them.) Both these extensions to the PFIS-H algorithm are summarized in Figure 5.3, marked in blue.

Definitions

Patch set P: set of all patches in the topology seen by a programmer. Patches are hierarchical: a patch contains other smaller patches.
Word set W: set of all words in all patches in P.
Graph $G = (V_p \cup V_w, E_p \cup E_w)$ as defined in Table 2.
Navigation history H: sequence of patches to which the programmer navigated.

To spread activation from node V_1 to node V_2

if V_2 is a patch node **then**
    decay at rate of $\alpha = 0.85$
Value $\leftarrow \alpha \cdot$ weight(N_1) / no. of N_1's neighbors
Cost $\leftarrow$ topological distance between the current patch and N_2
Weight(V_2) $\leftarrow$ Weight(V_2) + (Value / Cost)

To activate navigation history based on recency

Weight $\leftarrow$ 1.0
for each visited patch in H from latest seen to oldest **do**
    activation $\leftarrow$ weight for the path and its containing patch hierarchy
    Weight $\leftarrow$ Weight $\cdot$ 0.9

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

if programmer has not seen exact patch earlier **then**
    if programmer has seen a similar patch P_S **then**
        Assume content of $(k+1)^{th}$ patch to be exactly similar to P_S for source-code or output patches and different from (empty) P_S changelogs
    else
        return "unknown"
Reset Weight for each node in G to 0
Activate navigation history so far in order of recency
Spread activation from patch nodes to work nodes
for each level L of patches in the patch hierarchy (from top to bottom) **do**
    Spread activation from patch nodes at level L to patch nodes at level $\leq L$
Spread activation from word nodes to patch nodes
Rank the patch nodes in the decreasing order of activation
if patch nodes are tied at rank r **then**
    for all t patch nodes **do**
        $rank \leftarrow rank + (t-1)/2$
return the rank for the node representing the $(k+1)^{th}$ patch

Figure 5.3. PFIS-H algorithm. The lines in blue indicate the hierarchical-foraging related extensions.

5.3 PFIS-H Evaluation

We used PFIS-H to predict the 1040 between-patch navigations participants made to methods, changelogs or outputs: these patch types form the leaf nodes in the hierarchy in Figure 5.1 (bottom). We compared the PFIS-H predictions against those by PFIS-V in its variant-and-equivalence-aware(text) configuration: as we saw in the previous chapter, this PFIS-V configuration was the most accurate predictor of participant navigations. In the rest of this chapter, any mention of PFIS-V refers to the variant-and-equivalence-aware(text) configuration, unless specified otherwise.

We compared the ranks of predictions PFIS-V and PFIS-H made for participants' navigations. A repeated measures ANOVA (RM-ANOVA) indicated significant difference ($p=0.0137$) in the mean ranks of PFIS-H and PFIS-V, with PFIS-H (mean=7.04, SD=10.4) resulting in relatively lower (better) ranks than PFIS-V (mean=9.83, SD=15.17) [RM-ANOVA, $F(1, 2990)=4.688$].

Zooming into the predictions for individual participants, Figure 5.4 compares the hit rates of different models for each participant. The height of the bar represents hit rates, i.e., the percentage of the participants' navigations a model predicts within top 10 ranks. As the figure shows, the hit rates from PFIS-H (blue) were better than those of PFIS-V (diagonal stripes) for 6 out of 7 participants (except P01), with PFIS-H's improvements in accuracy as high as 18.48% (P03).

Both these results, namely PFIS-H's significantly lower ranks and higher hit rates than PFIS-V, suggest that PFIS-H was a more accurate model of participants' foraging than PFIS-V.

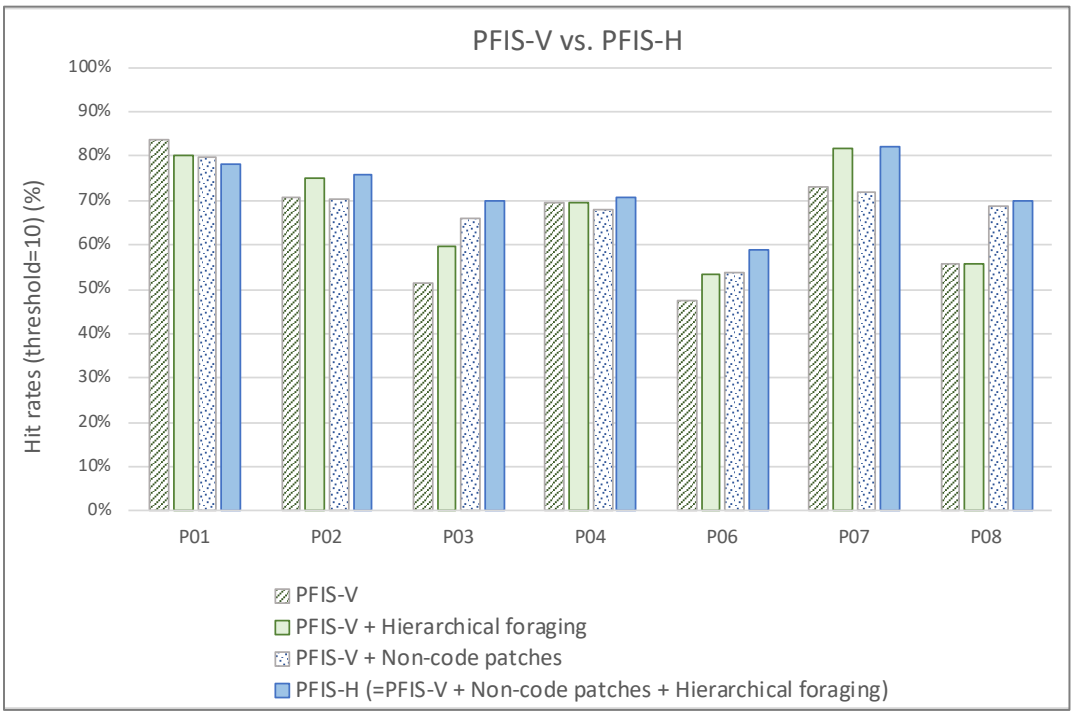

Figure 5.4. PFIS-V vs. PFIS-H hit rates. PFIS-H resulted in higher hit rates than PFIS-V for all participants except P01. The fact that PFIS-H hit rates were higher than either of PFIS-V_{NC} or PFIS-V_{H} suggests that PFIS-H's predictive advantages came from a combination of the two improvements, namely modeling non-code patches and hierarchical foraging.

5.4 Where did PFIS-H improvements come from?

PFIS-H makes two changes over PFIS-V: 1) modeling, and predicting navigations to, non-code patches and 2) modeling hierarchical foraging:

$$PFIS\text{-}H = PFIS\text{-}V + non\text{-}code\ patches + hierarchical\ foraging.$$

To investigate how each of these changes translated to PFIS-H's predictive advantages, we isolated the two factors in two distinct extensions to PFIS-V:

$$PFIS\text{-}V_{NC} = PFIS\text{-}V + Non\text{-}code\ patches,\ and$$

$$PFIS\text{-}V_{H} = PFIS\text{-}V + Hierarchical\ foraging$$

As Figure 5.4 shows, for all participants (except P01), PFIS-H was more accurate than either of these individual models, suggesting that PFIS-H's improvements came from modeling non-code patches as well as from modeling hierarchical foraging. In the

rest of this section, we discuss each the individual improvements from each of these factors.

5.4.1 Improvement #1: Modeling non-code patches

Modeling non-code patches resulted in better hit rates for individual participants' navigation predictions. As Figure 5.4 shows, PFIS-V_{NC} resulted in higher hit rates than PFIS-V for 3 out of 7 participants (P03, P06, P08), with the improvements being as high as 14.39%. For the other 4 participants, in which PFIS-V_{NC} was not higher than PFIS-V, the differences did not exceed 3.81% (P1). In fact, a comparison of ranks revealed that, on an average, PFIS-V_{NC} (mean=9.83, SD=15.17) made significantly more accurate predictions than PFIS-V (mean=8.65, S.D.=14.23) [RM-ANOVA, F(1, 2990)=4.688, p=0.0137].

Participant	No. of navigations			
	Method	Output	Changelog	Total
P01	60	40	14	114
P02	119	29	10	158
P03	150	69	1	220
P04	96	52	23	171
P06	105	29	2	136
P07	109	26	0	135
P08	79	27	0	106
Total	718	272	50	1040
	(69.04%)	(26.15%)	(4.81%)	(100.00%)

Table 5.1. Study-1 participant navigations: different patch types. Participants navigated to non-code patches over 30% of the time: whereas PFIS-V failed to account for these navigations, PFIS-H was able to predict them.

These improvements in PFIS-V_{NC}'s accuracy partly came from its ability to predict more navigations than PFIS-V. As Table 5.1 shows, participants navigated to changelogs and outputs about 30% of the time. Whereas PFIS-V only predicted method-to-method navigations (N=665), PFIS-V_{NC} filled this gap and predicted navigations to (and from) changelogs and outputs also (N=1040).

Not only did PFIS-V_{NC} predict more navigations than PFIS-V, it retained the accuracy of the predictions for the navigations that PFIS-V already predicted: the individual predictions of PFIS-V_{NC} were also about the same as PFIS-V for navigations that both models predicted.

