Exploring Eye Tracking Data on Source Code via Dual Space Analysis

Li Zhang, Jianxin Sun, Cole Peterson, Bonita Sharif, Hongfeng Yu
Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, Nebraska USA 68588
{liz, jianxins, cpeterso, bsharif, yu}@cse.unl.edu

***Abstract*—Eye tracking is a frequently used technique to collect data capturing users' strategies and behaviors in processing information. Understanding how programmers navigate through a large number of classes and methods to find bugs is important to educators and practitioners in software engineering. However, the eye tracking data collected on realistic codebases is massive compared to traditional eye tracking data on one static page. The same content may appear in different areas on the screen with users scrolling in an Integrated Development Environment (IDE). Hierarchically structured content and fluid method position compose the two major challenges for visualization. We present a dual-space analysis approach to explore eye tracking data by leveraging existing software visualizations and a new graph embedding visualization. We use the graph embedding technique to quantify the distance between two arbitrary methods, which offers a more accurate visualization of distance with respect to the inherent relations, compared with the direct software structure and the call graph. The visualization offers both naturalness and readability showing time-varying eye movement data in both the content space and the embedded space, and provides new discoveries in developers' eye tracking behaviors.**

***Index Terms*—visualization, eye tracking, developer classification, content space, embedded space**

I. Introduction

The cognitive process (e.g., code reading and debugging) of a software developer is often related to multiple factors in software engineering (SE) such as programmer efficiency, hierarchically organized software structures, relations among source code, and integrated development environments (IDEs). The knowledge of such processes can lead to many implications and benefits. For example, knowledge of efficient code reading processes can be explicitly exposed to students in their SE learning [1]–[3]. Knowing programmers' code viewing efficiency can also help project managers assign workload [3] and intervene when they see patterns of stress or evaluate task difficulties [4].

To understand such a cognitive process, SE researchers use eye tracking devices to keep track of the psycho-physiological states while performing debugging tasks using measurements such as pupil dilation, fixation duration, and saccade frequency [3]–[6]. By examining the time-varying data of participants' visual attention to particular content or IDE panes on a screen, researchers seek to identify patterns from the high-dimensional data across time, content, or screen.

Compared with the conventional analytics of eye tracking data [7], [8], the increasing complexity of software introduces unique challenges in exploring eye tracking data to identify and compare individual user patterns. A software system is typically organized in a hierarchical tree structure (e.g., the organization of classes and methods and the dependency relations between these elements). In addition, the system is associated with a call graph where each node represents a method and each link represents a caller/callee relation. Given a *content* space containing a software tree structure and an associated call graph, different users may have different strategies to explore this space for code reading or debugging tasks. Also, note that using iTrace [5], [9] to collect eye tracking data during bug fixes increases the complexity of the visualization since we are no longer limited to short code snippets. A common strategy is to directly visualize the software tree structure [10], [11], and/or the call graph [12], and superimpose eye trajectories which can intuitively display the sequence of user focus in the content space and *qualitatively* compare trajectories. However, these existing visualization techniques cannot *qualitatively* reveal the distance among trajectories as the inherent semantic information of a software system may not be appropriately displayed in the visualization.

In this paper, we present a new dual-space approach to enhancing the analysis of eye tracking data generated during users' viewing or debugging a software system. First, we directly visualize the content space by simultaneously depicting the hierarchical structure, the call graph, and user eye trajectories for a software system. We leverage a radial layout and an edge bundling algorithm to effectively depress visual clutter. The visualization in the content space allows users to intuitively examine the interplay between user trajectories, the software hierarchy, and the call relations. Second, we create an *embedded* space by transforming the call graph and/or the tree structure via graph embedding. The visualization of user trajectories in the embedded space allows us to quantitatively measure distances among trajectories. We link the visualizations in these two spaces and use our approach with an eye tracking dataset of debugging processes [13]. Our approach facilitates the study of different user behaviors through different views and reveals some discoveries that have not been identified in existing work.

II. Related Work

Though many studies consider software reviewing or debugging using eye tracking data, they do not directly consider

the graph structure in contents [3], [4], [14]–[16]. As seen in the work of Suliman et al. [17], a graph is used to model the communication relationships between software developers reading each other's files and is what researchers try to learn from eye tracking data. Our work focuses on understanding and evaluating reading and navigation patterns through program files and how such patterns reflect, interact with, and deviate from the underlying graphical software pattern defined by its tree structure and call relations. As our work spans two domains, i.e, eye tracking and software domain, we review the corresponding studies.

A. Eye Tracking Analytics

Fixations and saccades are two fundamental metrics for describing behaviors of eye movement. A complete sequence of fixations and saccades is a scanpath [6]. There are two major visualization tracks, direct visualization, and statistical analysis. The major difference is whether it represents and visualizes individual features (e.g., location, duration, and gaze) directly or use summary statistics such as saccade frequency to abstract individual features for statistical analysis. Area of interest (AOI) is an important feature to capture readers' comprehension patterns. Formally, an AOI can be any user-defined area that is quantified by fixation and saccade data. How researchers define their AOI may not be explicit and can be very different [3], [15].

1) Direct visualization: Direct visualizations put fixation circles, color, and/or saccade lines directly over the content and give a user-friendly overview of gaze patterns. It is widely used in various areas not limited to software [6], such as application stores [18], info-graphics design [19]. The content-focus overlay plot [1], [3], [14], [20] exploits the uniformity between contents and screen, i.e, that contents are static and fixed to a specific location on the paper, screen or objects. It is straightforward that the distance in contents can be directly represented by their position on a page or screen. In this plot, we cannot infer about user viewing pattern along time. We cannot distinguish from a user gazing less frequently for a longer duration from another gazing more frequently but for a shorter time.

A timeline flow visualization can remedy this deficiency. Sharif et al. use a line and time coordinate to plot the fixation points in time [1]. This visualization gives the false impression that there is no meaningful reading time because it shows only fixation points rather than duration specific to each line. To overcome this shortness, Clark and Sharif introduced the *skyline* visualization, which shows a user's duration of fixation and saccade in time [14]. This visualization gives a clear correspondence between the focus in content with the length of horizontal line signifying duration corresponding to that content. This work follows the direction pointed by Clark and Sharif and studies scan patterns in reference to control structures and programs with greater length and size.

2) Statistical analysis: The statistical analysis direction considers patterns in the features of interest to explain the variance between groups who succeed and groups who failed the tasks. Typical research questions focus on whether code fixation time, focus on the specific type of code (e.g. logical, declaration, and loops) [15], behaviors such as switching windows, or using particular IDE panes (e.g. variable, console, or code) [3] or regression rate [21] makes a significant predictor of debugging success. While these studies yield important insights as to what features characterize efficient code readers, it is insufficient to treat coding as natural text. If we compare method declarations and syntax to vocabularies of source code, the structure is the grammar of source code and is also essential for code comprehension. Since it is difficult to quantify structure statistically, we adopt a visualization method to examine how different participants interact with the structure of source code.

