A Longitudinal Study of Programmers' Backtracking

YoungSeok Yoon
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
youngseok@cs.cmu.edu

Brad A. Myers
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
bam@cs.cmu.edu

Abstract—**Programming often involves reverting source code to an earlier state, which we call *backtracking*. We performed a longitudinal study of programmers' backtracking, analyzing 1,460 hours of fine-grained code editing logs collected from 21 people. Our analysis method keeps track of the change history of each abstract syntax tree node and looks for backtracking instances within each node. Using this method, we detected a total of 15,095 backtracking instances, which gives an average backtracking rate of 10.3/hour. The size of backtracking varied considerably, ranging from a single character to thousands of characters. 34% of the backtracking was performed by manually deleting or typing the desired code, and 9.5% of all backtracking was selective, meaning that it could not have been performed using the conventional undo command present in the IDE. The study results show that programmers need better backtracking tools, and also provide design implications for such tools.**

Keywords—empirical study; backtracking; undo; interactive development environments (IDE)

I. INTRODUCTION

When working on source code, programmers often revert some changes made in code and return various code fragments back to an earlier state, which we refer to as *backtracking*[1]. Backtracking happens for many reasons. Previously, we conducted a preliminary lab study and an online survey to better understand the programmers' backtracking practices [1]. In the survey, 75% of the respondents reported that they face backtracking situations at least "sometimes," and we found that they use different strategies depending on the type of backtracking situation they face. In the lab study, we observed the coding behavior of the participants and qualitatively analyzed the data, which identified some problems programmers face while backtracking, such as failing to locate code fragments to be reverted, and not being able to use the undo command when there are other intermediate changes that they want to maintain.

These two preliminary studies, however, had many limitations. Because we wanted to maximize the chance of observing programmers' backtracking behavior in a short lab study, we designed the tasks specifically to require the participants to backtrack. As a consequence, the study does not really tell us how frequently those problems would occur in real development situations. In addition, although we asked how frequently they face various backtracking situations in the online survey, it is possible that the programmers backtrack unconsciously and the survey results may not correctly reflect what actually happens in real development. Plus, the survey cannot provide details about how and under what circumstances programmers backtrack.

In order to investigate programmers' backtracking further, we conducted an extensive, longitudinal study as a follow-up to the previous two studies, which we report here. The goals of our new study are twofold. First, we wanted to obtain backtracking statistics in order to quantify the need for backtracking tools. We specifically focused on collecting quantitative data from this study, as our previous studies were mostly qualitative. Second, we wanted to identify backtracking situations that are not very well supported by existing programming tools, and to extract design implications for developing better backtracking tools that might improve programmer productivity.

We performed the analysis with the following research questions in mind:

- How frequently do programmers backtrack in a real programming environment? (Section III.A)
- How large are the backtrackings? (Section III.B)
- How exactly do programmers perform backtracking? Are they backtracking manually? (Section III.C)
- Is there evidence of "exploratory programming"? (Section III.D)
- Are there backtrackings happening across multiple editing sessions? (Section III.E)
- Are there selective backtrackings, which cannot be performed by the undo command? (Section III.F)
- Do programmers backtrack to the same code repeatedly? (Section III.G)

In this paper, we first present the analysis method we devised to answer these questions (Section II). Using the fine-grained code edit logs we collected from 21 people, totaling 1,460 total hours of programming time, we could track the entire histories of all abstract syntax tree (AST) nodes in their source code, and use the per-node history data to detect backtracking instances.

Next, we present the backtracking related information we sought, and answers to the research questions listed above (Section III). The results show that programmers were backtracking 10.3 times per hour on average, and the backtracking size varied from a single character to more than a

[1] Note that this is a different use of the term *backtracking* from the algorithm often used in logic programming for solving constraint satisfaction problems.

TABLE I. PARTICIPANT GROUPS

Group	#	Description	Coding Time (hours) (min / avg / max / **total**[a])
G0	1	The first author of this paper	294 / 294 / 294 / **294**
G1	13	Graduate students @ CMU	3 / 40 / 216 / **520**
G2	5	Research programmers / system scientists @ CMU	6 / 118 / 446 / **588**
G3	2	Graduate students @ the University of Pittsburgh	6 / 29 / 51 / **57**

[a.] min / avg / max: per-user values
total: the sum of all the users' coding times in each group

thousand characters, which spans from backtracking out of simple parameter value changes to significant algorithmic changes. Programmers were backtracking manually by deleting or typing code in 34% of all backtracking cases. In 20% of the backtracking cases, programmers first changed some code, ran the application, and then backtracked the changes. About 97% of all backtrackings were done within the same editing session. 9.5% of all backtracking instances were selective, which means that the conventional undo command could not handle them, because they were intermingled with other changes that the programmers did not want to lose. Finally, only 15% of the backtracked nodes were reverted to the same state again later.

Section IV discusses the limitations of the analysis method and some possible future work directions. Section V discusses related work and Section VI concludes the paper.