As an example of how these improvements played out in individual participants' predictions, see Figure 5.5 comparing the predictions made by PFIS-V_{NC} and PFIS-V for P01's navigations. P01 is an example of the worst case, where PFIS-V_{NC}'s (and PFIS-H's) hit rate was worse than that of PFIS-V. The x-axis in the graph indicates predictions and the y-axis indicates the rank of the prediction (light blue = PFIS-V, dark blue = PFIS-V_{NC}). If a model fails to make a prediction, then the graph shows a corresponding diamond above the graph.

As the figure demonstrates, whenever participants navigated to methods and PFIS-V and PFIS-V_{NC} made a prediction, the ranks were mostly similar as the overlapping light blue "+" and the dark blue dots show. An RM-ANOVA indicated no significant difference in ranks between PFIS-V_{NC} (mean=10.28, SD=15.8) and PFIS-V (mean=9.83, SD=15.17), [RM-ANOVA, F(1, 1185)=0.588, p=.472]) for these navigations to methods.

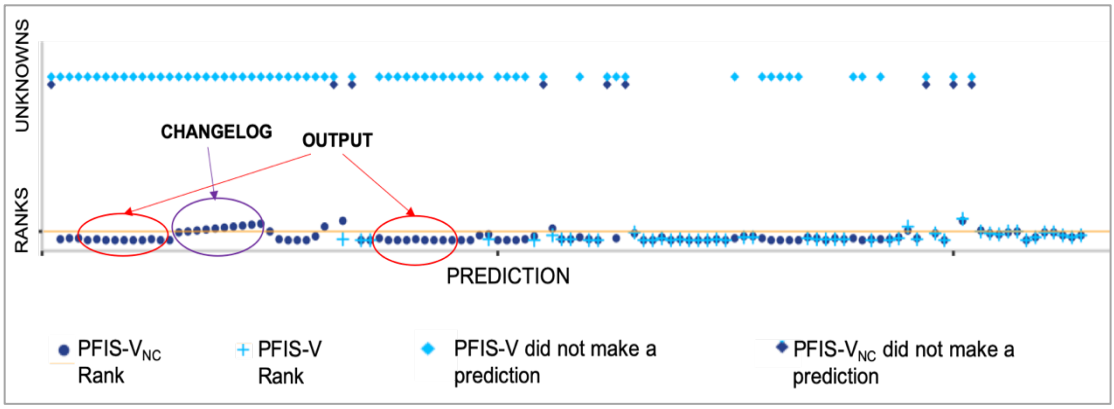

Figure 5.5. P01's navigation predictions (x-axis=predictions, light blue=PFIS-V, dark blue = PFIS-V_{NC}). For navigations to non-code patches, PFIS-V failed to make a prediction (top, light blue diamonds) whereas PFIS-V_{NC} (bottom, dark blue dots) predicted them at lower ranks. For other navigations, PFIS-V_{NC} predictions (bottom, dark blue dots) were mostly similar, or slightly worse than that of PFIS-V (bottom, light blue "+").

For other navigations, namely to changelogs and outputs, PFIS-V failed to make prediction (light blue "diamond" above the navigation), whereas PFIS-V_{NC} predicted them at low ranks (generally low dark blue dots).

This accuracy of PFIS-V_{NC} when predicting changelog and output navigations suggest the validity of our hypotheses about participants' foraging in these patches, namely that they estimated the contents of new changelog (or output) patch based on what they had seen in other changelogs (or outputs) and that they mostly attended to task-relevant information features in output patches.

5.4.2 Improvement #2: Modeling hierarchical foraging

PFIS-H also made significant improvements in accuracy from its modeling of hierarchical foraging. We measured these improvements in two ways.

First, we isolated the improvements from hierarchical foraging by comparing PFIS-V_{NC} and PFIS-H, both of which model code and non-code patches. This comparison resulted in PFIS-H's ranks (mean=7.03, SD=10.4) being significantly lower than that of PFIS-V_{NC} (mean=8.65, SD=14.23) [RM-ANOVA $F(1, 1860)=12.25$, $p=0.0128$].

Second, we compared PFIS-V and PFIS-V_H both of which predict method-to-method navigations only. In this case, PFIS-V_H (mean=7.82, SD=10.52) ranks were significantly lower than those of PFIS-V (mean=9.82, SD=15.15) [RM-ANOVA $F(1,1134)=10.56$, $p=0.0175$]. Thus, in both cases, modeling hierarchical foraging led to significantly better predictions of participants' navigations.

These comparisons also held at the individual participant level. As Figure 5.4 shows, for between-method navigations, PFIS-V_H resulted in similar or higher hit rates than PFIS-V for 6 out of 7 participants (except P01). Similarly, with the non-code patches accounted for, PFIS-H was more accurate than PFIS-V_{NC} for 6 out of 7 participants (except P01).

Finally, drilling down into the predictions for individual navigations, Figure 5.6 illustrates the effects of modeling hierarchical foraging one navigation at a time. The x-axis indicates navigations and y-axis indicates difference between PFIS-V_{NC} and PFIS-H ranks. A positive bar represents instances where modeling hierarchical foraging helped PFIS-H's predictions, a negative bar indicates instances where

modeling hierarchical foraging hurt PFIS-H's predictions. The height of the bars indicates the extent to which hierarchical foraging helped or hurt that navigation's prediction.

As the mostly taller and more frequent positive bars in Figure 5.6 indicate, modeling hierarchical foraging helped, more than it hurt, PFIS-H's ability to predict programmer navigations.

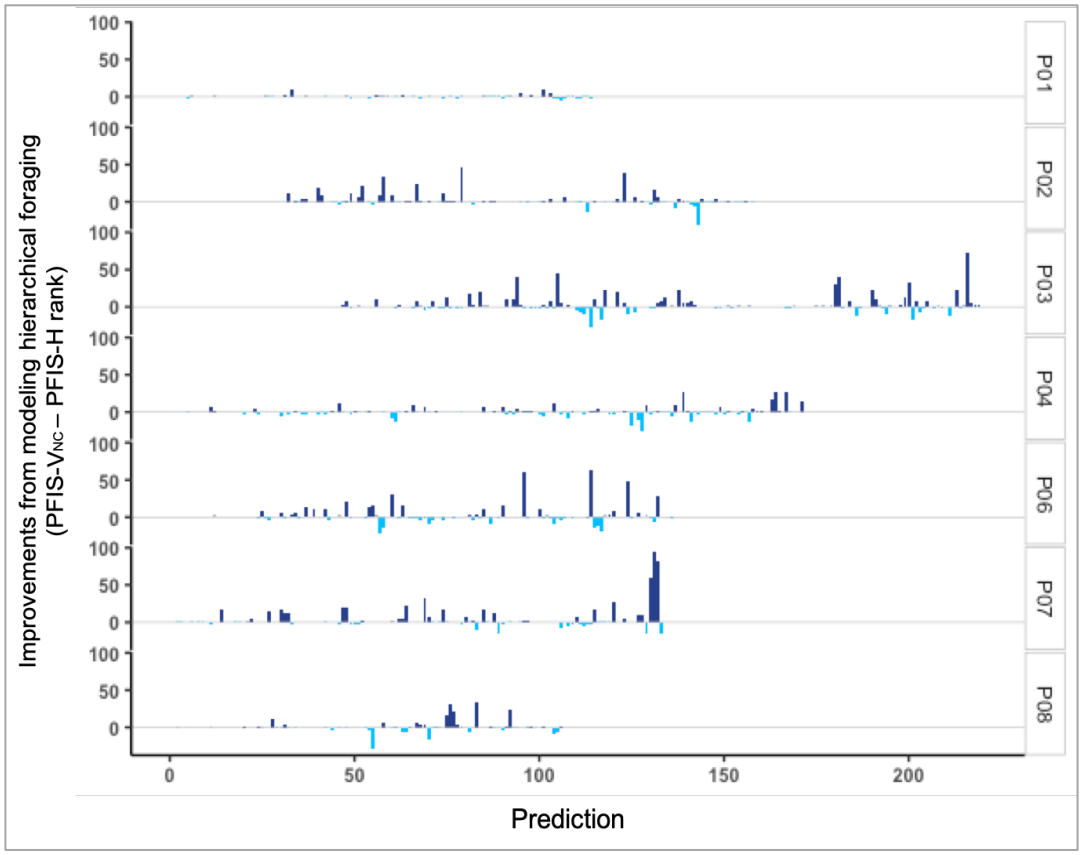

Figure 5.6. Improvements from hierarchical foraging. (Positive-bar = PFIS-H was better than PFIS-V$_{NC}$, negative bar= PFIS-V$_{NC}$ was better than PFIS-H.) The taller and more frequent positive bars (pointing upwards) indicate that PFIS-H made more accurate predictions than PFIS-V$_{NC}$.

Interpretation: Hierarchical foraging from a variations foraging standpoint

Modeling hierarchical foraging, namely that participants will forage within a patch (or file or folder or variant) rather than between them, turned out to be advantageous for PFIS-H when predicting within-variant navigations. Notably, PFIS-H made more

accurate predictions than PFIS-V_{NC} when participants' foraging within a variant stopped following prior scent and began following new scent (e.g., move from looking for the word "score" to looking for "multiplier") as in the following scenarios.