B. Software Visualization

Researchers propose many metaphors to visualize hierarchical or tree software structures. For example, Andrian et al. [10] present a 3D representation for software structures. Wettel et al. [11] visualize large-scale software systems as a city. Holten presents an aesthetic visualization to overcome the current issues associated with clustering in the rendering of large complex graphs [12]. A software program with classes, dependencies, and call relations can be represented as a graph with inclusion relations and adjacency relations. In related work [22], Mabakane designed a visualization of the call graph to help identify performance bottlenecks in the execution of a parallel program. A limitation of these visualization approaches is that the distance between nodes, which are usually perceived as the length of the curve connecting the nodes, are mostly suggestive of connectivity. In other words, one longer curve does not mean that the distance between a pair of nodes is longer than another pair of nodes with a shorter curve. The requirement of distance measure in our software visualization drives for a more meaningful graph representation, where the distance between the nodes actually captures their semantic distance in the software systems.

III. Approach

A. Design Requirements

The requirements from the domain experts on analyzing eye tracking data on source code mostly include:

R1. Identify user behavior according to their scanpath. Visualizations can help qualitatively understand how users read and navigate through the content space of both hierarchical and adjacent relations to locate bugs. More specifically, what patterns in visualization can reveal the success or failure of bug locating, and any general differences between those who located the bugs and those who did not.

R2. Interpret distance in both intuitive and meaningful ways. The semantic distance between one method and another method is not as straightforward as the Euclidean distance between two points in a 2D plane. However, the visualization should represent and layout methods in a meaningful way that preserves the distances.

R3. Support deeper analysis by visualizing the time dimension of scanning. Ideally, the visualization should integrate multiple dimensions in addition to the content dimension: the time dimension indicating sequences of fixations or saccades, the duration of fixations, and/or the repetitive pattern of back and forth reading. A reasonable assumption here is that if a method m_a calls another method m_b which further calls m_c, then a sequential saccade pattern of m_a, m_b, and m_c may be more effective than a pattern of m_b, m_a, and m_c. The synthetic visualization with local visiting relation details can be extended to study the transition of pattern in time with similarly structured relations.

B. Design Challenges

We identify four major challenges in the existing visualization approaches that lead to our current research efforts: the absence of content structure, limited content space, lack of meaningful distance representation in visualization, and separation or omission of time data.

Existing literature highlighted the general trend for the stimulus material or object using heatmap. The mapping of color directly onto the focused content is straightforward and works effectively when the material is a static web page or an informatics design on a poster. However, if the content space becomes much larger where there are tens or hundreds of text files and thousands of lines per file, this direct mapping will fall apart as the information becomes increasingly cluttered.

Another challenge with reading code is that the content has its own structures. Typically, they follow a tree structure in organizing the file systems and a graphical structure between the method calls. The visualization of human's reading focus on larger and structured contents is not fully explored yet.

For traditional visualization of contents, given that the AOIs are static or fixed to its screen or physical location, a distance measure between AOIs corresponds directly to their physical distances. However, for a software system rendered in an IDE, the distance measure between a method m_a in a file f_b and another method m_c in a file f_d is not intuitive. Additionally, suppose we have m_a and m_b both in a class c_c and suppose they are k lines apart from each other; however, m_a calls method m_b at some point, we can no longer simply use k as the sole measure of the distance between m_a and m_b. We need a better distance measure capturing the cost of reading in hierarchically structured content space.

Finally, the heatmap based visualization failed to suggest the sequence of focus which can be important in reading code. Readers who located an intermediate method to the bug should be expected to locate the bug more easily than those who did not. We can say safely that if a participant found the intermediate bug location (on the bug path) but failed to locate the bug, he/she may not be as experienced. However, if a participant did not find any relevant methods, we are not so sure about his/her chances of locating the bug. Sequences like these suggest the reading efficiency, logical reasoning, understanding of the structure and code, and familiarity with the IDE.

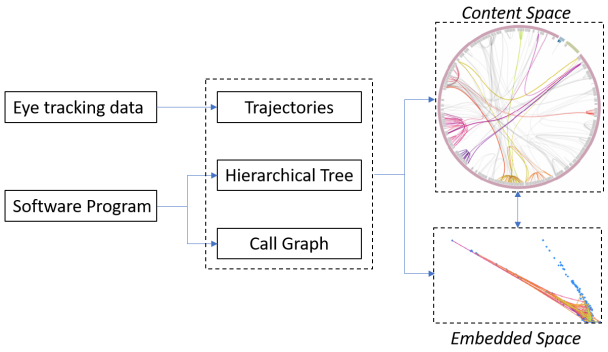


Fig. 1. Overview of our approach

C. Our Design

Figure 1 shows an overview of our approach. In this work, we study eye tracking data for understanding user behaviors during their code reading and debugging. We take the eye tracking data and the software program as the inputs to derive the trajectory data, the hierarchical tree structure, and the call graph. The dataset is visualized using two linked views in a content space and an embedded space to reveal different patterns of user behaviors.

1) Data collection and processing: We used the Tobii X60 eye tracker to record 22 participants debugging 3 different tasks, rated as either easy or difficult. Among the participants, 12 are labeled professional (i.e., the participants 1-12), and 10 are labeled novice (i.e., the participants 22-31). For each participant, the eye tracking data records time, duration, line, method, the file name of focus, pupil dilation, and coordinates in the screen for each task. Notice that the time interval between recordings is not equal and the number of observations differs across participants based on their reading. In this paper, we extract the user trajectories from the eye tracking data. One trajectory of a user consists of the sequence of methods that the user read, and the time and the duration that the user spent on a method.

We extract the hierarchical tree structure and the call graph of the software system using *java-callgraph*, an open source static analysis tool [23]. The tree structure is implied in the name of methods. For example, the method name *root.net.sf.jabref.EntryTable.addSelectionListener()* suggests that the class *EntryTable* is in the package *root.net.sf.jabref* and the method is *addSelectionListener()*. The call graph indicates the caller/callee relations among the methods. With the hierarchical tree and call graph combined, we are then ready to visualize users' trajectories on the software structure.

2) Content space analysis: We leverage different schemes to visualize the tree structure, the call graph, and the user trajectories and reveal participants' efficiency of reading in the content space. We choose the D3 library [24] in our implementation.

First, we use a common tree visualization method, the radial layout [12], to visualize the hierarchical structure where methods are visualized as leaves (rectangles along the inner

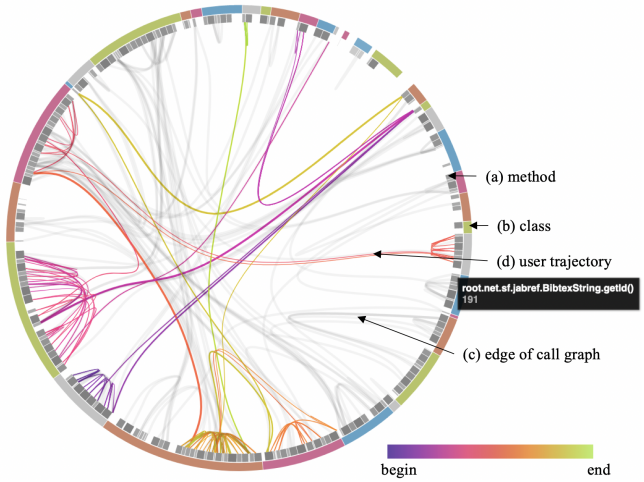


Fig. 2. An example of content space visualization. (a), each small box along the outer orbit represents a method, where details of the method are displayed on hover. (b), each arc along the orbit represents a class. (c), the gray curves are the edge bundling results of the call graph, where each edge connects a caller and a callee. (d), the color curve corresponds to a user trajectory. The color scalar indicates the scanning sequences.

