Snaps: A Tool for Understanding Students in Large Computer Science Classes

Itai Smith

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-118
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-118.html

May 14, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Snaps: A Tool for Understanding Students in Large Computer Science Classes

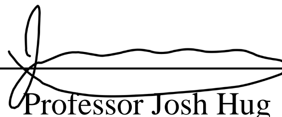
by Itai Smith

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Josh Hug
Research Advisor

14-May-2021

05/14/2021

* * * * * * *

Professor John DeNero
Second Reader

05/14/2021

05/14/2021

Snaps: A Tool for Understanding Students in Large Computer Science Classes

Itai Smith
University of California, Berkeley
iasmith@berkeley.edu

ABSTRACT

As computer science courses continue to grow in size, instructors inevitably rely more on automatic assignment grading and asynchronous modes of instruction, reducing the amount of direct interaction with students. As a result, instructors' have limited ability to effectively temperature check their students, identify students who are falling behind in a timely manner, and make informed decisions about the effectiveness of assignments.

In this paper, we present Snaps, a tool for collecting intermediate snapshots of student work on programming assignments, for any IntelliJ based IDE. We also present how we used the snapshots collected by this tool in UC Berkeley's CS 2 course to identify struggling students, assess the workload administered to students, and evaluate the effectiveness of course design choices.

1 INTRODUCTION

In recent years, universities across the country have seen a tremendous increase in the number of students pursuing Computer Science. This increase is felt most dramatically in introductory classes such as CS 1 and CS 2, which are often offered to all interested undergraduates. In UC Berkeley, the introductory courses, CS 61A and CS 61B, peaked at 2,000 and 1,600 enrolled students respectively in the past 2 years [1].

As teaching staff are faced with limited resources, these large enrollment numbers lead many CS courses to rely on auto-gradable assignments and asynchronous offerings of lecture and section content. As a result, instructors are mostly blind to students' development process and can only temperature check the class after final submissions of assignments or when exam scores become available [2].

Under these circumstances, instructors' ability to monitor student progress is limited. They are unable, for example, to detect misconceptions that thwart students when completing assignments, which will eventually lead to poor performance on exams. Additionally, it becomes difficult to assess the workload and the amount of time spent by different students on each assignment. These issues are particularly pronounced in courses that administer large programming assignments that span multiple weeks.

Moreover, the reality of large CS courses is most disadvantageous to students who are members of underrepresented communities in CS. These students are less likely to ask for help, and are more likely to be overwhelmed by large programming assignments and the barriers for seeking help in large classes [3].

To combat these issues, we developed Snaps, a tool that allows instructors to collect snapshots of the intermediate steps students take as they complete programming assignments. The data produced by this tool can then be used to reason about students' working habits, and extract useful information about the paths each student explores from starting an assignment to completion.

In this paper, we describe how Snaps was developed and used in CS 61B, UC Berkeley's Data Structures course (CS 2), and the insights we were able to gain from using the Snaps tool in the academic year 2020-2021.

2 RELATED WORK

The Snaps tool was inspired by the work of Yan et al. [4], who developed TMOSS, a system that captures intermediate snapshots of students' code for enhanced plagiarism detection. In TMOSS, snapshots were used to gain evidence for excessive collaboration. With the intermediate snapshots of the coding process available, instructors could detect whether one student copied code from another, or from an online resource, for example.

Our original goal was to create a similar tool for enhanced plagiarism detection for CS 61B, which uses the JetBrains' IntelliJ IDE, as TMOSS was developed for the Eclipse IDE. Instead, we decided to take a more general approach, and provide a framework for extracting useful information about students' working habits from the intermediate snapshots. We believe the data collected by our tool would assist instructors in scaling CS education and gain insight into student progress in their courses. Additionally, the tool we developed would work on other JetBrains IDEs, and thus can be used in courses taught in programming languages other than Java [5].

3 SNAPS

The Snaps tool was developed for CS 61B at UC Berkeley, and was designed to integrate as seamlessly as possible into the existing workflow that students follow in the course. In this section, we describe the structure of CS 61B, the Snaps tool and its integration into the course, and finally, some results we were able to gather from the data, and how they helped us evaluate different components of the course.

3.1 CS 61B

CS 61B is UC Berkeley's Data Structures class, and is the second course in the lower division introductory CS series. The three major populations of students enrolling in this course are those pursuing a BS in Electrical Engineering and Computer Science, BA in Computer Science, and BA in Data Science. CS 61B is also a prerequisite course for virtually all upper division CS and Data Science courses. We conducted the study described in this paper during two offerings of CS 61B. In Fall 2020 (1062 students) and in Spring 2021 (1542 students).