Within-variant scenario #1. Changing information goals. PFIS-H had an advantage when predicting participants' within-variant navigations when their foraging goals changed (e.g., from looking for score, to looking for multiplier) and they followed new scent pertaining to their new goal. For example, consider P08's foraging in the destination variant, as shown in Figure 5.7. In region 1A of the graph, P08 made several navigations based on the same word "score", and both PFIS-V_{NC} (light blue) and PFIS-H (dark blue) were able to accurately predict such same-scent navigations (low, mostly overlapping dots). In contrast, 1B is an instance where P08 changed goals to look for multiplier code and started searching for words such as "hiding", "text" and "random". Since these navigations were not made on what the participant had already seen, or the scent she was following so far, the scent computation apparatus in both PFIS-H and PFIS-V_{NC} made inaccurate predictions, as the peaks (high ranks) in the graph show at 1B. However, PFIS-H (dark blue) resulted in a better rank than PFIS-V_{NC} (light blue), because it was able to guess that the participant will navigate to some location within the same variant. In contrast, PFIS-V made no such assumption.

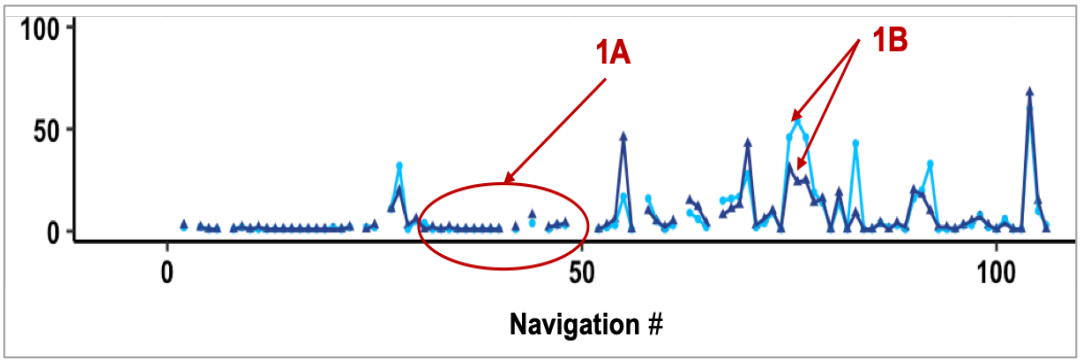

Figure 5.7. P08's navigation predictions (light blue = PFIS-V_{NC}, dark blue=PFIS-H). When P08 made within-variant navigations following the trail of the word "score" (1A), PFIS-V_{NC} and PFIS-H resulted in mostly similar ranks, but when he followed new scent, PFIS-H resulted in better ranks than PFIS-V_{NC} (1B).

Within-variant scenario #2: Navigations to other patch types. Another scenario where modeling hierarchical foraging helped was when participants navigated to different patch types within the same variant (e.g., from code to changelog). Here again, PFIS-H made better predictions than PFIS-V_{NC}, as in the previous scenario. As an example, consider the case of P06: when foraging in the destination variant's source-code patches, P06 realized that the variant did not work as he expected. Therefore, at 2A in Figure 5.8, he navigated from the variant's code to the variant's changelog—a different patch type. Both PFIS-V_{NC} and PFIS-H mispredicted this navigation, since the changelog was not linked from the current method, nor based on recently visited patches, nor based on words the participant had demonstrated interest in, in the source code patches (light and dark blue peaks at 2A). However, as in scenario #1, the PFIS-H rank (dark blue) was lower (better) than that of PFIS-V_{NC} (light blue) because PFIS-H accurately guessed that the participant will remain within the same variant, whereas PFIS-V_{NC} had no such clue about where and why a programmer will navigate. We also observed similar instances when participants changed patch types to navigated to/from output patches to other patch types within the same variant.

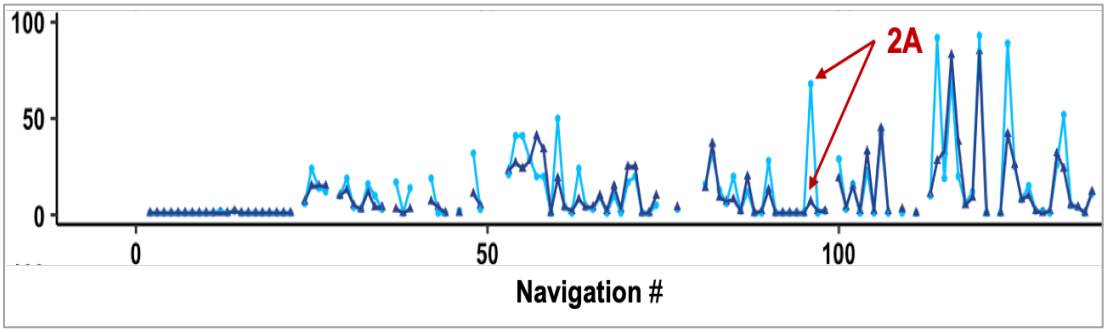

Figure 5.8. P06's navigation predictions (light blue = PFIS-V_{NC}, dark blue=PFIS-H). PFIS-H predicted navigations to different patch types within the same variant better than PFIS-V_{NC} did. At 2A, P06 navigated from the source-code patches to the changelog patch within the source variant.

Unfortunately, participants *did not* always forage within the same variant; they made several navigations across variants, such as to find a suitable source variant or to integrate code from the source variant into the destination variant. In such between-

variant cases, PFIS-H sometimes made *worse* predictions than PFIS-V_{NC} and at other times it made similar predictions as PFIS-V_{NC}, as the following scenarios show.

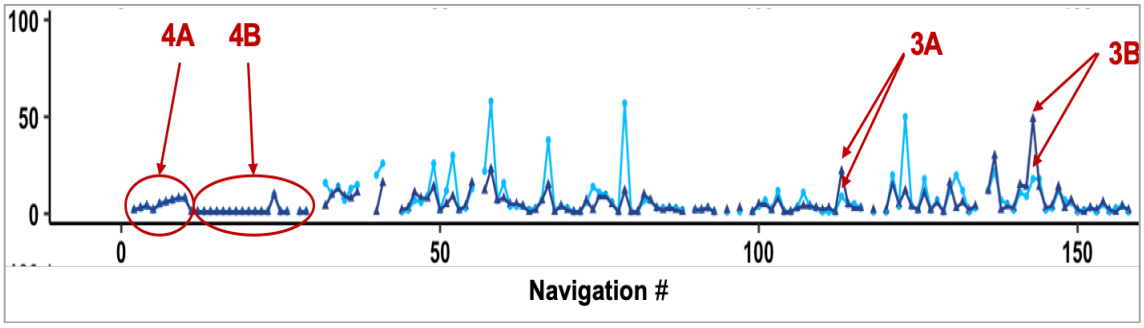

Figure 5.9. P02's Navigations predictions. (dark blue = PFIS-H, light blue = PFIS-V_{NC}). For between-variant navigations, PFIS-H made worse predictions than PFIS-V_{NC} when participants navigated to methods (high cost, higher PFIS-H ranks than PFIS-V_{NC} at 3A, 3B) and similar predictions as PFIS-V_{NC} for navigations to changelogs and outputs (low cost, overlapping light and dark blue dots in 4A and 4B).

Between-variant scenario #1. Navigations to methods. One scenario where PFIS-H made worse predictions than PFIS-V_{NC} was when participants navigated from one variant's source-code to another variant's source code: these navigations contribute to some of the negative bars in Figure 5.6. For instance, see Figure 5.9: at 3A, P2 navigated from 2014-05-21-15:39:02 variant's *renderText*() to Current variant's *checkGameOver*() and at 3B, he navigated from Current variant's *drawPolgyon*() to 2014-05-21-15:39:02 variant's *renderText*(). For these between-variant navigations, PFIS-H (dark blue), that expects participants to forage within variants than across, resulted in worse predictions than PFIS-V_{NC} (light blue).

Between-variant scenario #2. Navigations to changelogs/outputs. In contrast, PFIS-H resulted in similar ranks as PFIS-V_{NC}, when participants navigated between variants, navigating from one variant's patches (methods or changelogs or outputs) to another variant's changelogs or outputs (open variant → open changes.txt or run index.html). For example, in Figure 5.9, P02 navigated to from one variant's change log to another variant's changelog at 4A and from one variant's output to a new variant's output at

4B: for both these navigations, PFIS-H (dark blue) and PFIS-V_{NC} (light blue) ranks almost overlap.

This is because, PFIS-H not only expected participants to forage within a variant, it also expected them to navigate in ways that will maximize value/cost. In this case, PFIS-H modeled that between-variant navigations from changelog/output to another changelog/output can be cheaper (expand variant folder → open changes.txt or run index.html: cost=2) than within-variant navigations from changelogs/outputs to methods (js folder → view.js → render() = 3) and that participants are likely to navigate to changelogs/output over methods across variants. Thus, modeling cost offset some of the potential disadvantages at blindly favoring within-variant navigations over between-variant navigations in PFIS-H.

In summary, PFIS-H's predictive improvements came from modeling both non-code patches as well as from modeling hierarchical foraging—in the latter case, both from favoring within-patch navigations over between-patch ones, as well from accounting for the navigation costs.

5.4 Does hierarchical foraging generalize beyond variants?

As we described earlier, hierarchical foraging has its underpinnings in IFT's notions of between-patch vs. within-patch foraging—that a forager will forage within a patch than across them—and the costs of those foraging navigations. Therefore, we have no reason to believe that hierarchical foraging, as a phenomenon, applies only to variations foraging. Specifically, the question arises whether hierarchical foraging also applies to programmers' foraging in a single variant of a program.