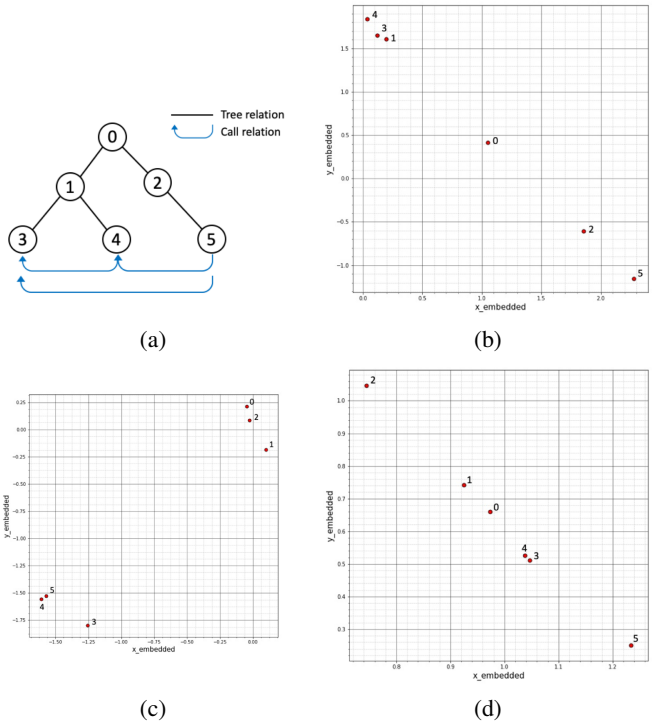


Fig. 3. A simple example of embedding a graph (a) into 2D space according to (b) the tree, (c) the call graph, and (d) the combination of the tree and the call graph.

circle) and their parents are visualized as outer arcs. Both types of visual elements are arranged in a radial manner. The detailed information of each method is displayed when hovered over.

Second, we employ the hierarchical edge bundling method [12] to visualize the call graph. The method uses piece-wise cubic B-spline curves to draw edges, and similar shaped edges are bundled together. Compared to other graph visualization methods, the edge bundling method more effectively reduces visual clutter and is more visually distinct. The bundling alpha value is set to 1 in our implementation. A higher alpha value gives a more bundling effect and renders a clearer and less clustered visualization [12]. The curves of the call graph are rendered in gray.

Third, we render user trajectories as curves laid over the gray curves of the call relations in the software. We use a color scale for the trajectories to indicate the scanning sequence with a darker purple color as the beginning of the scan and lighter green color as the end. Fewer colors in a trajectory indicate that less transitions between methods occurred.

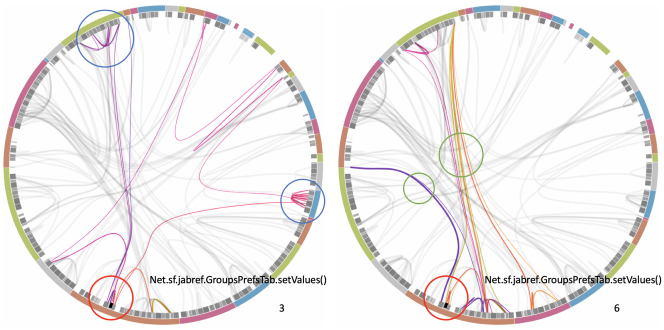
Figure 2 shows an example of our visualization in the content space. We can see that the tree structure of the software is suggested in the layout of the methods where the methods of the same class are next to each other. Through the superposition of a group of user trajectories and the calling relation graphs with edge bundling, we can clearly see the relationships between the user scanning sequences and the caller/callee relations. We note that other tree visualization techniques (e.g., rooted tree, radial tree, treemap, etc.) can be also exploited with edge bundling [12] to visualize the content space.

3) Embedded space analysis: The visualization results in the content space can concisely and simultaneously show the software hierarchical structure, call relations, and user trajectories. However, it cannot be directly used to quantitatively measure the distance between the visual metaphors (e.g., the curves of the trajectories and the call graphs), as the semantic meaning of the graphs is not fully preserved in the visualization [25].

We employ graph embedding techniques to address this issue. Graph embedding effectively converts a graph into a low dimensional space where the graph's structural information and properties are maximally preserved [25]. In particular, we use the node2vec [26] graph embedding method to map graphs into a 2D embedded space while preserving the underlying graph structure. This enables us to layout graph nodes (i.e., methods) with meaningful distances to each other.

The node2vec algorithm is semi-supervised and learns a mapping of nodes to a low-dimensional feature space. For the purpose of content space visualization, we decide to embed the graph into a 2D space. The node2vec algorithm can preserve graph neighborhoods of nodes in a 2D space such that two neighborhood nodes in a graph can be mapped to two close 2D points. Different content spaces can be generated from different graphs. In this work, we generate the content spaces using all three graphs that we investigate: tree structure graph, call structure graph and tree-call combined structure graph.

Figure 3 shows an example of embedded results using a simple graph. The tree in Figure 3(a) represents a simple hierarchical structure of a software program where the blue curves represent the call relations. Figure 3(b) shows the result using the tree structure. We can clearly see that the distances among the nodes in the embedded space (b) exactly correspond



(a) (b)

Fig. 4. The visualization results of the software structure, the call graph, and the trajectory for the participants 3 (a) and 6 (b) in the content space. Participant 3 scans the methods in a local tree structure, as the ones in the blue circles, while participant 6 read the code frequently along the call relations, as the ones in the green circles. The red circles highlight a possible AOI that is shared by 3 and 6 and is also across most participants (5 out of 7) who successfully fixed the bug. The name of the intermediate bug location is *setValues ()* in the class *GroupsPrefTab*.

to the tree structure (a). For example, the nodes 3 and 4 are the children of the node 1, and they are embedded closely in (b). Similarly, the nodes 2 and 5 are close to each other in (b), while the node 0 is roughly at the center of the space. Figure 3(c) shows the result using the call graph, where we can clearly see the nodes are distributed according to their call relations (e.g., the group formed by the nodes 3, 4, and 5). In Figure 3(d), the placement of the nodes is generated using the combined graph of the tree relations and the call relations. For example, the nodes 3 and 4 are close to each other as they have the same parent as well as the call relation. The node 5 becomes farther away from 2 compared to the results in Figure 3(b). This is because the node 5 has more call relations to 3 and 4 even though 5 is the child of 2.

After we generate an embedded graph, we overlay user trajectories on the embedded graph and also use juxtaposition for comparison between different users. The user trajectories are colored using the same scale in the content space visualization. In this way, we can examine if a user would scan the code according to the software structure and/or the call relations.