II. ANALYSIS METHOD

A. Log Data

We analyzed programmers' fine-grained code editing logs to see how they actually backtrack while coding. The log data was collected using FLUORITE [2], a logging plug-in for the Eclipse interactive development environment (IDE) that we developed to capture all of the character-level code changes and the editing commands such as Copy, Paste, and Undo. A total of 21 participants were recruited for the study (Table I). Among the participants, about 8 out of 13 in G1 were Masters students who were working on their Studio projects with real clients, and the people in G2 are professional programmers whose primary job is programming. After signing the consent form, the participants were asked to install FLUORITE on their machines and perform their own programming tasks as usual. They were periodically asked to archive and send their logs to us, and were paid up to $50 for their participation. The first author of this paper also collected his own logs, which were analyzed together with the data collected from the participants. The data has been collected since April 2012, and contains 1,460 hours of coding activities (excluding all the idle time exceeding 5 minutes) as of the writing of this paper. All of the participants were programming in Java using Eclipse (v3.6 or higher) across a variety of Windows and Macintosh machines.

Whenever a source file is opened in the Eclipse editor for the first time, FLUORITE writes the initial snapshot of the file into the log. From then on, it keeps track of all the edit operations (insert, delete, or replace) of the source file, so that the snapshots at any point in time for that file can be reconstructed. A logged edit operation contains the timestamp of the edit, the offset indicating where in the file the edit was made, and the actual text deleted and/or inserted by the edit.

B. AST-Node Base Change History Tracking

We define backtracking as "programmers going back at least partially to an earlier state of code either by removing inserted code or by restoring removed code" [1]. By this definition, a trivial backtracking detector would detect any pairs of edit operations <op1, op2> where the later performed operation op2 reverts what op1 did earlier, at least partially. For example, if an earlier edit operation inserts "foo" in the code (op1) and that "foo" is deleted anytime later by another operation (op2), this pair of operations could be considered as a backtracking instance. We first started out using this formalism. However, this naïve approach had several major limitations:

- Too many false positives were found, caused by auto-formatting, organizing import statements, auto-generated comments, etc.
- It could not detect multi-step backtracking, where a code fragment is reverted to an earlier state by multiple edit operations.
- It could not tell syntactically equivalent code fragments very well, as it was purely text-based. For example, if a method call `foo();` is deleted and then later put back as `foo␣();` with an additional space before the parentheses, it could not know the two statements are actually identical and failed to detect such backtracking.
- It treated comments and source code the same way.
- It was difficult to tell the high-level intent of the detected backtracking automatically, and required substantial manual inspection to get useful information about the detected backtracking instances.

In order to address these issues, we devised an analysis approach based on the abstract syntax tree (AST) of the source code. The basic idea of this approach is to keep the evolution history of individual AST nodes of interest throughout the lifetime of the nodes.

Our new analyzer processes the logged edit events sequentially off-line (separate from Eclipse). When a new file open event is seen, the analyzer keeps the snapshot of that source file and parses the snapshot (using `ASTParser` from the Eclipse JDT) to store all the AST nodes of interest, for example all the *statement* nodes in the source file. For each AST node, the analyzer remembers the start position and length of the node within the file, and the initial snapshot of that node. When keeping the snapshot, the analyzer normalizes all the formatting such as whitespaces and indentations.

Whenever an edit operation is seen, the analyzer applies that operation on the last known snapshot of the file in order to get an updated snapshot, and parses the updated version. Then the analyzer determines all the AST node(s) that were affected by this change and updates the start position and length of the node. In addition, the analyzer adds the new version to the evolution history of that specific node, but only when the normalized snapshot differs from the last known snapshot.

Because the log data contains character-level edit operations rather than AST-level differences, a source file can be in an incomplete state (i.e., containing parse errors) after applying an edit operation. In such cases, the analyzer tries to update only the start position and length values of all the known nodes according to the edit offset and length. When the file returns to a parseable state after some subsequent edits, the analyzer again tries to match all the previously known nodes and the current nodes to find all the affected nodes. By doing this, the analyzer can keep track of the per-node evolution history over time.

When an AST node gets removed from the AST tree (i.e., the corresponding code fragment is deleted from the source code), the node cannot have more evolution history in the future. At that point, the analyzer checks whether the evolution history of that node contains any backtracking instances. Let the evolution history $h \coloneqq v_1 v_2 \dots v_n$, where v_i is the i-th snapshot of the node. A backtracking instance b is then defined as a sublist $v_f \dots v_l$ of h where:

$$f + 1 < l \wedge v_f = v_l \wedge \forall i \in (f, l) \cdot v_f \neq v_i$$

In summary, a backtracking instance is a sub-history of a node where the first and the last snapshots are the same, and all the intermediate snapshots are different from the first snapshot. This indicates that this node digressed from the first version and then backtracked to the first version. Fig. 1 shows an example node history, which contains three backtracking instances. Note that $v_1 \dots v_6$ does not count as backtracking, because there is an intermediate version v_4 which is the same as v_1 and v_6. Throughout the paper, we will formally refer to v_f as the *first version*, v_l as the *last version*, and any v_i $(f < i < l)$ as an *intermediate version* of a backtracking instance. When there is no need to distinguish the first and last version, we will use the term *original version* to refer to either version.