To answer this question, we obtained the navigation data from a prior study by Piorkowski et al. that did not involve variants [64]. Participants in the study, namely 9 professional programmers, worked in Eclipse IDE to fix a bug in jEdit, an open-source Java-based project. The jEdit program was hierarchically organized into packages and subpackages, classes and methods.

We predicted participants' method-to-method navigations in jEdit using PFIS-V and PFIS-V_H. A comparison of prediction ranks from the two models indicated significant differences [RM-ANOVA $F(1,930)=5.919$, $p=.0378$]—with PFIS-V_H

(mean=24.53, S.D.= 63.7) resulting in significantly lower ranks (better predictions) than PFIS-V (mean=33.54, S.D.=94.42) and PFIS-V_H). Further, when predicting individual participant navigations, PFIS-V_H resulted in up to 10% (P6) higher hit rates than PFIS-V.

These results lend evidence supporting the hypothesis that led to the development of PFIS-H—that programmers adapt their foraging to the hierarchical organization of the environment and that they account for the costs of doing so. These results also suggest the generality of hierarchical foraging, as phenomenon, to professional programmers, and to non-variations situations. In the next chapter, we'll continue along these lines and further evaluate the generality of our results and models.

CHAPTER 6. GENERALIZATION: EVALUATION WITH NEW DATA

So far, we have been exploring the question: *does IFT explain and predict information seeking in the presence of variants?* In Chapters 3-5, we considered this question empirically for novice programmers, with some attempt at generalizing our results on hierarchical foraging to single-variant situations and to more experienced programmers. In this chapter, we continue in the direction of generalization, considering whether our results generalize to more experienced programmers, in variations foraging situations.

6.1 Methodology

Towards this end, we conducted a replication of the investigation into novice programmers' variations foraging in Chapter 3-5, but with a new population, namely experienced programmers.

Because our goal was to investigate whether our results generalize to another population, we kept all variables unchanged except the participants' level of experience. Thus, we used the same programming environment[9] (Cloud9), the same game program (Hextris), similar data collection apparatus and the same tutorials and tasks as in the previous study. We also replicated all other experience-related variables except participants' overall years of experience: in both studies, we did not explicitly include or exclude participants with Javascript experience, familiarity with the Hextris codebase or the Cloud9 environment. This way, we attempted to eliminate as many sources of uncontrolled variations that might affect the results.

We recruited 10 experienced programmers from our graduate CS program. As Table 6.1 shows, the experienced programmer population carried greater programming experience (mean=10.9 years, median=10 years) than the novice programmers that

[9] In our second study, we used the latest versions of the Cloud9 IDE, operating systems and web browsers available at the time of conducting the study. However, we did not observe any changes (e.g., navigation affordances, navigation costs) in the newer versions that might lead to different foraging behaviors than the previous study.

participated in our prior study (Mean= 4.06 years, median=4 years). See page 21, Table 3.1 for the demographics of the original study participants.

Participant	Gender	Age	Overall programming experience	Javascript experience	Ever built a web / mobile app with JS?	Cloud9 experience
S01	Male	20s	4	0	No	No
S02	Male	20s	6	0.5	Website similar to amazon as course project.	No
S03	Male	30s	17	0	No	No
S04	Male	30s	15	3	Yes, as a demo for students as well as for research tools.	Yes, briefly just playing with it.
S05	Male	20s	6	<1	No	No
S06	Male	40s	20	0	No	No
S07	Male	20s	14	0	No	No
S08	Male	30s	>10	3	Android app using Java/Javascript; Desktop client app using Electron/Javascript.	No
S09	Female	20s	7	<1	Yes. 3 web apps with React and Nodejs.	No
S10	Female	30s	10	0	Yes, with help from someone, I built a JS web app [sic].	No

Table 6.1. Replication study: Participant demographics.

6.2 Research questions

We evaluated the generality our prior empirical results via the evaluation of our two computational models, by way of answering the following questions:

- *RQ1: PFIS-V generalization.* Does PFIS-V model more experienced variations foraging behaviors as well as it modeled novice programmers' variations foraging?

- *RQ2: PFIS-H generalization.* Does PFIS-H predict the hierarchical foraging behaviors of more experienced programmers (including their navigations to non-code patches) as well as it did for novice programmers?

To answer these questions, we re-ran the PFIS-V and PFIS-H algorithms on the new participants' navigation data. Together, the 10 participants made over 1500 between-patch navigations, navigating to both source-code and non-code patches.

6.3 Results: PFIS-V generalization (RQ1)

To evaluate the generality of PFIS-V, we conducted the same analyses with the new study's data, as we had done with the original data: 1) comparison of PFIS-V and PFIS3 predictive accuracies, 2) comparison of the four data models, 3) comparison of the two groups of participants. As we discuss these results, we also juxtapose graphs/tables with those from the original study, to facilitate easy comparison.

6.3.1 PFIS-V vs. PFIS3

We compared the predictiveness of PFIS-V and PFIS3 in terms of both unknown rates and hit rates.

Unknown rates: Figure 6.1 (left) compares the unknown rates from PFIS-V and PFIS3; the corresponding graph from the prior study is reproduced in Figure 6.1 (right). Recall that unknown rates refer to the percentage of navigations a model failed to make a prediction for, namely when the participant navigated to a location s/he (and the model) did not know existed.

As Figure 6.1 (left) shows, for all participants except S02, S09, PFIS-V ended up with fewer unknowns than PFIS3. For S02 and S09, there was no advantage (and no disadvantage) of PFIS-V vs. PFIS3 because: 1) S02 foraged predominantly in a single variant and 2) S09 predominantly navigated based on word-similarity and method call relationships, navigating less frequently to newer locations.

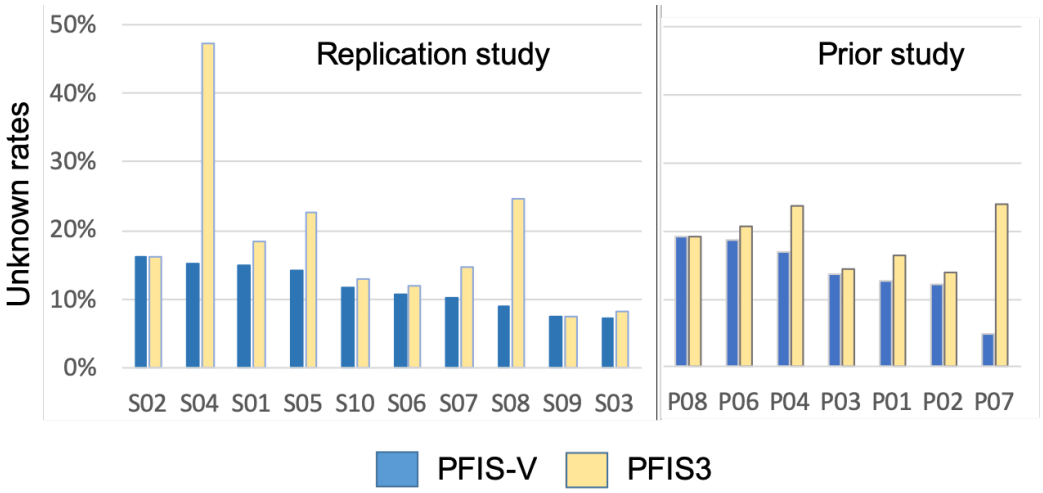

Figure 6.1. PFIS-V generalization: unknown rates (ordered by PFIS-V unknown rates). PFIS-V resulted in similar or lower unknown rates than PFIS3 for all participants in the replication study (left), just as it did in the original study (right).

Hit rates: In terms of the accuracy of the predictions, as Figure 6.2 (left) shows, PFIS-V, on an average, was more accurate than PFIS3 in all data model configurations. Considering predictions for individual participant navigations, Table 6.2 (right) shows that PFIS-V was more accurate than PFIS3 for all participants except S02 and S09. For S09, PFIS-V still had a predictive advantage over PFIS3 at higher rank thresholds, but for S02, who foraged within a single variant, PFIS-V was about the same as PFIS3.

Both the above results, namely the generally lower unknown rates and the overall higher hit rates from PFIS-V than PFIS3, are consistent with our original study's findings, suggesting the generality of the assumptions encoded in the PFIS-V algorithm.