IV. Results

We documented participants solutions for the bugs they located and asked about their confidence levels, comments, and evaluations of task difficulty. We also further evaluated each solution to be either acceptable or not. For example, in Task 2, we have valid data from 20 participants, 5 out of 11 professional participants successfully fixed the bug and 3 out of 9 novice participants also fixed the bug. We use this evaluation as supplemental information as well as ground truth for our analysis.

A. Content Space Results

The radial visualization presents an intuitive analysis of the participants' scanning pattern and whether they follow the tree structure and/or call relations. For example, Figure 4 shows the visualization results for participants 3 and 6. We can see that participant 3 mostly follows the tree structure when reading methods in the same class, as represented by the more aggregated local curves (e.g., the ones in the blue circles in Figure 4 (a)). Alternatively, the participant 6 navigates through the code seemingly following the call relations, as shown in the curves in the green circles in Figure 4 (b), which appear to overlap with the underlying software relations in grey lines. Such uncertainty can be further examined with the embedded call space (see Section IV-B1).

We can further identify a possible AOI, corresponding to the method 228 *GroupsPrefTab.setValues ()*, in the red circles in both Figure 4 (a) and (b). Figure 5 gives the trajectories for all 20 participants for task 2 [13]. Examining across the participants in Figure 5, we can find that participants 3, 4, 5, 6, 11, 25, 29, and 30 all show this pattern in their trajectories of which 80% of these participants who are professionals successfully fixed bugs (the participant 11 did not fix the bug). It is also interesting to observe that the participants 29 and 30 did not fix the bug even when they are in this AOI. An alternative interpretation is that getting to an intermediate bug location will give professionals more of a guarantee to finding the final bug location than novice participants. The participants 29 and 11 shared almost identical scanning patterns and may have failed the task because too much focus was put on the local files and did not relate to the bug problem well. In addition, it is difficult to distinguish the participants 5 and 30 because they share almost the same pattern in this visualization.

B. Embedded Space Results

1) Interpretation of embedded spaces: We embed the software tree structure, the call graph, and the combined graph of the tree and call relations to understand how the embedding algorithm interprets the distance of the software methods.

To understand the effect of different graphs (i.e., the tree, the call graph, and the combined graph) in graph embedding, we first highlight the points of two methods *getText ()* and *setText ()* in red in Figure 6. Similar to the *get* and *set* methods in most programs, these two methods are also defined in the same class and do not have direct call relations between them. Intuitively, the distance between them should not be far as they are semantically related even though there is no call relation. The distance shown in the embedded space of the tree graph is indeed close, as shown in Figure 6 (a). However, the distance between the two becomes very far in the embedded space of the call graph, as shown in Figure 6 (b). Figure 6 (c) shows a medium distance in the embedded space of the combined graph of tree and call relations.

Figure 7 offers a more complex example with a method calling another both directly and indirectly where more relations imply a closer distance and a shorter curve. For example, the method 175 calls 163 directly and also indirectly through 171. The method 171 calls 163 only directly. Therefore, the method 175 appears closer to 163 than 171. For the method 159, since it calls 163 only indirectly, it shows as the most distant from 163 among the three methods. The embedded call graph gives

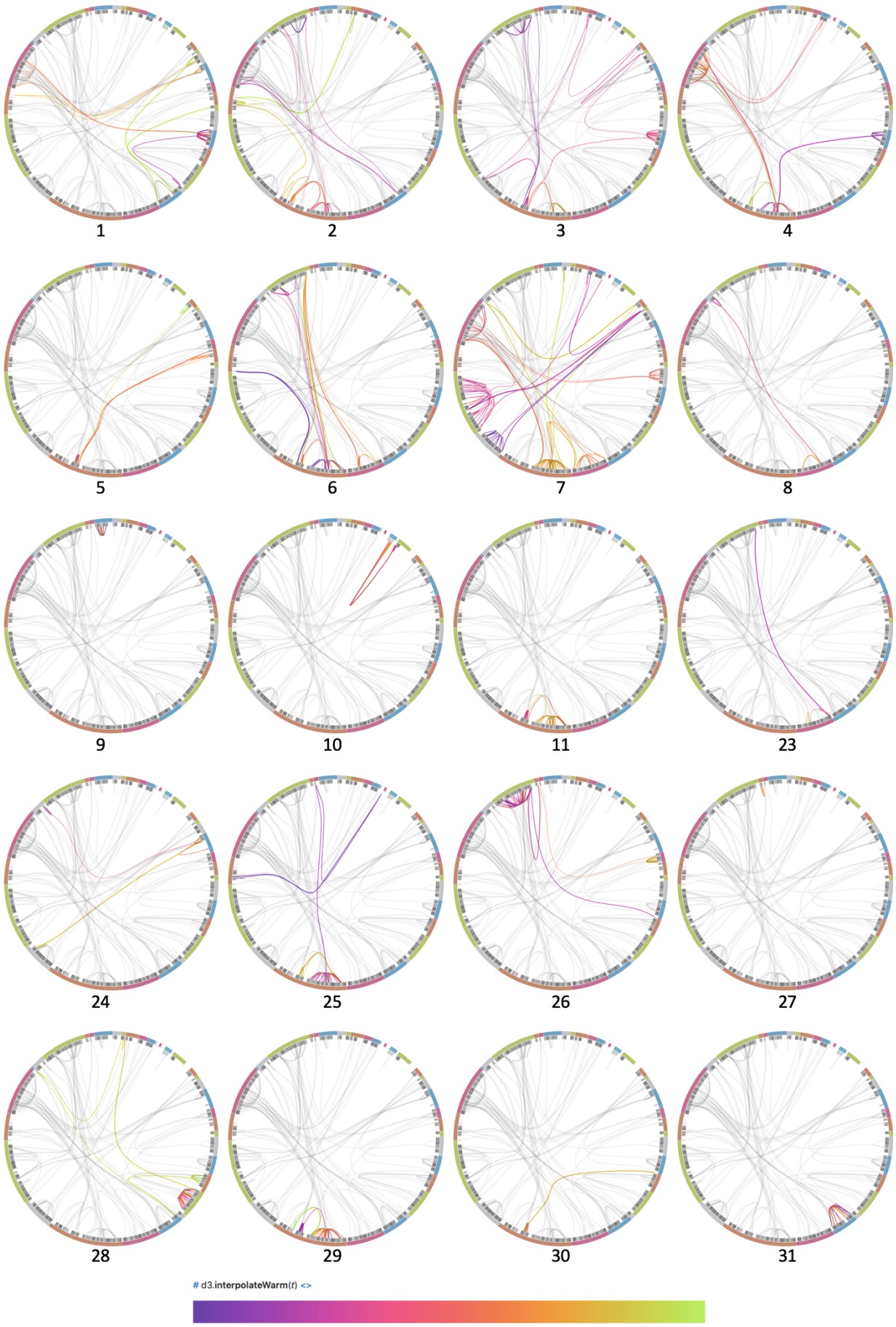
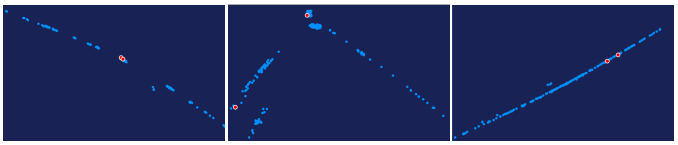
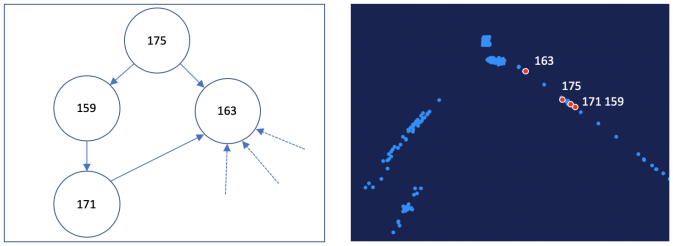