Fig. 2 shows an example history of a statement node that contains a backtracking instance. On the left side are the IDs of the edit operations unique within a single programmer's entire log. Each line shows the snapshot of the node at a given time, and it shows the evolution history of this specific node from top to bottom. The green shaded code indicates newly inserted code, and the pink shaded strikethrough code indicates deleted code compared to the previous version. In this example, there was one backtracking instance [4263, 4629], which means that the normalized snapshot of this node at ID 4263 – the first version – was identical to the one at 4629 – the last version.

The analysis was performed on three different levels of granularity of AST nodes: *statement* level, *block* level, and *type*

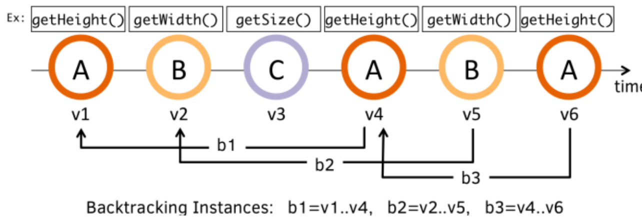


Fig. 1. An example of a node evolution history, which contains three backtracking instances. The node first appeared in the code as "getHeight();" (v1), changed a few times (v2 through v5), and finally ended up back at the original code (v6). The different contents are symbolized as capital letters A, B, and C. There are three backtracking instances in this node history indicated as black backward arrows.

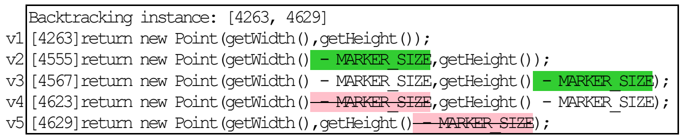
definition level. A type definition is any of a whole class, an interface, or an enum, which is usually a whole Java file with the exceptions of nested type definitions. We analyzed the logs at these levels for two reasons. First, we wanted to know the granularity of backtracking instances. Second, this is needed to detect certain types of backtrackings. For instance, when a statement (s1) is deleted from the code and then the identical statement (s2) is put back at the same position in the future, the statement level analyzer would consider s1 and s2 as separate nodes, and thus not detect this as backtracking. On the other hand, if we run the analysis at the block level, both the deletion of s1 and insertion of s2 would be in the evolution history of the surrounding block, thus the block-level detector would successfully detect this as a backtracking instance. In addition, block level analysis can detect the changes spanning across multiple statements. Similarly, when a code block or an entire method is removed and restored later, or when multiple code blocks are changed together and backtracked, the type-level detector would catch that instance.

Comments in the source code were excluded from the analysis, as we were mainly interested in actual code changes and exploratory programming. However, when the programmer comments out some code and uncomments it (or vice versa), the analyzer still detects this as a backtracking, because it is seen by the analyzer as if the commented out code disappeared and was put back in.

C. Data Preparation & Removing Duplicated Results

Before performing the analysis, we cleaned up the data in order to get more meaningful results. First, we removed all of the minor typo correction instances from the logs using our typo correction detector [2]. Even though typo corrections are backtracking instances by our definition, these are not very interesting in that it is hard to imagine a useful programming tool that helps fixing typos any better than current mechanisms. There were 40,229 edits removed as part of typo corrections, out of a total of 343,685 edits (11.7%).

The second clean-up process that we applied was removing the noise related to the "Rename Refactoring" command of Eclipse. When the rename refactoring is invoked, all the occurrences of the element being renamed (e.g., a variable name) are highlighted directly in the code editor, and they all change together as the user types. When the user confirms the renaming by hitting the enter key, however, Eclipse automatically reverts all the character-level changes made during the renaming process, and turns them into word-level changes so that users can undo the renaming with a single command. Since all the intermediate character-level changes are also logged in the FLUORITE logs, they resulted in many false positives in our backtracking analysis results. Thus, we cleaned these up before analyzing the logs.



```
   Backtracking instance: [4263, 4629]
v1 [4263]return new Point(getWidth(),getHeight());
v2 [4555]return new Point(getWidth() - MARKER_SIZE,getHeight());
v3 [4567]return new Point(getWidth() - MARKER_SIZE,getHeight() - MARKER_SIZE);
v4 [4623]return new Point(getWidth()~~ - MARKER_SIZE~~,getHeight() - MARKER_SIZE);
v5 [4629]return new Point(getWidth(),getHeight()~~ - MARKER_SIZE~~);
```

Fig. 2. An example output of our analyzer, showing the history of a statement node. Each row maps to each version (v1,v2, …, v5). This node contains a single backtracking instance, which is v1…v5. The edit operation IDs were originally 6-digits long (e.g., 184263), but were shortened for brevity.