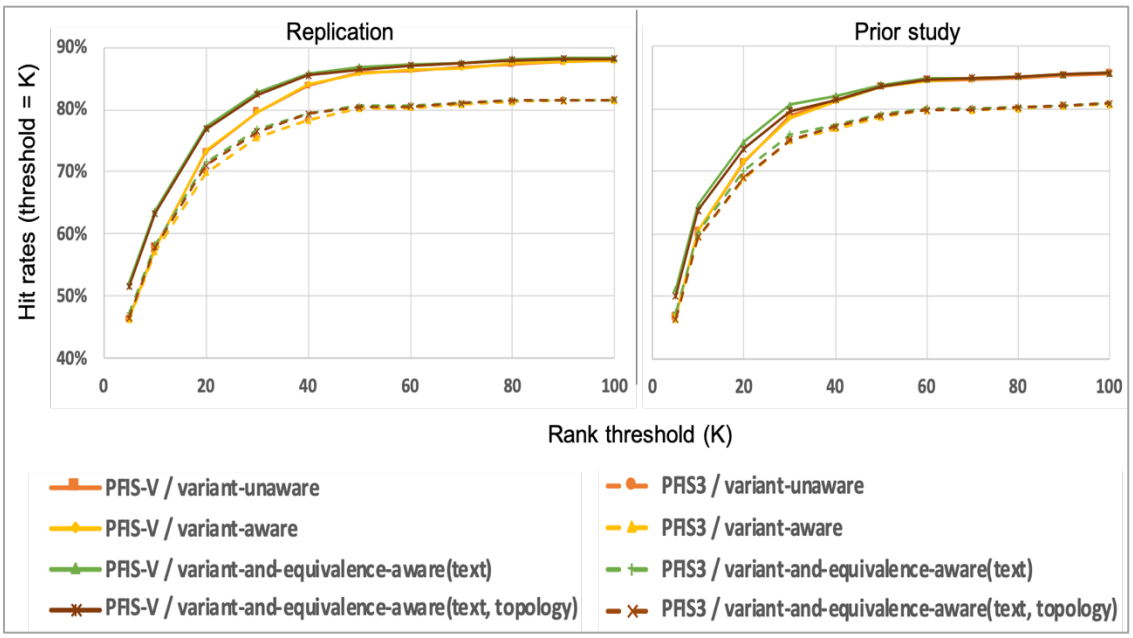

Figure 6.2. PFIS-V generalization: hit rates. PFIS-V (solid lines) resulted in higher hit rates than PFIS3 (dotted lines) in all four data model configurations in both the studies.

6.3.2 PFIS-V: Which data model is most accurate?

In terms of the data model configurations also, the results from the original study generalized to our new study. As Table 6.2 (Replication study) shows, the predictiveness of the four data models followed the order: variant-unaware ≤ variant-aware ≤ variant-and-equivalence-aware (text, topology) ≤ variant-and-equivalence-aware(text), suggesting that our assumptions about participants' mental models (e.g., similarity of patches, the notion of equivalence) generalized to experienced programmers.

Replication study

Participant	No. of variants	No. of between-method navigations (N=1023)	Per participant hit rate (rank threshold=10) orange=PFIS3, black=PFIS-V			
			Variant-unaware	Variant-aware	Variant-and-equivalence-aware(text)	Variant-and-equivalence-aware(text, topology)
GROUP-1						
S04	37	125	21.60%	20.00%	58.40%	53.60%
			21.60%	21.60%	31.20%	26.40%
S05	8	106	47.17%	47.17%	52.83%	52.83%
			47.17%	47.17%	47.17%	47.17%
S06	4	84	70.24%	70.24%	71.43%	71.43%
			70.24%	70.24%	70.24%	70.24%
S07	5	88	57.95%	57.95%	60.23%	60.23%
			55.68%	55.68%	55.68%	55.68%
S08	17	122	67.21%	67.21%	75.41%	75.41%
			63.11%	63.11%	63.11%	63.11%
S10	3	147	57.82%	57.82%	58.50%	58.50%
			57.14%	57.14%	57.14%	57.14%
GROUP-2						
S01	2	88	35.63%	35.63%	39.08%	39.08%
			35.63%	35.63%	37.93%	37.93%
S02	1	31	70.97%	70.97%	70.97%	70.97%
			70.97%	70.97%	70.97%	70.97%
S03	2	97	78.35%	78.35%	79.38%	79.38%
			78.35%	78.35%	78.35%	78.35%
S09	2	135	71.85%	71.85%	71.85%	71.85%
			71.85%	71.85%	71.85%	71.85%

Original study

Participant	No. of variants	No. of between-method navigations (total=665)	Per participant hit rate (rank threshold = 10) black = PFIS-V; orange = PFIS3			
			variant-unaware	variant-aware	variant-and-equivalence-aware(text)	variant-and-equivalence-aware(text, topology)
GROUP-1						
P04	5	89	65.17%	65.17%	69.66%	69.66%
			64.04%	64.04%	64.04%	64.04%
P07	21	104	49.04%	49.04%	73.08%	67.31%
			48.08%	48.08%	53.85%	48.08%
GROUP-2						
P01	3*	55	83.64%	83.64%	83.64%	83.64%
			80.00%	80.00%	80.00%	80.00%
P02	2	116	70.69%	70.69%	70.69%	70.69%
			69.83%	69.83%	69.83%	69.83%
P03	2	132	51.52%	51.52%	51.52%	51.52%
			51.52%	51.52%	51.52%	51.52%
P06	2	101	47.52%	47.52%	47.52%	47.52%
			46.53%	46.53%	46.53%	46.53%
P08	2	68	55.88%	55.88%	55.88%	55.88%
			55.88%	55.88%	55.88%	55.88%

* P01 made an additional navigation a third (potential source) variant.

Table 6.2. PFIS-V generalization: per participant hit rates. (orange=PFIS3, black=PFIS-V). For most participants in both studies, PFIS-V yielded higher hit rates than PFIS3. For other participants, PFIS-V hit rate was similar to that of PFIS3.

6.3.3 PFIS-V: Two groups and two foraging behaviors

But we saw a difference from the original study in terms of the predictive advantage from modeling equivalence. Recall that participants fell into two groups based on their between-variant foraging behaviors: whereas Group-1 participants navigated to code and non-code patches in multiple variants, Group-2 participants navigated exclusively to non-code patches in several variants, looking at source code only in the source and destination variants. Since the source code in consecutive variants were very similar, modeling equivalence of patches led to improved PFIS-V predictions for Group-1 participants in the original study (Table 6.2: prior study), but no such improvements were observed for Group-2 participants.

However, in the replication study (Table 6.2: replication study), modeling equivalence led to slight improvements for Group-2 participants also. As 2 out of 4 Group-2 replication participants (S01, S03) foraged in the source and destination variants, they encountered source-code patches (e.g., game analytics-related code) that happened to be identical between the two variants, leading to better predictions by the equivalence-aware models.

6.4 Results: PFIS-H Generalization (RQ2)

To answer RQ2 on the generality of PFIS-H, we first compared the accuracies of PFIS-H and PFIS-V. We then delved down into the individual improvements from modeling non-code patches and hierarchical foraging, as we did in Chapter 5.

6.4.1 PFIS-H vs. PFIS-V

Consistent with the results from the previous study, PFIS-H was better than PFIS-V at predicting participants' navigations in the new study. A comparison of the prediction ranks from the two models indicated significant differences [RM-ANOVA $F(1, 2285)=21.22$, $p=0.00128$] with PFIS-H ranks (mean=6.08, SD=10.32) being significantly lower and hence better than PFIS-V's (mean=9.15, SD=13.64). In fact, as Figure 6.3 (left) shows, PFIS-H yielded better hit rates than PFIS-V for every one of the participants.

In the rest of this section, we tease apart the predictive gains in PFIS-H from its two distinct improvements, namely modeling non-code patches and modeling hierarchical

foraging. Indeed, as Figure 6.3 (left) shows, both these changes contributed to PFIS-H's improvements: PFIS-H which modeled both non-code patches and hierarchical foraging resulted in better hit rates than PFIS-V_{NC} which modeled only non-code patches, or PFIS-V_H which modeled only hierarchical foraging. (As in the previous study, PFIS-V_{NC} = PFIS-V + non-code patches, PFIS-V_H = PFIS-V + hierarchical foraging; PFIS-H = PFIS-V + non-code patches + hierarchical foraging.)

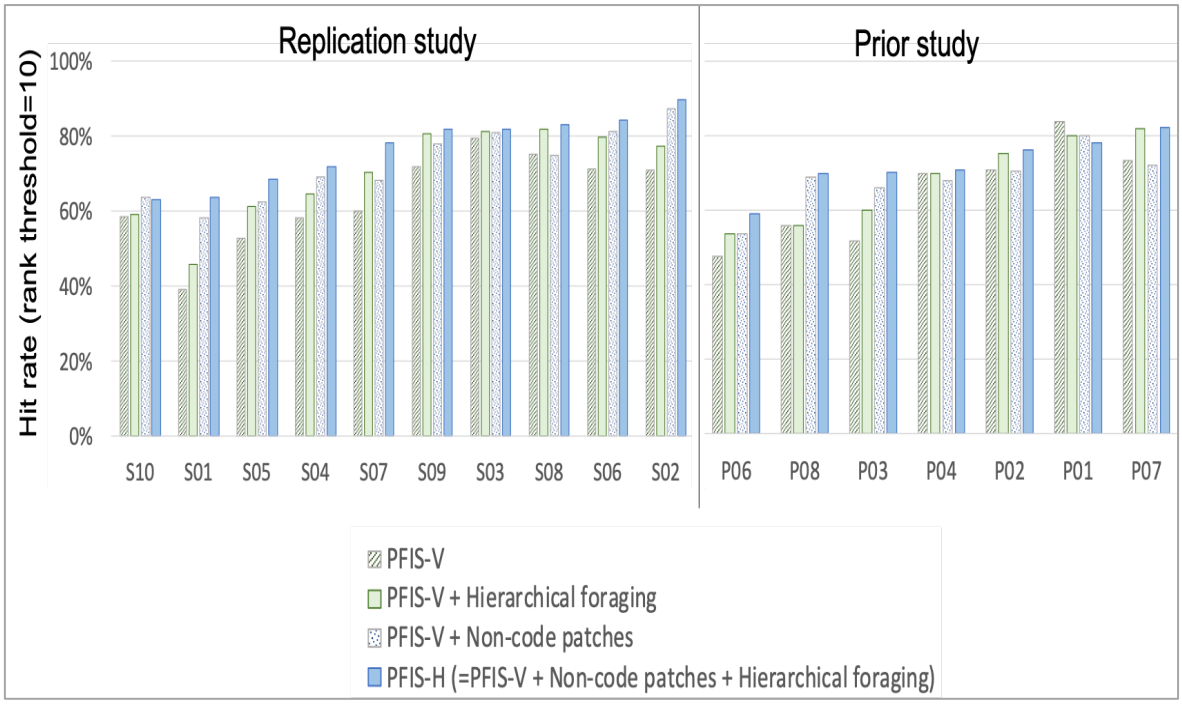

Figure 6.3. PFIS-H generalization (ordered by PFIS-H hit rates). In both the replication and the original studies, modeling hierarchical foraging (PFIS-V vs. PFIS-V_H) as well as non-code patches (PFIS-V vs. PFIS-V_{NC}) led to higher predictive accuracy, resulting in higher accuracy of PFIS-H than PFIS-V.