Fig. 5. The visualization results of the software structure, the call graph, and the trajectory for all participants in the content space. The bottom color scale is used for all participants, indicating starting time in deep purple and ending time in light green in the sequence of scanning. Our survey-based evaluation suggests that the participants 3, 4, 5, 6, 9, 25, 27, and 31 have successfully located and fixed the bug.



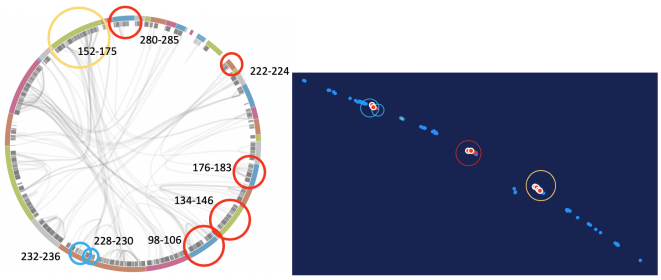
(a) tree (b) call (c) combined

Fig. 6. The two methods *getText ()* and *setText ()* are in the same *FieldEditor* class, and are colored in red. They are plotted in the tree graph (a), the call graph (b), and the combined graph (c).



(a) (b)

Fig. 7. An example illustrating more complex call relations. The rank of distance to the method 163 from smallest to largest is 175, 171, and 159. This is because the method 175 features both direct and indirect calls to 163, and the methods 171 and 159 feature a direct call and a indirect call to 163, respectively.



(a) radial (b) tree

Fig. 8. The visualization of embedded tree graph in correspondence with the radial layout. Eight sample areas from (a) are selected. The methods within a circle represent the leaf nodes in the same subtree. The methods in a circle in (a) belong in the cluster circled in the same color in (b). These methods are also nearest in (b). Note that the two blue clusters of methods in (a) share the same depth and are one level deeper than the methods in the red and yellow circles, which causes the blue circled methods are distinct in (b). Similarly, the red and yellow circled methods share the same depth, but the yellow circled methods are more than twice the size of the other circled ones. The embedding algorithm can also capture the structure of the subtree by its size and structure, and distinguish the yellow and red circled methods in (b).

a distance according to two principles implied here: 1) more paths of calling means a closer distance; and 2) direct calling is closer in distance than indirect calling.

To demonstrate how different graph visualization results correspond to each other, we first circle eight groups of methods, where each group is in the same subtree, as shown in Figure 8 (a). The subtrees may be at different levels of depth. Methods in the same subtree stay closely together as shown in Figure 8 (b). Figure 9 shows an example of embedding the calling graph. Two subgraphs 9 (a) and 9 (b) are plotted into 9 (c). The methods with close calling relations stays together (within a or b) and those with distant calling relations (between a and b) are further away from each other.

Figure 10 illustrates further with the participant 6's trajectories. For the listed methods in the figure, the user's trajectories do not seem to be more effective in the call graph than those in the tree graph. However, if we examine the distance between the methods 67 and 72 and between the methods 239 and 253, the distance given by the tree graph becomes much shorter. Therefore, the participant 6 mainly follows the tree structure while visiting these methods. The result provides a more clear indication, compared with Figure 4(b).

2) General patterns: This section presents the results based on the general trends observed visually among all participants. The first question we ask is whether or not any trajectories follow the call graph more dominantly. Figure 11 shows the comparison results of the participants 1 and 11 using the different embedded graphs. We can see the busy and long lines in both Figure 11 (a) and Figure 11 (c), implying that neither the call nor tree relations govern the participant 1's reading. By comparing Figure 11 (b) and (d), we can see that participant 11 paid more attention to the call relations than the tree relations. In addition, the participants 1 and 11's call distances are average when compared across all participants as shown in Figure 13. We identify 7 participants featuring this pattern. Considering both the tree and call relations as shown in Figure 11 (e) and (f), we can observe seemingly more distant lines in the participant 1. We no longer include analysis of the combined graph in this subsection as the plotting resembles a straight line and causes serious overlapping of nodes and trajectories.

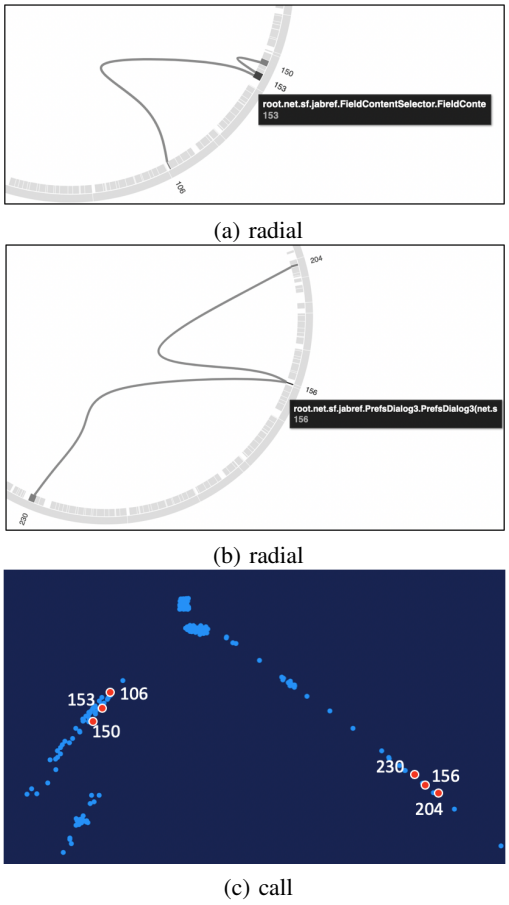
Figure 12 and Figure 13 show the trajectories from all 20 participants in the embedded tree graph and call graph, respectively. Based on the visual comparison across all the trajectories, we summarized a few general patterns as shown in Figure 14. As shown in Figure 12, the tree graph in the embedded space generally depicts a line-shaped plot of blue points. We can identify a pattern **c** that approximately represents the middle cluster along the line-shaped plot. These are the methods at the same depth level and with similar community features. Using this pattern **c** as the cutting point, we can further identify the patterns **lt** and **rt** that characterize the left and right position of methods along the line-shaped plot, respectively. For the call graph as shown in Figure 13, the patterns show up are **l1**, which represents a group of similar paths to the AOI (i.e, the method 228) and its close neighborhood methods (e.g., 229 and 230 in Figure 8(a)). The other ends of the paths are close the method 197.