TABLE II. SUMMARY OF ANALYSIS RESULTS

PID	Group	Time (h)	BI[a]	BI/h[b]	CRR[c]	SR[d]
P0	G0	294.1	2278	7.7	37.5%	10.1%
P1	G1	68.1	961	14.1	6.0%	11.8%
P2	G1	216.2	2450	11.3	26.3%	13.7%
P3	G1	2.6	73	28.4	0.0%	6.8%
P4	G1	64.2	1616	25.2	45.4%	8.1%
P5	G1	13.7	110	8.0	29.1%	17.3%
P6	G1	16.4	164	10.0	8.5%	6.1%
P7	G1	25.3	486	19.2	11.7%	8.4%
P8	G1	29.5	296	10.0	45.6%	10.8%
P9	G1	22.5	380	16.9	36.3%	11.6%
P10	G1	19.5	193	9.9	3.1%	14.5%
P11	G1	22.7	87	3.8	13.8%	10.3%
P12	G1	5.5	65	11.8	13.8%	1.5%
P13	G1	14.0	126	9.0	4.0%	1.6%
P14	G2	5.7	47	8.3	42.6%	6.4%
P15	G2	87.3	622	7.1	19.0%	11.4%
P16	G2	446.0	4179	9.4	4.3%	5.8%
P17	G2	28.0	116	4.1	44.0%	8.6%
P18	G2	21.2	186	8.8	4.8%	4.3%
P19	G3	51.2	605	11.8	2.0%	14.9%
P20	G3	6.2	55	8.9	0.0%	7.3%
Total		1459.9	15095	10.3	20.4%	9.5%

[a] BI: Number of backtracking instances
[b] BI/h: Backtracking instances per hour
[c] CRR: Cross-run backtracking instances rate
[d] SR: Selective backtracking instances rate

We also took extra care to remove duplicate instances appearing in the analysis results. When running the analysis on different levels of granularities (statement < block < type definition), the same backtracking can appear in multiple granularities. For instance, when a statement is changed and immediately backtracked, this instance will appear in all three levels. When counting the backtracking instances at a coarse level of granularity, we excluded all the instances that were also detected at a finer grained level. There can also be duplications within the same level of granularity, because a code block can contain another block, and a type definition can contain a nested type definition inside. We only counted the backtracking instances at the innermost code element.

Finally, because the programming language used was Java, there can also be duplications when a statement contains one or more type definitions or code blocks. In Java, anonymous class instances can be defined and assigned to a variable or passed as a parameter of a method *inline*. We made sure to exclude these types of duplications as well by not counting such statements.

III. RESULTS

This section presents the analysis results, which are summarized in Table II. The backtracking instances were investigated with several specific questions in mind, each of which is explained in the following subsections.

A. Frequency of Backtracking

Overall, the analyzer detected a total of 15,095 backtracking instances within the 1,460 hours of coding activities, which gives an average rate of 10.3 instances per hour. That is, programmers were backtracking every six minutes on average, which is quite frequently considering that this number excludes all the trivial typo correction types of backtrackings. The rate varied across participants (min=3.8/h, max=28.4/h), but *all* participants were backtracking frequently.

B. Size of Backtracking

We measured the size of each backtracking, in terms of how far the intermediate versions digressed from the original version. Minor backtrackings might not need much tool support, but it would probably be very helpful to have better tools for relatively larger backtrackings. To measure this size, we used the Levenshtein distance [3], commonly referred to as the edit distance, between two strings. In a backtracking instance, we calculated the edit distances between the original version and each of the intermediate versions, and took the maximum value as the size of the backtracking. For instance, if an instance had a version history of A→B→C→A, then the size of the instance would be the maximum value of the distance between (A, B) and (A, C). For example, in the earlier example presented in Fig. 2, the backtracking size is 28, which is the edit distance between v_1 and v_3.

Fig. 3 shows the distribution of backtracking sizes. The horizontal axis (which is non-linear) represents the groups of backtracking sizes in ascending order, and the vertical axis shows the total number of backtracking instances in each group. We can see that 1,304 backtrackings were a single character, which were 8.6% of all backtrackings. The most common backtrackings were between 10 and 49 characters. There were also 220 backtrackings that were larger than or equal to 1,000 characters.

Of all the single character backtrackings, we could automatically determine that 36% were performed on variable or method names, 26% on number literals, and 13% on string literals. We randomly sampled 50 instances from each of the larger size categories and manually inspected them to get a better sense of what kinds of backtrackings are represented at the different sizes. The 2-9 and 10-49 groups seem to be dominated by simple parameter/expression changes as in Fig. 2, followed by simple name changes on methods or variables. The majority of 50-99 group seem to be single statement changes, and some instances were about surrounding existing code with control blocks (e.g., `if`, `try-catch`) and reverting it. The 100-499 group instances seem to be mostly adding/removing/modifying multiple statements. The instances with sizes larger than 500 seemed to be significant algorithmic changes, adding or removing multiple methods, and so on.

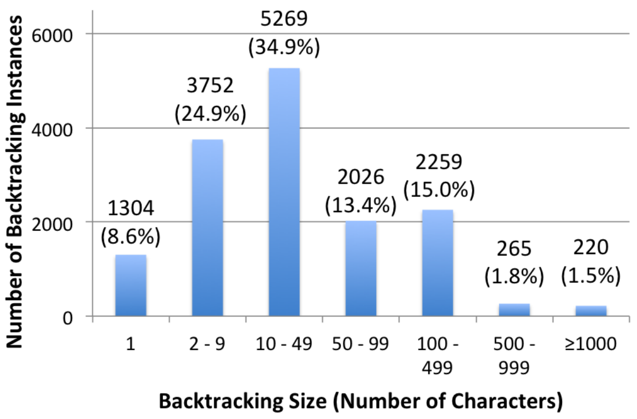


Fig. 3. Distribution of all the detected backtracking sizes

C. *Backtracking Tactics*

Our ultimate goal is to provide useful backtracking tools for programmers. Therefore, it is important to understand how programmers are backtracking now, and to determine whether there is room for improvement. Although we have studied this in our previous survey [1], this time we were able to identify the editor features used for backtracking with actual data. The logs contained not only the code changes but also the editing commands (e.g., copy, undo, etc.), which made it possible to detect what types of commands were used to accomplish the backtracking. We are going to use the term *tactics* to refer to the low-level editor features used for each backtracking, in accordance with prior research [4, 5].