Throughout the semester, CS 61B students complete weekly lab assignments, in which they implement data structures and algorithms introduced in lecture. These assignments don't require complete correctness for full credit, and are designed to be completed in approximately two hours. Students also complete 4 large software engineering projects that span multiple weeks. Usually, the last two projects involve a design component.

The data in this paper was collected in Spring 2021 [6], in which the course projects were:

1. 2048 - students were given the skeleton code for a Java based 2048 game, and had to implement the logic for merging blocks on the board according to the game's defined behavior.
2. Data Structures - students were given the specification for the Deque interface, and were tasked to provide a linked list and an array based implementation of the interface with no skeleton code provided. Students were also expected to write their own tests to verify the correctness of their implementations.
3. Gitlet - students implemented a lite version of the Git version control system. The course staff provided detailed descriptions of the commands Gitlet should support and minor suggestions for the classes students should use in their program, but otherwise the entire design of the program was left to the students' discretion.
4. BYOW - given a graphic rendering engine, students implemented a program for generating random explorable worlds, supporting user keyboard interaction. No skeleton code was provided for the world generation engine and students came up with their own designs.

At the beginning of the semester, the course staff sets up Git repositories, on which students complete their assignments. Students also add a separate Git repository as a remote, from which they pull the skeleton code for the different assignments. CS 61B is taught in Java, and students use the intelliJ IDE to complete their assignments. During the first lab section of the semester, students are instructed to clone their personal repository, learn basic Git commands for interacting with it and with the skeleton repository, and install intelliJ on their personal machines.

From our experience teaching an introductory course of the scale of CS 61B over multiple semesters, we identified three major challenges, which we set to combat using our Snaps tool.

3.1.1 The Workload Mystery. Evidently, CS 61B is a coding heavy course and students often report that they spend well over the university's recommended number of hours for a 4-unit course (6 hours of outside lecture or section work) [7]. Additionally, since CS 61B has a grade requirement for declaring the CS major in Berkeley, some students often report that the workload of the course causes them considerable stress. Hence,

these student anecdotes pose serious concerns around the workload administered by the instructor.

Assuming that the majority of time students spend in CS 61B is dedicated to writing code, this challenge is most obviously solved by the Snaps tool. Given all the snapshots of the intermediate steps students take when completing programming assignments, we can confidently determine how much time students are spending on CS 61B, and adjust the workload accordingly if necessary. Additionally, being able to identify the students who are more burdened by the course's workload, can allow us to target them for supporting resources such as priority in office hours or extra tutorials offered by the course staff.

3.1.2 The Unknown Paths to Success. Every semester we see students with a strong background in programming and those who achieved higher grades in the prerequisite course (CS 1, CS 61A in Berkeley), perform better in CS 61B. Beyond these somewhat obvious dry metrics, we don't have much insight into what makes students successful in our class, mainly due to the course's large population and limited interaction with students during their development process. We hypothesize that the data collected by the Snaps tool may help us uncover what are the habits of successful students and what helps them stand out. We hope to answer questions like: do students who test their code perform better? How early do successful students begin assignments? And more. We also believe that answering these questions will help us inform our syllabus design, and to put more students on their path to success.

3.1.3 The Perils of Auto-Complete. The different coding assignments in CS 61B (labs and projects) are designed to give students hands-on experience with the material introduced in lecture. And while the absolute majority of students are able to get full credit on the coding assignments, many are unable to perform as well on exams. Specifically, on exams, students are unable to demonstrate understanding of concepts that were meant to be reinforced by the coding assignments.

We believe that intelliJ, the IDE used by students, plays a role here. A simple example of this case can be illustrated with Java interfaces. Many students would lose points on an exam for trying to instantiate an interface

```
List<Integer> list = new List<>()
```

However, if they wrote the same line in intelliJ, the IDE would prompt a compiler error, and suggest to modify the line to something like

```
List<Integer> list = new ArrayList<>()
```

Under the pressure of finishing an assignment, students are likely to accept this suggestion without recognizing their misconception.

Given visibility to students' development process with the Snaps tool, we believe that we can help students identify such misconceptions, and assure that the learning goals of our coding assignments are met.

3.2 The Snaps Plugin

In order to capture the intermediate steps taken by students in programming assignments, we developed a plugin for intelliJ that interacts with the files currently open on a student's editor. The plugin listens to "save" events that are accessible by the intelliJ API [8]. A save event occurs when a student initiates a manual save, an autosave occurs (every few seconds or so, if changes are made to a file), and when a student compiles code. When a save event occurs, the plugin initiates the snapshot taking procedure.