Using the flag patterns in Figure 14, we characterize each participant by whether they follow the patterns in Table I. Note that **rt** is not included in Table I, as we found that **rt** is not noticeably helpful to differentiate the participants. In Table I the distance column, long, medium, and short represent the participant's trajectories in the tree graph in Figure 12. Considering L as the longest distance in the distances between any pair of methods, we categorize a trajectory according to the longest distance x among methods traversed. The

TABLE I

IDENTIFYING GROUPS WITH FLAG PATTERNS. THE PARTICIPANTS MARKED WITH * HAVE SUCCESSFULLY LOCATED AND FIXED THE BUG.

Distance	long				medium									short						
Participant	1	2	7	25*	3*	4*	5*	6*	10	23	24	26	30	8	9*	11	27*	28	29	31*
l1		✓	✓	✓	✓	✓	✓					✓	✓			✓			✓	
lt				✓	✓	✓	✓			✓	✓	✓	✓							
c																	✓	✓		✓



(a) radial

(b) radial

(c) call

Fig. 9. The visualization of embed call graph in correspondence with the radial layout. Upon clicking a method node in the radial visualization, the methods with direct relations get highlighted as shown in (a) and (b). For both (a) and (b), the methods with direct calling relations show as close to each other in (c). But the distance of the methods between that in (a) and that in (b) are much more distant.

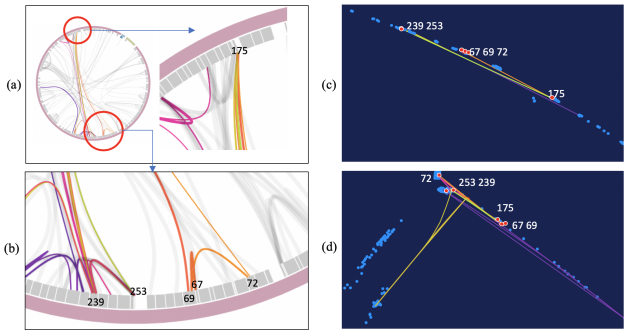
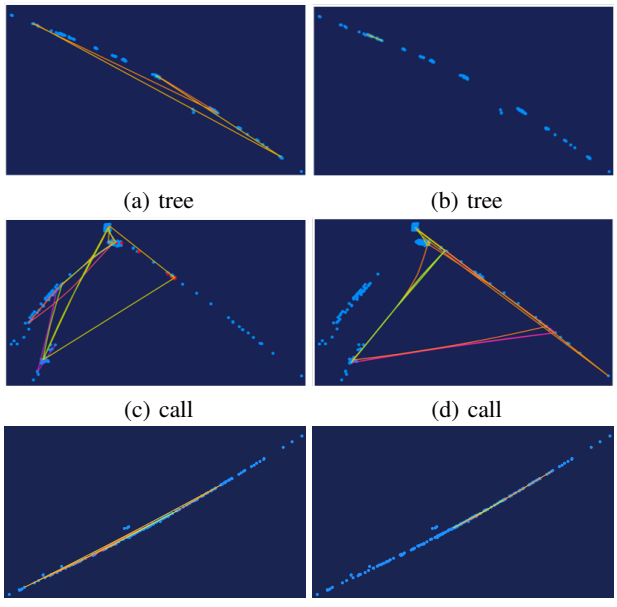


Fig. 10. An example translating the participant 6's trajectories in the radial layout (a, b) into the embedded spaces of the tree (c) and the call graph (d). (c) reflects the tree relation between methods. Note that the distance between 175 and 69 are similar to that between 239 and 253 in (d), even through the former looks much more distant than the later in radial layout.



(a) tree (b) tree

(c) call (d) call

(e) combined (f) combined

Fig. 11. Comparison of the participants 1 (left) and 11 (right) in their reading patterns across different embed graphs (i.e, the tree, the call graph, and the combined graph). We can see clearly that the participant 11's reading pattern mostly follows the tree graph. However, the participant 1 does not show an obvious pattern following either tree or call graph.

categories are long ($x > \frac{L}{2}$), medium ($\frac{L}{4} \leq x \leq \frac{L}{2}$), and short($x \leq \frac{L}{4}$). We observed that out of 7 participants with short trajectories, 3 were bug solvers. Note that the tiny trajectories, such as ones of the participant 31, do not necessarily mean poor code coverage as methods of different classes may be clustered together as shown in Figure 8. However, if we also consider the call graph which features fewer colors and nodes visited, it is safe to conclude that a participant reads with less code coverage. The participants in the group of short distances in tree graph are most likely tree structure followers.

In general, we can see a few patterns characterizing all trajectories in call graphs. For example, the trajectories of the participants 2, 3, 4, 5, 7, 11, 25, 26, 29 and 30 are similar in that they follow the flag pattern **l1** as shown in Figure 14 (b). Table I shows detailed results of grouping with different flag patterns. Examining whether participants traverse **l1** or focus on the central location **c** as shown in Figure 14 (a) gives a pool of 13 participants, among whom 6 are the bug solvers.