We looked for the editing command that caused all the backward changes of the instance, which are the changes following the intermediate version with the greatest edit distance with the original version (Fig. 4). When all the commands were of same type, we determined that command to be the backtracking tactic. Otherwise, we marked the tactic as *multiple*. We automated this process, but the analyzer could not determine the backtracking tactics for 9.43% of the instances. This happened when the logs did not have enough information, for example when the source code changed outside of Eclipse.

Fig. 5 shows the identified backtracking tactics. The most frequently used backtracking tactic was using the undo and redo commands, constituting 36.63% and 2.57% of all the backtracking instances, respectively. This implies that these instances were already supported by existing editor features. We also noticed that many of the undo commands were invoked repeatedly in sequence. In other words, the participants often used the undo command multiple times in order to revert the source file to an earlier state.

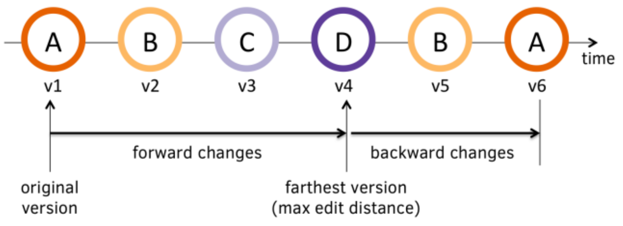


Fig. 4. A backtracking instance illustrated. The analyzer determines the farthest version within each instance, and considers all the changes following the farthest version as backward changes.

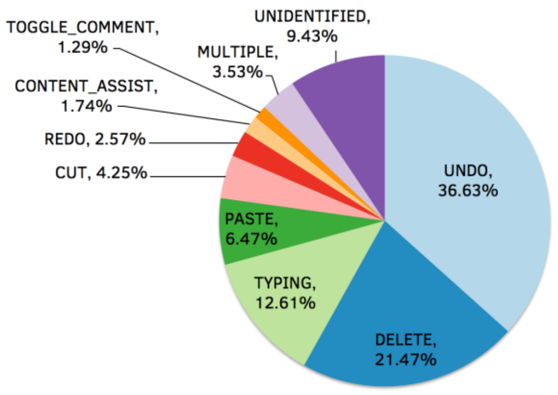


Fig. 5. The identified backtracking tactics

The next most frequently used tactic was deleting some text from the code, constituting 21.47% of all instances. This includes using backspace key (16.33%), delete key (3.74%), delete line command of Eclipse (0.91%), and some combinations of these (0.49%). For these cases, once the code is located, backtracking itself is trivial; the programmer can easily select the code and delete it. Therefore, backtracking tools should help programmers *locate* the right piece of code to be removed, because that is the biggest challenge in this case.

The third most popular tactic was manually typing the desired code, which was used in 12.61% of all instances. We considered the cases with typing and deleting intermixed as typing, since these deletions can be seen as minor corrections happened while typing. This is particularly interesting, because it shows the lack of tool support for restoring deleted code. Manually restoring the deleted code or reverting modified code to the original version relies entirely on the programmers' memory, and thus can be error-prone. Even if the programmer knows exactly what she has to do, manual typing would be less efficient compared to just undoing – even selectively – that piece of code to the desired version.

Cutting (4.25%) and pasting (6.47%) were the next most popular tactics. Of all the backtracking performed by pasting, 41% were just undoing the preceding cut commands, which can be considered as an alternative way of copying the code.

3.53% of the instances were marked as multiple, which means that more than one type of editing commands were used to accomplish the backtracking. The rest of the identified tactics were using content assist features such as code completions and quick fixes (1.74%), and using the toggle comment feature (1.29%).

D. *Cross-Run Backtracking Instances*

In situations such as designing a system or learning an unfamiliar API, programmers must explore and try out multiple alternatives, which has been called exploratory programming [6]. We were interested in how many of the detected backtracking instances were performed as part of exploratory programming. When programmers experiment with code, they make some changes, run the application, and revert the code back to the way it was before if the code does not behave as desired. We checked whether there was an application run command between the first change and the last change of a backtracking instance, which we call *cross-run* instances. The FLUORITE logs contain run commands, such as launching the application under development and running unit tests within Eclipse, which made it possible to count such instances.

The cross-run instance rates are shown in Table II, in the column named "CRR" (cross-run rate). Overall, 20.4% of all instances were cross-run instances. This rate varied among participants, and two of them (P3 and P20) had no cross-run instances at all. Interestingly, the logs from these two participants were relatively short (2.6h and 6.2h, respectively) compared to the other logs, and did not contain any run commands at all. This could mean that these two people ran the application outside Eclipse for some reason, or did not run the application at all. All the other participants had some number of cross-run instances.