6.4.2 Improvement #1: Foraging in non-code patches.

Similar to novice programmers (in the original study), more experienced programmers (in the replication study) also foraged among changelogs and outputs, navigating to them ~23% of the time (Table 6.3: left). Whereas PFIS-V failed to account for these navigations, which constituted close to one-fourth of all participant navigations, PFIS-H (and PFIS-V_{NC}) which accounted for non-code patches was able to predict these navigations.

Further, PFIS-V_{NC} (and PFIS-H) were able to also predict these navigation accurately. In fact, PFIS-V_{NC}'s ranks (mean=7.41, SD=12.57) were significantly lower than those of PFIS-V (mean=9.15, SD=13.64) [RM-ANOVA F(1, 2285) = 12.49, p=0.00638]. These improvements in ranks also reflected in hit rates: as Figure 6.3 (left) shows, PFIS-V_{NC} yielded higher hit rates than PFIS-V for all participants but one (S08). These improvements in accuracy are similar to what we observed in the previous study, suggesting that our earlier assumptions about how novice programmers foraged in non-code patches (that we modeled in PFIS-V_{NC}) also generalize to experienced programmers.

REPLICATION STUDY					**PRIOR STUDY**				
Participant	No. of navigations				Participant	No. of navigations			
(N=10)	Method	Output	Changelog	Total	(N=7)	Method	Output	Changelog	Total
S01	112	32	0	144	P01	60	40	14	114
S02	54	25	0	79	P02	119	29	10	158
S03	112	26	0	138	P03	150	69	1	220
S04	135	57	0	192	P04	96	52	23	171
S05	122	22	0	144	P06	105	29	2	136
S06	98	57	0	155	P07	109	26	0	135
S07	96	24	0	120	P08	79	27	0	106
S08	125	19	4	148					
S09	156	70	0	226					
S10	160	19	0	179					
Total	1170	351	4	1525	Total	718	272	50	1040
	(76.72%)	(23.02%)	(0.26%)	(100.00%)		(69.04%)	(26.15%)	(4.81%)	(100.00%)

Table 6.3. PFIS-H generalization: navigations to non-code patches. (left=replication, right=original study). About one-fourth of all participants' navigations were to non-code patches; however, experienced programmers (left) made fewer navigations to changelogs than the novice programmers did (right).

However, as Table 6.3 left vs. right shows, one notable difference between the two populations is that the experienced programmers (left) made fewer navigations (mean=0.4, SD=1.26) to *changelog* patches than novice programmers (right) did (mean=7.14, SD=8.88) [Welch's t-test, t(6.1709)=-1.9954, p=0.09169]. One possible reason is that experienced programmers (who were likely to be familiar with version control) did not expect to find changelog information in our study environment that was very different from traditional version control environments. Another possibility is that experienced programmers were aware that changelogs were about what changed-

-and not what is contained--in a variant and therefore did not expect to gain any valuable information from them (and hence either ignored them or did not look for them). As we speculated earlier in Chapter 3, this might also be the reason that novice programmers abandoned changelog patches in favor of the more valuable outputs.

6.4.3 Improvements #2: Hierarchical foraging

The second set of PFIS-H improvements came from modeling hierarchical foraging, which led to significantly better predictions (in terms of ranks) when predicting method-to-method navigations as well as when predicting navigations to code and non-code patches (PFIS-V_{NC} vs. PFIS-H). Table 6.4 shows the results of the statistical tests in the two cases.

Considering only between-method navigations, without accounting for non-code patches (N=1022)	**With accounting for navigations to code and non-code patches (N=1525)**
PFIS-V vs. PFIS-V_H	PFIS-V_{NC} vs. PFIS-H
Significant differences between PFIS-V and. PFIS-V_H ranks: RM-ANOVA F(1, 1794) = 20.48, p=0.00144].	Significant differences between PFIS-V_{NC} and. PFIS-H ranks [RM-ANOVA, F(1, 2776)=16.46, p=0.00286].
PFIS-V_H ranks: mean=7.09, SD=10.21 PFIS-V ranks: mean=9.15, SD=13.64	PFIS-H ranks : mean=6.08, SD=10.32 PFIS-V_{NC} ranks : mean=7.41, SD=12.57

Table 6.4. PFIS-H generalization: Hierarchical foraging improvements. Irrespective of whether non-code patches were included or not, modeling hierarchical foraging led to significant improvements in predictions for experienced programmers' navigations.

These significant differences in prediction ranks also translated to higher hit rates when predicting individual participant navigations. In Figure 6.3s (left), for almost all participants PFIS-V_H > PFIS-V and PFIS-H > PFIS-V_{NC}, suggesting that modeling hierarchical foraging led to significantly better predictions of more experienced

programmers' navigations, just as it led to significantly better predictions of novice programmers' navigations.

6.3 Bottomline: Do our models generalize?

Revisiting the research questions that we mentioned earlier in this chapter:

1. *(RQ1) PFIS-V generalization.* Does PFIS-V model more experienced variations foraging behaviors as well as it modeled novice programmers' variations foraging?

Our results in Section 6.3 suggest yes, the results from the new study being similar to that from the old study data. This similarity suggests that PFIS-V, which we developed and evaluated based on novice programmers' foraging behaviors also generalized to experienced programmers.

2. *(RQ2) PFIS-H generalization.* Does PFIS-H predict the hierarchical foraging behaviors of more experienced programmers (including their navigations to non-code patches) as well as it did for novice programmers?

Yes, as the results in Section 6.4 suggest. PFIS-H was more predictive than PFIS-V of participants' navigations in the new study, just as it was in the previous study, suggesting that PFIS-H, which originally modeled novice programmers' hierarchical foraging also generalized to the experienced programmer population. Further, as we discussed in the previous chapter (Section 5.4), hierarchical foraging also generalized to experienced programmers' foraging in a new situation, namely in a single variant of a Java program during a debugging task.

CHAPTER 7. CONCLUDING REMARKS

Discussion and contributions

Our goal in this dissertation has been to gather evidence defending (or rejecting) the thesis: IFT can explain and predict people's information seeking in the presence of variants. Towards this end, we first conducted a user study: qualitative results suggest that IFT's constructs and propositions do help able to explain variations foraging behaviors of programmers. We then built two computational models, namely PFIS-V and PFIS-H, that operationalized the notions we derived from this study of how IFT applies to variants. Quantitative evaluations of the models with data from two empirical studies suggested that IFT's notions of cost, value and scent are able to predict programmers' navigations during variations foraging. Thus, the fundamental contribution of this dissertation is the theory of variations foraging, which is grounded in the framework of IFT.

One intended utility of these theoretical foundations is in tool building and evaluation. Tool builders can now leverage IFT--its propositions, computational models and design patterns—for building and evaluating variations-support tools. For example, one result in Chapter 4 is that modeling similarity ("variant-of" links) led to better predictions by PFIS-V. This result suggests that providing navigation affordances between similar patches in different variants could aid foragers foraging among variants. Similarly, tool builders can leverage IFT computational models such as PFIS-V and PFIS-H to evaluate their tools, such as to predict how a user will use a tool or to compare different tool design options. For example, as we did with PFIS-V, tool builders can compare text-based vs. text-and-topology-based equivalence to gauge which of these two comparison schemes will better aid foragers in a given foraging task.

Looking beyond variations foraging, this dissertation demonstrated the benefits of modeling hierarchical foraging in IFT computational models. By accounting for the hierarchical organization of information and the variable costs of navigations to different locations, PFIS-H was able to make better predictions of programmer navigations both in the presence of variants and without them. This result suggests that

tools supporting programmer navigations, such as programmer recommendation tools, can benefit by taking the hierarchical and cost aspects into consideration.

This thesis also contributes the first IFT computational model—in programming as well as non-programming domains—that accounts for non-textual patches. By modeling graphical outputs, PFIS-H was able to make more and better navigation predictions than its predecessors, thereby demonstrating the benefits of including non-textual patches to understanding people's foraging.