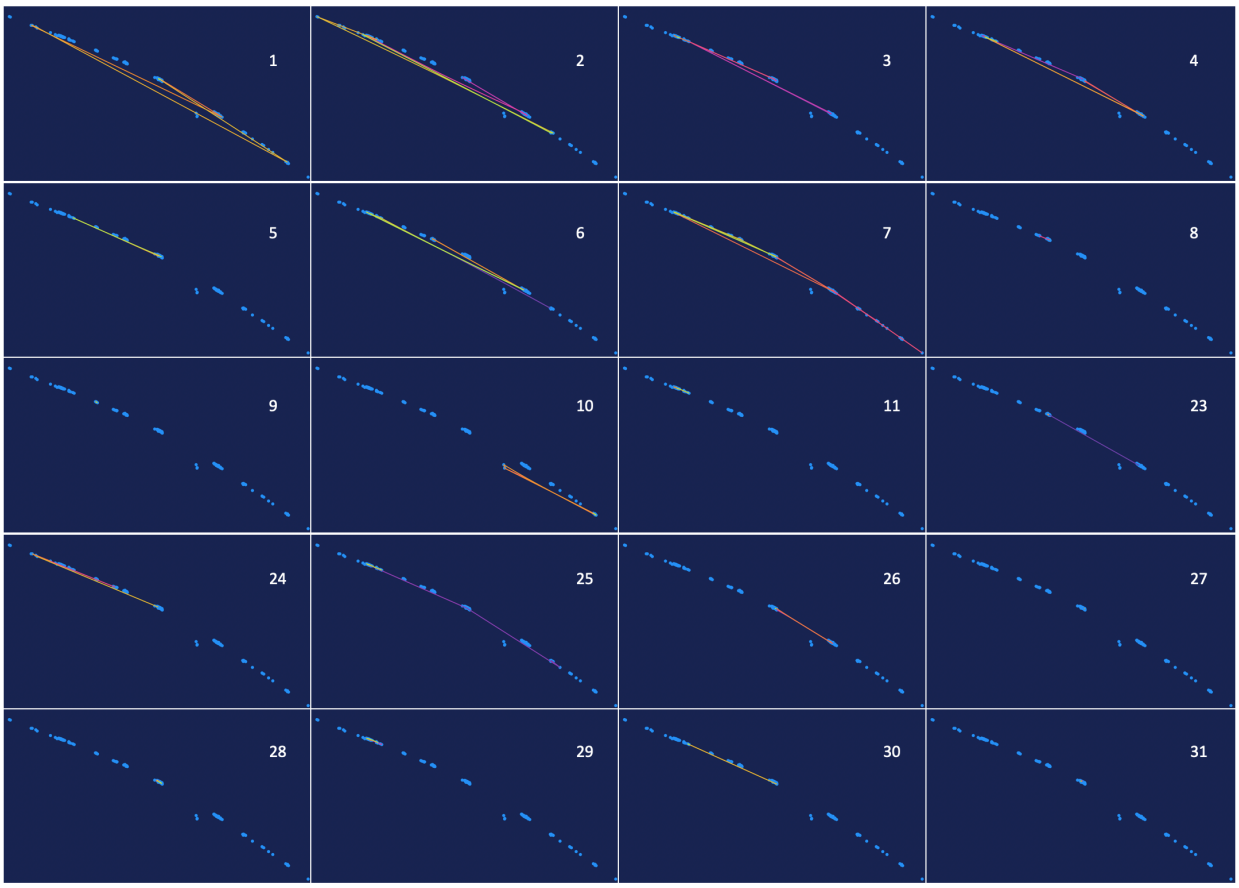


Fig. 12. The visualization results of the tree graph. Some similar patterns in radial layout become more distinct. For example, we can quickly identify participants 8, 9, 11, 27, 28, 29, and 31 read most likely following the tree structure.

Only the bug solvers 6 and 9 do not have either pattern.

C. Deeper into the patterns

With help of AOIs we can examine patterns in greater detail. The above visual analysis yields a pool of 13 participants with 6 of them correctly fixing bugs, we still have 7 participants (2, 7, 11, 26, 28, 29, and 30) who are false positives. The method 228 can be visited from its close neighbor method 230 whose caller method is 156. Alternatively, it can be visited from its callee method 163 which is called by the methods 239 and 249. Looking at individual trajectory data and tracing how each come to visit the intermediate AOI, we can further divide participants into three groups: a) reasonable landers (3, 4, 5, 6, 11, 25, 29), b) backward travelers (30) who visited first from callee to caller in the sequence of visits, and c) missed travelers (2, 7, 26, 27, 28) who did not visit the AOI.

As we have expected, sequences in scanning can reflect the developers' reasoning. If we examine the participants 11, 25, and 29 in Figure 5, though they have very similar patterns related to the AOI around the method 228, the participant 25 who successfully fixed the bug started his reading in the area distinct from the method 228 and searched his way into the AOI at the end, whereas the other two were jumping right into the AOI and then reading away from it and missed the bug.

In addition, the color changes of a trajectory offers more insight. For example, the trajectories with more colors mean that the participants (e.g., 1, 2, 4, 6, 7, 25) went through more methods and potentially had better code coverage. For the professional participants, their trajectories showed more colors (i.e., from purple to red to green) implying that the scanpath is less repetitive. In tree graphs, we verify that bug solvers typically start with longer purple curves and ended with shorter green curves, which indicates that they are working to narrow down the space of searching.

D. Discussion

The embedded space results provide a more accurate depiction of distance, where neighborhood methods, according to different relations, can be mapped to nearby 2D points. Thus, we can compare user trajectories' distances in a more meaningful way. However, as we use a 2D embedded space, certain trajectory segments can overlap. For example, although several participants show similar patterns in Figures 12 and 13, it is less direct to tell whether or not the methods are visited in a similar sequence, thereby making it difficult to perceive more detailed differences among trajectories. To address this issue, we plan to investigate data in a 3D space [27] by introducing a time axis additional to the 2D embedded space to generate more comprehensive analytics results.

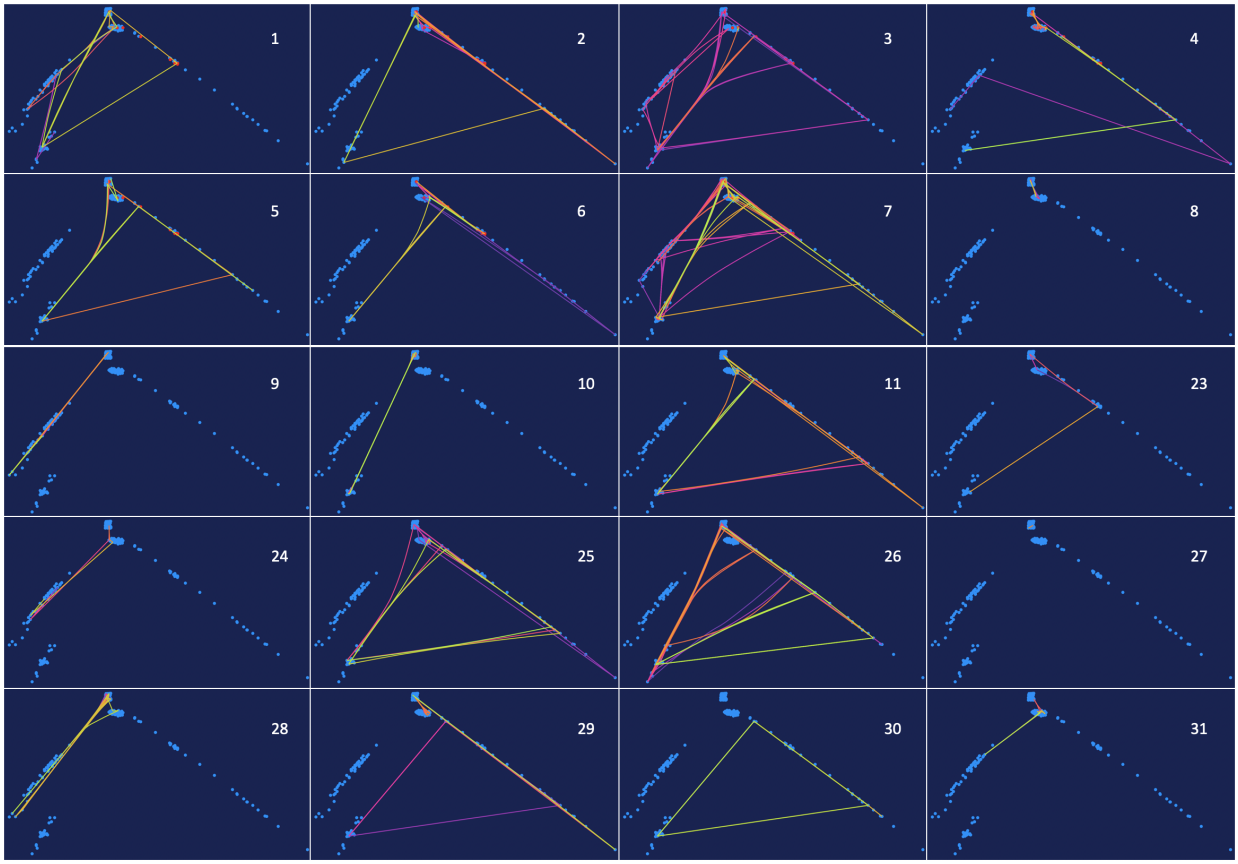
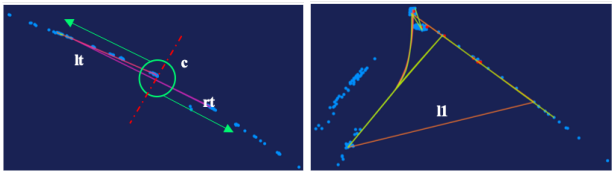


Fig. 13. The visualization results of the call graph, and the trajectories for all participants for in the embedded space using the same color scale as Fig. 5. Some similar patterns in radial layout become more distinct. For example, if we compare the participant 30 with 5, we can see that the participant 30 1) is less active in exploring the codes (less color in trajectory) and 2) shows a stroke (the tiny line in the bottom cluster) that are not found in other participants.