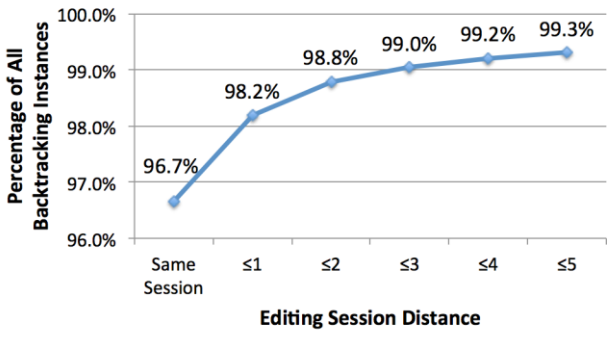


Fig. 6. Cumulative percentage of all backtracking instances with different editing session distances. 96.7% of all backtrackings were performed within the same editing session. 99.0% of all instances have less than or equal to a 3 session distance.

P14 had a 100% cross-run backtracking rate from the statement level analysis. By manually inspecting these six cross-run instances, it was found that all of them were parameter tuning instances, for example adjusting some threshold value from `0.3` to `0.4` and then back to `0.3`.

E. *Cross-Session Backtracking Instances*

In most code editors, the undo feature works only within the same editing session, and users cannot undo the changes made in the past sessions (where a session is from starting Eclipse to exiting). To determine whether it would be useful to make backtracking tools work across sessions, we counted the number of cross-session backtracking instances, where the editing sessions of the first change and the last change were not the same.

The analysis showed that 96.7% of backtrackings were done within the same session, and only 3.3% of all instances were cross-session backtracking. We also measured how many sessions did each instance go back, which is depicted in Fig. 6. It shows that 99.0% of all backtracking was done within 3 editing sessions. In other words, a backtracking tool would work for the most (97.0%) cases with only the history within the same editing session, and providing the history of the last 3 sessions would cover 99.0% of the cases.

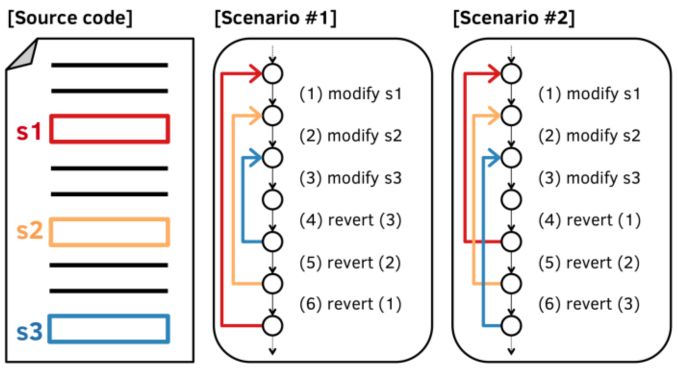


Fig. 7. Two possible backtracking scenarios, whose backtracking instances are *not* selective. The source file has three different statement nodes being affected (s1-s3). Each backtracking scenario has three backtracking instances in each node. Except for the backtracking instance in s3 in scenario #1, all the backtracking instances have some changes to other parts of the same file within their timespan. Nevertheless, these are not selective because the undo command *can* handle both cases.

F. *Selective Backtracking Instances*

We also investigated whether each backtracking instance was *selective* in nature. A backtracking instance is selective when there are edits in the middle of the backtracking that change *other* parts of the same file, that are not backtracked together. We were interested in this, because the restricted linear undo command [7], the conventional undo command used in most editors, cannot handle selective backtracking.

When determining selectiveness, we tried to be as conservative as possible, even when there are intermixed changes to other parts of the code. First, we excluded all the backtracking done by undo/redo commands. In addition, there were other subtle cases that we wanted to exclude. For example, Fig. 7 shows two possible backtracking scenarios in a source file. The source file has three statement nodes, s1, s2, and s3. In both scenarios, the three nodes are modified in turn and they are reverted back to the original version in various orders. We did not want to mark these instances as selective, because the undo command could be used multiple times to handle these cases. The first scenario shows such a case. However, the second scenario is also possible when the user performs backtracking by hand. To exclude these cases, we also looked ahead and checked the changes immediately following the last change of a backtracking instance, to see if those changes are reverting the intermixed changes to the other parts.

The results showed that 9.5% of all instances were selective, as indicated in Table II in the "SR" (selective rate) column. We also investigated the tactics of all the selective backtrackings, as discussed above in Section III.C. 63% of selective backtrackings were performed by manually deleting or typing the desired code, suggesting the need for backtracking tools. We did not find any significant difference between selective and non-selective backtrackings in terms of their sizes (that is, the selective backtracking had small and large sizes in a similar proportion as shown in Fig. 3).

G. *Repeat-Count*

When a certain node comes back to the same version more than once, we kept track of the repeat count of each backtracking instance. For example in Fig. 1, $b1$ and $b2$ are of repeat count 1, but the repeat count of $b3$ is 2, because the node content backtracked to A for the second time.