The snapshots taken by the plugin are Git commits. When the snapshot procedure is invoked, the plugin copies the contents of all the open files in intelliJ into a separate Git repository, the "Snaps repository", stages the files, and creates a commit. The commit message contains metadata about that snapshot, like what files were included. If the files open in intelliJ don't exist yet in the Snaps repository, the plugin will create them. The plugin will only try to take snapshots if the current working directory is a CS 61B student repository.

We chose Git as the snapshot taking and storing mechanism to minimize the logic the plugin has to handle, and thus have minimal to no effect on the student experience when writing code [9]. For example, the plugin can just attempt to commit all open files, without keeping track of what changes were made. Git will then reject any files to which no changes were made before making a new commit [10]. Additionally, Git offers many useful features for future data analysis, like navigating between snapshots, generating diffs between snapshots, running autograder tests on past versions, and more. Finally, since intelliJ plugins can be written in Java, we were able to use JGit, a lightweight

Java library implementing Git, to easily interact with Git repositories programmatically [11].

The downside of using a separate Git repository for our snapshots is that the plugin needs to become aware of its location on the student's machine. We achieve this through having students set an environment variable, SNAPS_REPO, with the path to their Snaps repository. The plugin queries the value of this environment variable upon installation and caches it for future runs.

The installation of the Snaps plugin was integrated into the regular set up process for CS 61B. In the traditional set up, students clone their CS 61B student repository, and download some intelliJ plugins used in the course, like our custom style checker. The plugins are installed through JetBrains' plugins marketplace [12]. In the new set up, students just need to clone an additional repository, and install an additional plugin. They then follow instructions to set their SNAPS_REPO environment variable, which is the only major difference from the aforementioned traditional set up. Finally, after completing an introductory assignment, they push the commits from their Snaps repository, and run an autograder test that verifies that snapshots were made successfully.

In practice, we found that the additional step of manually defining the SNAPS_REPO environment variable burdened the installation process and was prone to errors. The installation process occurs at the beginning of the semester, when students are not yet comfortable interacting with their terminal. Furthermore, instructions for setting environment variables vary between operating systems and shell types. We are currently exploring options for students to provide the plugin with the path to their Snaps repository through intelliJ's user interface.

In Fall 2020, we rolled out the plugin to CS 61B students and monitored its performance. We wanted to verify that the Snaps repository's size remains acceptable and that the plugin does not affect the user experience on intelliJ. At the end of the semester, we found that Snaps repositories averaged at 1.5 MB on disk. Students also reported that they experienced no latency issues while writing code in intelliJ, when the plugin was enabled.

We also found that the intermediate snapshots were taken at a very fine granularity. For example, figures 1 through 4 show snapshots of a student starting to work on a constructor in one of Project 1's classes. In this sequence of events, the student is attempting to initialize an array of a generic type in Java. In Figure 1, we can see that the student is starting their attempt to initialize the `items` instance variable, which was declared as an array of a generic type. In Figure 2, the student is preparing to cast the initialization as a generic type array to agree with the declaration. In Figure 3, the student erroneously uses parentheses, `()`, instead of brackets, `[]`, to initialize a Java array. Finally, in Figure 4, the student corrects their error and successfully initializes the instance variable.

To get the snapshots from students' local Snaps repositories, we asked them to push their commits at 4 checkpoints throughout the semester, usually after one of the large projects was due. We set up a server with all the Snaps repositories cloned, and pulled the commits on our end after each such checkpoint. We also developed a set of modules for extracting data of interest from the Snaps repositories in the form of CSV files. For example, instructors can provide a list of assignments to the `AssignmentTimes` module to get a breakdown of how many minutes, over how many days, each student spent on each assignment.

Finally, while the Snaps plugin was designed to work with IntelliJ, it can be installed and become functional for any IntelliJ-based IDE, such as PyCharm (for Python), CLion (for C), etc [5]. Hence, courses which use programming languages other than Java can also use Snaps to collect data from students.

4 RESULTS

4.1 Course Workload

The first question we were eager to answer was whether the CS 61B workload matches the university guidelines for a 4-unit course. We assume that students spend 3 hours a week watching lecture, 2 hours attending a lab section and completing the lab assignments, and 1 hour a week attending a discussion section. Under these assumptions, they should spend around 6 hours a week working on the programming projects.