(a) (b)

Fig. 14. Flag patterns from the tree graph (a) and the call graph (b). In (a), we identify three patterns, **lt** for left, **c** for center, and **rt** for right. In (b), we identify a line **l1** that appears frequently among the trajectories.

V. Conclusions and Future Work

In this paper, we leverage multiple visualization schemes for the analysis of participants' patterns in reading and debugging software programs. We visualize the software program with the software's tree and call graphs in the content and embedded spaces and overlay participants' trajectories on the graphs using edge bundling techniques and color scales to indicate the sequence of focus. Our visualization and clustering results provide an intuitive and effective way of understanding reading patterns, such as whether participants follow the tree or call graph structure and how participants generally navigate to the bug location and in what sequence. Our visualization uses the eye trace of the software system as its use case, but it has the potential of applications in other domains as well. For example, in social science, we can visualize the pattern of a person getting himself into a social network, suggesting his social skills or personality. Similarly, we can investigate the pattern of how information or news gets spread. In this paper, we are primarily concerned with eye tracking data for debugging patterns and efficiency.

In our future work, we would like to visualize multiple perspectives, such as psycho-physiological features and screen usage, in an integrated way. The possible extra measurements (e.g., electrogastrographic data, body movement data, facial expression, galvanic skin response, pupil dilation, etc.) may be integrated into data analysis with user eye trajectories. In this way, we can not only identify more holistic user patterns but also gain a deeper understanding of the interplay between software structures and user exploration and comprehension. Finally, we plan on performing a user study to gain feedback on the usefulness of this visualization to developers and researchers.

Acknowledgment

This research has been sponsored in part by the National Science Foundation through grants IIS-1652846 and CCF-1855756.

References

[1] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM, 2012, pp. 381–384.

[2] M. Ciolkowski, O. Laitenberger, D. Rombach, F. Shull, and D. Perry, "Software inspections, reviews and walkthroughs," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 641–642.

[3] K. Chandrika, J. Amudha, and S. D. Sudarsan, "Recognizing eye tracking traits for source code review," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–8.

[4] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using psycho-physiological measures to assess task difficulty in software development," in *Proceedings of the 36th international conference on software engineering*. ACM, 2014, pp. 402–413.

[5] B. Sharif, T. Shaffer, J. Wise, and J. I. Maletic, "Tracking developers' eyes in the IDE," *IEEE Software*, vol. 33, no. 3, pp. 105–108, 2016.

[6] T. Blascheck, K. Kurzhals, M. Raschke, M. Burch, D. Weiskopf, and T. Ertl, "State-of-the-art of visualization for eye tracking data," in *Proceedings of EuroVis*, vol. 2014, 2014.

[7] K. Kurzhals, B. Fisher, M. Burch, and D. Weiskopf, "Evaluating visual analytics with eye tracking," in *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization*. ACM, 2014, pp. 61–69.

[8] T. Blascheck, M. Burch, M. Raschke, and D. Weiskopf, "Challenges and perspectives in big eye-movement data visual analytics," in *2015 Big Data Visual Analytics (BDVA)*. IEEE, 2015, pp. 1–8.

[9] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "itrace: eye tracking infrastructure for development environments," in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications, ETRA 2018, Warsaw, Poland, June 14-17, 2018*, 2018, pp. 105:1–105:3. [Online]. Available: https://doi.org/10.1145/3204493.3208343

[10] A. Marcus, L. Feng, and J. I. Maletic, "3D representations for software visualization," in *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003, pp. 27–ff.

[11] R. Wettel and M. Lanza, "CodeCity: 3D visualization of large-scale software," in *Companion of the 30th international conference on Software engineering*. ACM, 2008, pp. 921–922.

[12] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[13] K. Kevic, B. Walters, T. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Eye gaze and interaction contexts for change tasks - observations and potential," *Journal of Systems and Software*, vol. 128, pp. 252–266, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2016.03.030

[14] B. Clark and B. Sharif, "iTraceVis: Visualizing eye movement data within Eclipse," in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2017, pp. 22–32.

[15] F. Peng, C. Li, X. Song, W. Hu, and G. Feng, "An eye tracking research on debugging strategies towards different types of bugs," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2016, pp. 130–134.

[16] H. Uwano, A. Monden, and K.-i. Matsumoto, "DRESREM 2: An analysis system for multi-document software review using reviewers' eye movements," in *2008 The Third International Conference on Software Engineering Advances*. IEEE, 2008, pp. 177–183.

[17] M. Suliman, H. Bani-Salameh, and A. Saif, "Visualizing communications between software developers during development," *International Journal of Software Engineering and Its Applications*, vol. 10, no. 3, pp. 131–140, 2016.

[18] J. Fu, "Usability evaluation of software store based on eye-tracking technology," in *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*. IEEE, 2016, pp. 1116–1119.

[19] A. Majooni, M. Masood, and A. Akhavan, "An eye-tracking study on the effect of infographic structures on viewer's comprehension and cognitive load," *Information Visualization*, vol. 17, no. 3, pp. 257–266, 2018.

[20] G. Andrienko, N. Andrienko, W. Chen, R. Maciejewski, and Y. Zhao, "Visual analytics of mobility and transportation: State of the art and further research directions," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 8, pp. 2232–2249, 2017.

[21] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking metrics in software engineering," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 96–103.

[22] M. S. Mabakane, "Effective visualisation of callgraphs for optimisation of parallel programs: a design study," Ph.D. dissertation, Faculty of Science, 2019.

[23] G. Gousios, "Programs for producing static and dynamic (runtime) call graphs for java programs," https://github.com/gousiosg/java-callgraph, 2018.

[24] N. Q. Zhu, *Data visualization with D3. js cookbook*. Packt Publishing Ltd, 2013.

[25] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.

[26] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," *CoRR*, vol. abs/1607.00653, 2016. [Online]. Available: http://arxiv.org/abs/1607.00653

[27] J. Wei, H. Yu, R. W. Grout, J. H. Chen, and K.-L. Ma, "Dual space analysis of turbulent combustion particle data," in *2011 IEEE Pacific Visualization Symposium*. IEEE, 2011, pp. 91–98.