Fig. 8 shows the repeat counts of all backtracking instances. The blue bars indicate the number of instances in each repeat count group. When a backtracking happens more than once, the first time is counted in the first bar, the second time is counted in the second bar, etc. In other words, each bar is included in the previous bar. Each point on the red line is the next bar value divided by the current bar value, indicating the percentage of the instances that come back to the same state again. For example, 12,430 instances have the repeat count value of 1, and only 15.0% (1,868 out of 12,430) of them come back to the same state after some exploration in the future. While the number of instances dramatically falls as the repeat count increases, the revisiting fraction goes up. This implies that when a node comes back to the same state a few times, it is likely that the node will digress and return again.

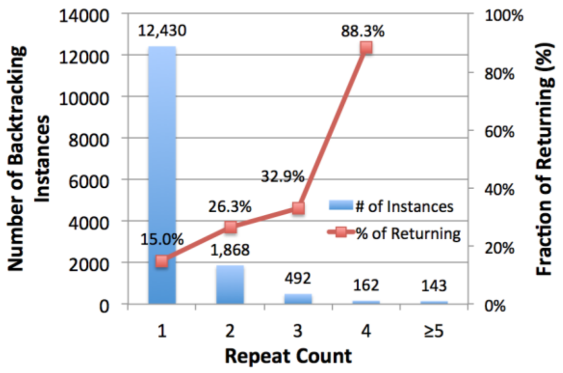


Fig. 8. Repeat counts of all backtracking instances, along with the percentage fraction of revisiting the same state in the future.

One participant (P2) backtracked a statement to the same state 24 times, which was the maximum value of all repeat counts. This was another parameter tuning instance, where she was experimenting with different width value for a line drawn on a canvas. The next biggest repeat count was 9, where a statement that translated a graphical object on the screen was removed and put back again multiple times.

IV. Limitations and Future Work

There are a few limitations of our analysis method. First, our analyzer can only detect *exact* backtracking instances, which has several implications. For instance, when a programmer changes two parts within a statement and reverts only one of them, the analyzer would not be able to catch this backtracking instance because the smallest level of granularity we used was the statement level.

Moreover, all the detected instances are *successful*, and the analyzer cannot detect cases where the programmer intended to backtrack but failed, which we had seen happen in our lab study [1]. If programmers fail to backtrack correctly, it would imply the need for better backtracking tools even more. Our analysis will also miss *near-exact* backtracking instances. For instance, suppose a variable name `fontSize` changes to `rectangleSize`, then to `regionArea`, and finally to `fontArea` in that order (changed parts are underlined). Conceptually, this is backtracking because the front part of the variable name has changed from `font` and then back. We were interested in this, because even a selective undo tool cannot easily handle this because selectively undoing the first change (`fontSize` → `rectangleSize`) has what we call a "region conflict" with the following change which deleted a portion of `rectangle`. Our analysis would not count this case.

All the participants were programming in Java. There may be some backtracking patterns that occur more often when coding in Java, and the statistics presented in this paper may not be generalizable to other programming languages. In addition, if a similar analysis were to be performed for a different programming language, the analyzer might need some language-specific tweaks. For example, we needed to take Java's anonymous class definition into account, in order to filter out duplicated results.

In addition, all the logs were collected within Eclipse, which might have affected the results. The backtracking patterns may vary if programmers are using different code editors or IDEs that provide different code editing features. Nevertheless, considering that most available IDEs provide a similar set of editor commands and all programmers need to do tasks like understanding APIs and determining the right parameters for methods, we believe that the patterns would not be drastically different from what we reported.

All participants, including the professional programmers, were working in academic settings. We do not know whether there would be any significant differences in industrial settings. None of the participants were novices learning to program, and studying novice backtracking behaviors would be an interesting area for future work. We included the first author's own logs (P0 in Table II), which might have biased the results. However, we confirmed that excluding the first author's log does not change the results much. For example, the backtracking rate becomes 11.0/hour when P0 is excluded, compared to 10.3/hour as we reported in Section III.A, and the selective backtracking rate remains the same at about 9%.

There were a few types of information we sought but could not obtain. First, because the log data did not record the version control system (VCS) features used by the programmers such as *commit* and *revert*, we could not tell the fraction of backtracking done by the VCS. If the source code is backtracked using the revert feature of a VCS, our analyzer would still detect the backtracking but mark the tactic as unidentified (see Fig. 5). If we knew which ones were due to the use of a VCS, we would have been able to distinguish the types of backtracking which would be better handled by a VCS and those most suited for in-editor features. This would help us design backtracking tools to work better with existing VCSs.

In addition, we wanted to know whether there were semantic relationships among the backtracking instances. For example, suppose a programmer first renames a method and all the callsites, and then reverts these changes later on. Our analyzer would detect the individual backtracking which occurred in the method definition and the callsites separately. It would be clear that these instances are all related to each other by manually inspecting these instances, but the analyzer could not automatically determine such relationships.

We also wanted to learn how programmers navigate to the backtracking location before they actually perform backtracking changes. However, we could not determine this information partly because there is some missing information we needed in the logs, and also because it requires too much manual investigation. There are diverse ways of navigating in Eclipse, including keyboard keys, mouse clicks and scrolling, and other Eclipse commands such as various searches. The logger did not catch mouse scroll events, and when the users are performing searches, only the initial search command is logged but the following events for clicking on one of the search results to actually jump to the relevant code are not logged, which makes it difficult to analyze the exact navigation command used by the programmers. Improving the analysis method and looking for this missing information remains as future work.