Using the Snaps data, we tallied the number of minutes a student spends writing code in intelliJ for snapshots made in each week. We iterate over the commits in each Snaps repository, and find gaps of 10 minutes or more between two commits. Once such a gap is found, we add the total number of minutes counted so far to a running counter, excluding the gap time. For example,

```
2 proj1/deque/ArrayDeque.java
@@ -5,6 +5,6 @@
5   5       private int size;
6   6
7   7       public ArrayDeque() {
8     -         items
    8 +         items = ()
9   9       }
10  10  }
```

Figure 1: First student snapshot. Starting to initialize the items array.

```
2 proj1/deque/ArrayDeque.java
@@ -5,6 +5,6 @@
5   5       private int size;
6   6
7   7       public ArrayDeque() {
8     -         items = ()
    8 +         items = (T[] )
9   9       }
10  10  }
```

Figure 2: Second student snapshot. Preparing to cast an Object array as a generic type array.

```
2 proj1/deque/ArrayDeque.java
@@ -5,6 +5,6 @@
5   5       private int size;
6   6
7   7       public ArrayDeque() {
8     -         items = (T[] )
    8 +         items = (T[]) new Object()
9   9       }
10  10  }
```

Figure 3: Third student snapshot. Attempts to initialize an array of Objects, but incorrectly initializes an Object.

```
3 proj1/deque/ArrayDeque.java
@@ -5,6 +5,7 @@
5   5       private int size;
6   6
7   7       public ArrayDeque() {
8     -         items = (T[]) new Object()
    8 +         items = (T[]) new Object[8];
    9 +
9   10      }
10  11  }
```

Figure 4: Fourth student snapshot. Rectifies the error from the previous snapshot, using brackets to initialize the array properly.

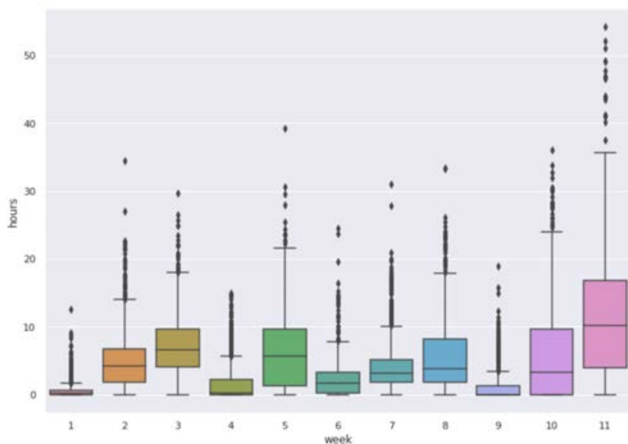


Figure 5: Times spent by students in intelliJ by week.

say we see a sequence of 100 snapshots, with the first starting at 2:00:00, second at 2:00:10 PM, etc., with the last snapshot at 2:10:00 PM. We so far counted 10 minutes. If the 101st snapshot was committed at 2:30:00 PM, we add 10 minutes to the current total time tallied and reset the counter. This procedure essentially identifies continuous time periods in which a student was writing code, and sums them up per week, or per assignment.

The threshold for identifying a continuous coding session, 10 minutes, was chosen somewhat arbitrarily. However, we did not find distinguishable differences in our data when experimenting with lower thresholds.

The box plot in Figure 5 shows how many hours students were spending coding in intelliJ every week. For context, Project 0 spanned weeks 1 and 2. Project 1 spanned weeks 3 to 5, with Midterm 1 taking place in week 4. Project 2 spanned weeks 5 to 11, with an extra credit opportunity due during week 8.

Overall, across all weeks the mean time students spent coding on intelliJ is 4.6 hours per week, the median is 4.3 hours, the first quartile is at 3.2 hours and the third is at 6 hours. This data suggests that overall, the workload we administered matches the amount recommended by the university, assuming that beyond coding time, students also allocate time for reading the specs of assignments, sketching solutions, reviewing material, etc.

Often, instructors' perspective on course workload gets skewed since they interact most frequently with students from extreme ends of the spectrum, either those who find the class too easy, or those who find it too challenging. The availability of the Snaps data was very reassuring, as it showed that overall, the course's workload is manageable for most students.

The exception is evident during week 11, when the course's large design project, Gitlet, was due. This provided evidence that either syllabus changes are needed to incentivize students to start the project earlier, or that the project is inappropriate in the course's current format.