Open problems

One avenue for future research is to address the following challenges of modeling non-textual patches. (1) How can we automatically caption graphical patches (which can be highly specific as in our study) so as to model these at scale, or to include in tools such as just-in-time recommendation systems? (2) How can we model other kinds of non-textual patches, such as video, audio and interactive visualizations? (3) Will a captioning approach still work for modeling these non-textual patch types (as it did for the graphical patches in our study)?

Another avenue for future research is about understanding and supporting the foraging behaviors in non-textual patches. (1) How do notions of cues, links and scent differ between textual and non-textual patches? (2) How will traditional foraging activities such as between-patch foraging, within-patch foraging and enrichment apply to non-textual patches? (3) What foraging strategies do foragers adopt when foraging in non-textual patches? (4) How can tools better support foraging in non-textual patches, or more generally environments with heterogeneous patch types, such as by providing better links, aggregation or filtering capabilities?

This dissertation also raises the following open questions about variations foraging. First, as we discussed in Chapter 3, our study participants constructed "stories" to guide their foraging. It remains an open question what construct of IFT we should instantiate these stories as (e.g., scent or cues or patches or a new construct).

Second, future research should investigate the creation of variants (producer side), such as the following questions. (1) At what intervals should a producer (or automatic tool) save a variant (e.g., every little change, once an hour, once a day or every time

the program is executed)? (2) What cues should a producer should leave for future consumers (foragers) of the variants?

Third, variations foraging remains to be investigated for other kinds of variants, such as different syntactic representations (or variational representations) for the same information (e.g., audio and its textual transcript, flowchart and its corresponding program code). These variants are different from the program variants (same syntax, different semantics) in our study. It remains an open question whether a forager will adopt different foraging strategies in these variants than in our studies: will they focus less on difference-comparison when the different variants are not syntactically similar to each other? will foragers build different kinds of stories when foraging in such non-chronological variants?

Conclusion

In summary, this work contributes the theoretical foundations for variations foraging. Grounded in IFT, this dissertation extends the scope of IFT's validity beyond a single variant of textual information: it explains and predicts foraging in the presence of multiple variants of an artifact, accounts for non-textual patches of information and demonstrates the advantages of including the hierarchical organization of information and its associated foraging costs. We believe that these contributions will enable a principled approach to engineering variations-support tools as well as motivate further research into the fundamentals of information seeking.

REFERENCES

[1] Anderson, J. R. (1996). ACT: A simple theory of complex cognition. *American Psychologist, 51*(4), 355-365.

[2] Beckwith, Laura, and Margaret Burnett. "Gender: An important factor in end-user programming environments?" In *2004 IEEE symposium on visual languages-human centric computing*, pp. 107-114. IEEE, 2004.

[3] Bhowmik, Tanmay, Nan Niu, Wentao Wang, Jing-Ru C. Cheng, Ling Li, and Xiongfei Cao. "Optimal group size for software change tasks: A social information foraging perspective." *IEEE transactions on cybernetics* 46, no. 8 (2015): 1784-1795.

[4] Bhowmik, Tanmay, Nan Niu, Prachi Singhania, and Wentao Wang. "On the role of structural holes in requirements identification: an exploratory study on open-source software development." *ACM Transactions on Management Information Systems (TMIS)* 6, no. 3 (2015): 10.

[5] Bragdon, Andrew, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. "Code bubbles: a working set-based interface for code understanding and maintenance." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2503-2512. ACM, 2010.

[6] 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. http://doi.acm.org/10.1145/1518701.1518944

[7] Burnett, Margaret M., and Brad A. Myers. "Future of end-user software engineering: beyond the silos." In *Proceedings of the on Future of Software Engineering*, pp. 201-211. ACM, 2014.

[8] Bush, Vannevar. "As we may think." *The Atlantic Monthly* 176.1 (1945): 101-108.

[9] Chi, Ed H., Peter Pirolli, and James Pitkow. "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*, pp. 161-168. ACM, 2000.

[10] Chi, Ed H., Peter Pirolli, Kim Chen, and James Pitkow. "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*, pp. 490-497. ACM, 2001.

[11] Chi, Ed H., Adam Rosien, Gesara Supattanasiri, Amanda Williams, Christiaan Royer, Celia Chow, Erica Robles, Brinda Dalal, Julie Chen, and Steve Cousins. "The bloodhound project: automating discovery of web usability issues using the InfoScentπ simulator." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 505-512. ACM, 2003.

[12] Clements, Paul, and Linda Northrop. *Software product lines: practices and patterns*. Vol. 3. Reading: Addison-Wesley, 2002.

[12] Codoban, Mihai, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. "Software history under the lens: A study on why and how developers examine it." In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1-10. IEEE, 2015.

[13] Concurrent Versions Systems. https://www.nongnu.org/cvs/. Retrieved 5th August 2019.

[14] Kumar, Deepthi S. "A Language for Querying Source Code Repositories." (2017). Oregon State University. https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/qn59q849h

[15] Dodge, Jonathan, Sean Penney, Andrew Anderson, and Margaret M. Burnett. "What Should Be in an XAI Explanation? What IFT Reveals." In *IUI Workshops*. 2018.

[16] Fleming, Scott D., Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. "An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, no. 2 (2013): 14.

[17] Git Version control system. https://git-scm.com/. Retrieved 5th August, 2019.

[18] Glassman, Elena L., Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. "OverCode: Visualizing variation in student solutions to programming problems at scale." *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, no. 2 (2015): 7.

[19] Google Docs. https://en.wikipedia.org/wiki/Google_Docs. Retrieved 5th August, 2019.

[20] Guo, Philip J., and Margo I. Seltzer. "Burrito: Wrapping your lab notebook in computational infrastructure." (2012).

[21] Guo, P.J., 2012. *Software tools to facilitate research programming* (Doctoral dissertation, Stanford University).

[22] Hartmann, Björn, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. "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*, pp. 91-100. ACM, 2008.

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

[24] Henley, Austin Z., and Scott D. Fleming. "Yestercode: Improving code-change support in visual dataflow programming environments." In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, pp. 106-114. IEEE, 2016.

[25] Hill, Charles, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. "Trials and tribulations of developers of intelligent systems: A field study." In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 162-170. IEEE, 2016.

[26] Holmes, Reid, and Robert J. Walker. "Systematizing pragmatic software reuse." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, no. 4 (2012): 20.

[27] "Information Age". .https://en.wikipedia.org/wiki/Information_Age. Retrieved 5th August, 2019

[28] "Information Overload". https://en.wikipedia.org/wiki/Information_overload. Retrieved 5th August, 2019.

[29] "Information Pollution". https://www.wikiwand.com/en/Information_pollution. Retrieved 5th August, 2019.

[30] Logan Engstrom, Garrett Finucane. 2015. Hextris. Retrieved September 23, 2015 from https://hextris.github.io/hextris/

[31] Logan Engstrom, Garrett Finucane, Noah Moroze, Michael Yang. 2015. Hextris. Retrieved September 23, 2015 from https://github.com/Hextris/hextris

[32] Kery, Mary Beth, Amber Horvath, and Brad A. Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In *CHI*, pp. 1265-1276. 2017.

[33] Kery, Mary Beth, and Brad A. Myers. "Exploring exploratory programming." In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 25-29. IEEE, 2017.

[34] Jensen, Carlos, Heather Lonsdale, Eleanor Wynn, Jill Cao, Michael Slater, and Thomas G. Dietterich. "The life and times of files and information: a study of desktop provenance." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 767-776. ACM, 2010.

[35] Johnson, Pontus, Mathias Ekstedt, and Ivar Jacobson. "Where's the theory for software engineering?" *IEEE software*29, no. 5 (2012): 96-96.

[36] Kactus. https://kactus.io/. Retrieved on 5th August, 2019.

[37] Ko, Andrew J., Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks." *IEEE Transactions on Software Engineering* 32, no. 12 (2006): 971-987.

[38] Kumar, Ranjitha, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. "Bricolage: example-based retargeting for web design." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2197-2206. ACM, 2011.

[39] Kuttal, Sandeep Kaur, Anita Sarma, and Gregg Rothermel. "Predator behavior in the wild web world of bugs: An information foraging theory perspective." In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pp. 59-66. IEEE, 2013.

[40] Kuttal, Sandeep K., Anita Sarma, and Gregg Rothermel. "On the benefits of providing versioning support for end users: an empirical study." *ACM Transactions on Computer-Human Interaction (TOCHI)* 21, no. 2 (2014): 9.

[41] Lawrance, Joseph, Rachel Bellamy, and Margaret Burnett. "Scents in programs: Does information foraging theory apply to program maintenance?." In Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on, pp. 15-22. IEEE, 2007.

[42] Lawrance, Joseph, Rachel Bellamy, Margaret Burnett, and Kyle Rector. "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*, pp. 1323-1332. ACM, 2008.

[43]Lawrance, Joseph, Margaret Burnett, Rachel Bellamy, Christopher Bogart, and Calvin Swart. "Reactive information foraging for evolving goals." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 25-34. ACM, 2010.

[44] Lawrance, Joseph, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. "How programmers debug, revisited: An information

foraging theory perspective." *IEEE Transactions on Software Engineering* 39, no. 2 (2013): 197-215.

[45] Le, Duc, Eric Walkingshaw, and Martin Erwig. "# ifdef confirmed harmful: Promoting understandable software variation." In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 143-150. IEEE, 2011.

[46] Lunzer, Aran, and Kasper Hornbæk. "Subjunctive interfaces: Extending applications to support parallel setup, viewing and control of alternative scenarios." *ACM Transactions on Computer-Human Interaction (TOCHI)* 14, no. 4 (2008): 17.