Another direction is to develop actual backtracking tools according to the design implications extracted from this study. We are already working on a backtracking tool called AZURITE

[8], which supports selectively undoing the desired previous edits without undoing the other intermediate changes that the user wants to keep. According to our study results presented here, at least 9.5% of the backtracking was selective, all of which can be handled with AZURITE.

V. RELATED WORK

This study can be seen as a software evolution study performed at a fine-grained level. While mining software repositories [9], a popular software evolution research methodology, works at the commit level, our analysis is performed at the individual code edit level. For our backtracking study, it was necessary to use the fine-grained history, because programmers would often backtrack while experimenting, and the intermediate versions are very unlikely to be captured in version control system histories.

There are several other research tools that capture fine-grained code changes as our FLUORITE does. OperationRecorder [10] and CODINGTRACKER [11] both take the raw text changes as inputs and turns them into AST-level change operations. Our analysis method could be used to analyze the data collected by these other tools, because they also work at the AST level. The data can be used in various ways, and there are other researchers who have analyzed their own fine-grained code change data to extract different information. For example, CODINGTRACKER logs were analyzed by adapting existing data mining techniques [11], which is different from our per-node history keeping approach. The goal of their analysis was also different: they identified 10 previously unknown program transformation patterns. This shows that analyzing fine-grained code change history can be useful in many different ways.

More generally, detailed tool usage data can be used to identify usability problems of specific tools. Akers et al. devised a study method called *backtracking analysis*, which is designed to capture usability problems of graphical creation-oriented programs such as Google SketchUp [12]. To capture richer contextual information, their system automatically captured both the screens of participants and the backtracking events such as undo or erase. Although we are also detecting backtracking events in an IDE, the goal of our work is very different. In their backtracking analysis, backtracking events such as undo or erase are assumed to be indicators of usability problems of the creation-oriented programs. On the other hand, we believe that backtracking events in code editing are natural in exploratory programming, and our goal is to support programmers to backtrack more easily and effectively.

Vakilian et al. collected detailed usage data of Eclipse refactoring tools using their CODINGSPECTATOR tool, and analyzed the data to discover usability problems of the refactoring tools [13]. In their analysis, they detected the situations where the users used the refactoring tools in a way that is not ideal, indicated, for example, by cancellations or undoing of the refactoring commands. Unlike their study, our focus was on backtracking.

VI. CONCLUSION

Backtracking is inevitable in programming, including when programmers are exploring a design space, experimenting with different options, or just make a mistake. Using the abstract syntax tree node history tracking analysis, we analyzed the fine-grained code change logs to detect interesting backtracking-related information. We confirmed that programmers are in fact backtracking a lot, and there are backtracking situations not very well supported by existing programming tools. We believe that providing better tools will help programmers to perform their daily backtracking tasks more effectively, and thus improving their productivity.

We also believe that our analysis technique may have applications beyond detecting backtracking, and other researchers could benefit from analyzing fine-grained code change patterns to better understand programmers' coding practices and to provide useful tools for programmers.

ACKNOWLEDGMENT

We thank all the developers who participated in our study. We also thank Jonathan Aldrich, Christian Kästner, Emerson Murphy-Hill, the exploratory programming group (exploratory-programming.org), and the anonymous reviewers for helping us improving the early drafts of this paper. Funding for this research comes in part from the Korea Foundation for Advanced Studies (KFAS) and in part from NSF grant IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of KFAS or the National Science Foundation.

REFERENCES

[1] Y. Yoon and B. A. Myers, "An Exploratory Study of Backtracking Strategies Used by Developers," *Proc. CHASE 2012*, pp. 138-144.

[2] Y. Yoon and B. A. Myers, "Capturing and Analyzing Low-Level Events from the Code Editor," *Proc. PLATEAU 2011*, pp. 25-30.

[3] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet physics doklady*, vol. 10, 1966, p. 707.

[4] M. J. Bates, "Where should the person stop and the information search interface start?," *Info. Processing & Mgmnt*, vol. 26, 1990, pp. 575-591.

[5] V. I. Grigoreanu, M. M. Burnett, and G. G. Robertson, "A Strategy-Centric Approach to the Design of End-User Debugging Tools," *Proc. CHI 2010*, pp. 713-722.

[6] D. W. Sandberg, "Smalltalk and Exploratory Programming," *SIGPLAN Notices*, vol. 23, 1988, pp. 85-92.

[7] T. Berlage, "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects," *TOCHI*, vol. 1, 1994, pp. 269-294.

[8] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of Fine-Grained Code Change History," *Proc. VL/HCC 2013*, pp. 119-126.

[9] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, 2007, pp. 77-131.

[10] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," *Proc. MSR 2008*, pp. 31-34.

[11] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining Fine-Grained Code Changes to Detect Unknown Change Patterns," *Proc. ICSE 2014*, in press.

[12] D. Akers, R. Jeffries, M. Simpson, and T. Winograd, "Backtracking Events as Indicators of Usability Problems in Creation-Oriented Applications," *TOCHI*, vol. 19, 2012, pp. 1-40.

[13] M. Vakilian and R. E. Johnson, "Alternate Refactoring Paths Reveal Usability Problems," *Proc. ICSE 2014*, in press.