[47] Maier, S., & Minas, M. (2015, October). Recording, processing, and visualizing changes in diagrams. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on* (pp. 131-135). IEEE.

[48] Martos, Carlos, Se Yeon Kim, and Sandeep Kaur Kuttal. "Reuse of variants in online repositories: Foraging for the fittest." In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, pp. 124-128. IEEE, 2016.

[49] Mercurial SCM. https://www.mercurial-scm.org/. Retrieved 5th August 2019.

[50] Miller, George A. "Informavores." *The study of information: Interdisciplinary messages* (1983): 111-113.

[51] Mikami, Hiroaki, Daisuke Sakamoto, and Takeo Igarashi. "Micro-Versioning Tool to Support Experimentation in Exploratory Programming." In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 6208-6219. ACM, 2017.

[52] Microsoft Office. https://en.wikipedia.org/wiki/Microsoft_Office. Retrieved 5th August 2019.

[53] Muniswamy-Reddy, Kiran-Kumar, David A. Holland, Uri Braun, and Margo I. Seltzer. "Provenance-aware storage systems." In *USENIX Annual Technical Conference, General Track*, pp. 43-56. 2006.

[54] Myers, Brad, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. "How designers design and program interactive behaviors." In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 177-184. IEEE, 2008.

[55] Nabi, Tahmid, Kyle MD Sweeney, Sam Lichlyter, David Piorkowski, Chris Scaffidi, Margaret Burnett, and Scott D. Fleming. "Putting information foraging theory to work: Community-based design patterns for programming tools." In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, pp. 129-133. IEEE, 2016.

[56] Niu, Nan, Anas Mahmoud, and Gary Bradshaw. "Information foraging as a foundation for code navigation (NIER track)." In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 816-819. ACM, 2011.

[57] Niu, Nan, Anas Mahmoud, Zhangji Chen, and Gary Bradshaw. "Departures from optimality: understanding human analyst's information foraging in assisted requirements tracing." In *Proceedings of the 2013 International Conference on Software Engineering*, pp. 572-581. IEEE Press, 2013.

[58] Newman, Mark W., and James A. Landay. "Sitemaps, storyboards, and specifications: a sketch of Web site design practice." In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, pp. 263-274. ACM, 2000.

[59] Ong, Kevin. "Using information foraging theory to understand search behavior in different environments." In *Proceedings of the 2017 Conference on Conference Human Information Interaction and Retrieval*, pp. 411-413. ACM, 2017.

[60] Parnin, Chris, and Carsten Gorg. "Building usage contexts during program comprehension." In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 13-22. IEEE, 2006.

[61] Penney, Sean, Jonathan Dodge, Claudia Hilderbrand, Andrew Anderson, Logan Simpson, and Margaret Burnett. "Toward Foraging for Understanding of StarCraft Agents: An Empirical Study." In *23rd International Conference on Intelligent User Interfaces*, pp. 225-237. ACM, 2018.

[62] Perez, Alexandre, and Rui Abreu. "A diagnosis-based approach to software comprehension." In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 37-47. ACM, 2014.

[63] Piorkowski, David, Scott D. Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E. John, Margaret Burnett, and Rachel Bellamy. "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models." In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pp. 109-116. IEEE, 2011.

[64] Piorkowski, David, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. "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, pp. 1471-1480. ACM, 2012.

[65] Piorkowski, David J., Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel KE Bellamy, and Joshua Jordahl. "The whats and hows

of programmers' foraging diets." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3063-3072. ACM, 2013.

[66] Piorkowski, David, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. "To fix or to learn? How production bias affects developers' information foraging during debugging." In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 11-20. IEEE, 2015.

[67] Piorkowski, David, Austin Z. Henley, Tahmid Nabi, Scott D. Fleming, Christopher Scaffidi, and Margaret Burnett. "Foraging and navigations, fundamentally: developers' predictions of value and cost." In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 97-108. ACM, 2016.

[68] Piorkowski, David, Sean Penney, Austin Z. Henley, Marco Pistoia, Margaret Burnett, Omer Tripp, and Pietro Ferrara. "Foraging goes mobile: Foraging while debugging on mobile devices." In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*, pp. 9-17. IEEE, 2017.

[69] Pirolli, Peter, and Stuart Card. "Information foraging in information access environments." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 51-58. ACM Press/Addison-Wesley Publishing Co., 1995.

[70] Pirolli, Peter. "Computational models of information scent-following in a very large browsable text collection." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3-10. ACM, 1997.

[71] Pirolli, Peter, and Stuart K. Card. "Information foraging models of browsers for very large document spaces." In *Proceedings of the working conference on Advanced visual interfaces*, pp. 83-93. ACM, 1998.

[72] Pirolli, Peter, Stuart K. Card, and Mija M. Van Der Wege. "Visual information foraging in a focus+ context visualization." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* pp. 506-513. ACM, 2001.

[73] Pirolli, Peter, Stuart K. Card, and Mija M. Van Der Wege. "The effects of information scent on visual search in the hyperbolic tree browser." ACM Transactions on Computer-Human Interaction (TOCHI) 10, no. 1 (2003): 20-53.

[74] Pirolli, Peter, and Wai-Tat Fu. "SNIF-ACT: A model of information foraging on the World Wide Web." In *International Conference on User Modeling*, pp. 45-54. Springer, Berlin, Heidelberg, 2003.

[75] Pirolli, Peter. *Information foraging theory: Adaptive interaction with information*. Oxford University Press, 2007.

[76] Pirolli, Peter. "An elementary social information foraging model." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 605-614. ACM, 2009.

[77] Srinivasa Ragavan, Sruti, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. "Foraging among an overabundance of similar variants." In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 3509-3521. ACM, 2016.

[78] Ragavan, Sruti Srinivasa, Bhargav Pandya, David Piorkowski, Charles Hill, Sandeep Kaur Kuttal, Anita Sarma, and Margaret Burnett. "PFIS-V: modeling foraging behavior in the presence of variants." In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 6232-6244. ACM, 2017.

[79] Srinivasa Ragavan, Sruti. Version Control Systems: An Information Foraging Perspective. Oregon State University. 2018. https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/3x816s495

[80] Srinivasa Ragavan, Sruti, Mihai Codoban, David Piorkowski, Danny Dig, Margaret Burnett. “Version Control Systems: An Information Foraging Perspective”. Transactions of Software Engineering, IEEE, 2019. DOI: 10.1109/TSE.2019.2931296. To appear.

[81] Rochkind, Marc J. "The source code control system." *IEEE transactions on Software Engineering* 4 (1975): 364-370.

[82] Rosson, Mary Beth, and John M. Carroll. "The reuse of uses in Smalltalk programming." *ACM Transactions on Computer-Human Interaction (TOCHI)* 3, no. 3 (1996): 219-253.

[83] Ruthruff, Joseph R., Amit Phalgune, Laura Beckwith, Margaret Burnett, and Curtis Cook. "Rewarding" Good" Behavior: End-User Debugging and Rewards." In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, pp. 115-122. IEEE, 2004.

[84] Sheil, Beau. "DATAMATION®: POWER TOOLS FOR PROGRAMMERS." In *Readings in artificial intelligence and software engineering*, pp. 573-580. 1986.

[85] Simon, Herbert A. "Designing organizations for an information-rich world." (1971): 37-72.

[86] http://www.semdesigns.com/Products/SmartDifferencer/

[87] Smeltzer, K.J., 2018. Design and Application of Variational Representations. Oregon State University. https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/08612t93m

[88] “Software versioning”. https://www.wikiwand.com/en/Software_versioning. Retrieved on 5th August, 2019.

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

[90] Stephens, David W., and John R. Krebs. Foraging theory. Princeton University Press, 1986.

[91] Apache Subversion. https://subversion.apache.org/. Retrieved 5th August 2019.

[92] Time Machine. https://en.wikipedia.org/wiki/Time_Machine_(macOS). Retrieved on 5th August, 2019.

[93] Terry, Michael, and Elizabeth D. Mynatt. "Recognizing creative needs in user interface design." *Proceedings of the 4th conference on Creativity & cognition*. ACM, 2002.

[94] Terry, Michael, and Elizabeth D. Mynatt. "Side views: persistent, on-demand previews for open-ended tasks." In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pp. 71-80. ACM, 2002.

[95] Terry, Michael, Elizabeth D. Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. "Variation in element and action: supporting simultaneous development of alternative solutions." In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 711-718. ACM, 2004.

[96] Tichy, Walter F. "Design, implementation, and evaluation of a revision control system." In *Proceedings of the 6th international conference on Software engineering*, pp. 58-67. IEEE Computer Society Press, 1982.

[97] Versions: Git for designers. https://versions.sympli.io/. Retrieved on 5th August, 2019.

[98] Walkingshaw, Eric. "The choice calculus: A formal language of variation." (2013). Oregon State University. https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/k643b3668

[99] Yoon, Young Seok, and Brad A. Myers. "An exploratory study of backtracking strategies used by developers." In *Cooperative and Human Aspects of Software Engineering (CHASE), 2012 5th International Workshop on*, pp. 138-144. IEEE, 2012.

[100] Yoon, YoungSeok, and Brad A. Myers. "Supporting selective undo in a code editor." In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 223-233. IEEE, 2015.