In a similar manner, we were also able to use Snaps data to get the time and the number of days students spent on each assignment. Table 1 shows the statistics for lab completion times in hours. Again, we were able to verify that lab assignments overall meet their allotted 2 hour limit. The exception to this rule in Spring 2021 was Lab 6, which was a preparatory assignment for Project 2, Gitlet. In this assignment students learned how to use Java to interact with the file system, and about the concept of persistence. We assume the lab took students more time to complete as it featured concepts not formally introduced in lecture.

Lab 4 and Lab 5 show relatively low completion times, since they were short non-coding assignments. Lab 4 was mostly an exercise in Git, with a short debugging portion completed on intelliJ. In Lab 5, TAs presented the staff solution to Project 1, and students had to answer a few questions reflecting on their solution, comparing it with that presented in lab. Some students chose to write their answers using intelliJ.

Table 2 lists the completion time statistics for each project. Again, it is clear that students spent an abnormal amount of time on Project 2.

Table 3 shows the amount of days students invested in each project. Days are counted from the first snapshot in which a student edited one of the skeleton files, to the last snapshot containing files under the assignment's directory. With the exception of Project 0, we can see that the course staff has been successful in incentivizing students to start the projects early in some capacity, as students made use of almost all the days available to them.

In fact, Project 0 may be credited for this desired student behavior. In previous semesters, Project 0 was a simple exercise in object oriented programming, designed mainly to familiarize students with Java syntax. In Spring 2021, we decided to change Project 0 to include a more serious algorithmic challenge early in the

Table 1: Lab assignment completion times in hours

Lab	Mean	25%	50%	75%
Lab 1	0.36	0.12	0.23	0.48
Lab 2	1.16	0.78	1.05	1.4
Lab 3	2.25	1.33	1.93	2.74
Lab 4	0.34	0.15	0.28	0.45
Lab 5	0.39	0.23	0.35	0.48
Lab 6	2.53	1.23	2.15	3.5
Lab7	1.92	1.28	1.8	2.38
Lab 8	2.17	1.25	1.97	2.68
Average	1.39	0.8	1.22	1.77

Table 2: Project completion times in hours

Project	Mean	25%	50%	75%
Project 0	5.2	2.76	4.23	6.38
Project 1	14.32	9.58	13.12	18.03
Project 2	32.18	20.03	31.12	42.27

semester, in order to set students' expectation for the rigour of the course in following weeks. In a survey conducted after the project's due date, many students noted that they wish they started earlier. The Snaps data shows that overall they corrected their behavior for the subsequent projects. As a teaching staff, we were pleased to see this changed behavior evident from the Snaps data.

Table 3: Days spent on projects

Project	Days Provided	Mean	25%	50%	75%
Project 0	9	2	1	1	3
Project 1	15	11.93	11	13	14
Project 2	42	35.01	34	37	38

4.2 Paths to Success

A common predictor of success in CS 61B is student's grade in the preceding introductory course, CS 61A [13]. Beyond this crude anecdotal detail, we don't have a good understanding of what leads different kinds of students to success. To see what story the Snaps data tells in this regard, we trained a decision tree classifier using the Snaps data along with demographic and experience information that students self reported. Since the hierarchy of features used in a decision tree reflects their importance [14], we were curious to discover what were the factors of success according to our collected data.

At the beginning of the semester, we released a pre-semester survey, in which students self reported information such as CS 61A grade (if taken), gender, perceived programming level, knowledge of Java, and more. Together with the data collected from Snaps about completion time of assignments and weekly workload, we trained a decision tree classifier. To provide a label to each student, we divided the student population into 10 groups, 0-9, based on their current grade in the course. Students in group 9 were those with the highest grades. It is important to note that most variance in grades existed in exam scores (Midterm 1 and Midterm 2), and Project 2 grades, which was an abnormally challenging project.

Figure 6 shows a snippet of the graphical representation of our classifier. As expected, the first splitting feature is students' CS 61A grade. However, the paths down the tree tell an interesting story. From the figure, consider high performing students (class 8), who got a grade of B, or below in CS 61A. These make up 20% of the students in class 8.

(1) They spent 34 hours or less on Project 2, less than the class average of 35 hours.

Then, these students are split by their working hours during week 2, when Project 0 was due. Students in the first group:

(2) worked 5.6 hours or less during week 2, suggesting that they completed the project early during the first week of classes.
(3) invested an hour or less on Lab 6, the preparatory assignment for Project 2. However, in week 7 they worked more than some of their peers, perhaps getting a head start on Project 2, and working toward completing the extra credit opportunity for Project 2, due in week 8.

Students in the second group:

(2) worked more than 5.6 hours week 2, suggesting that they started Project 0 late, and submitted it near the deadline.

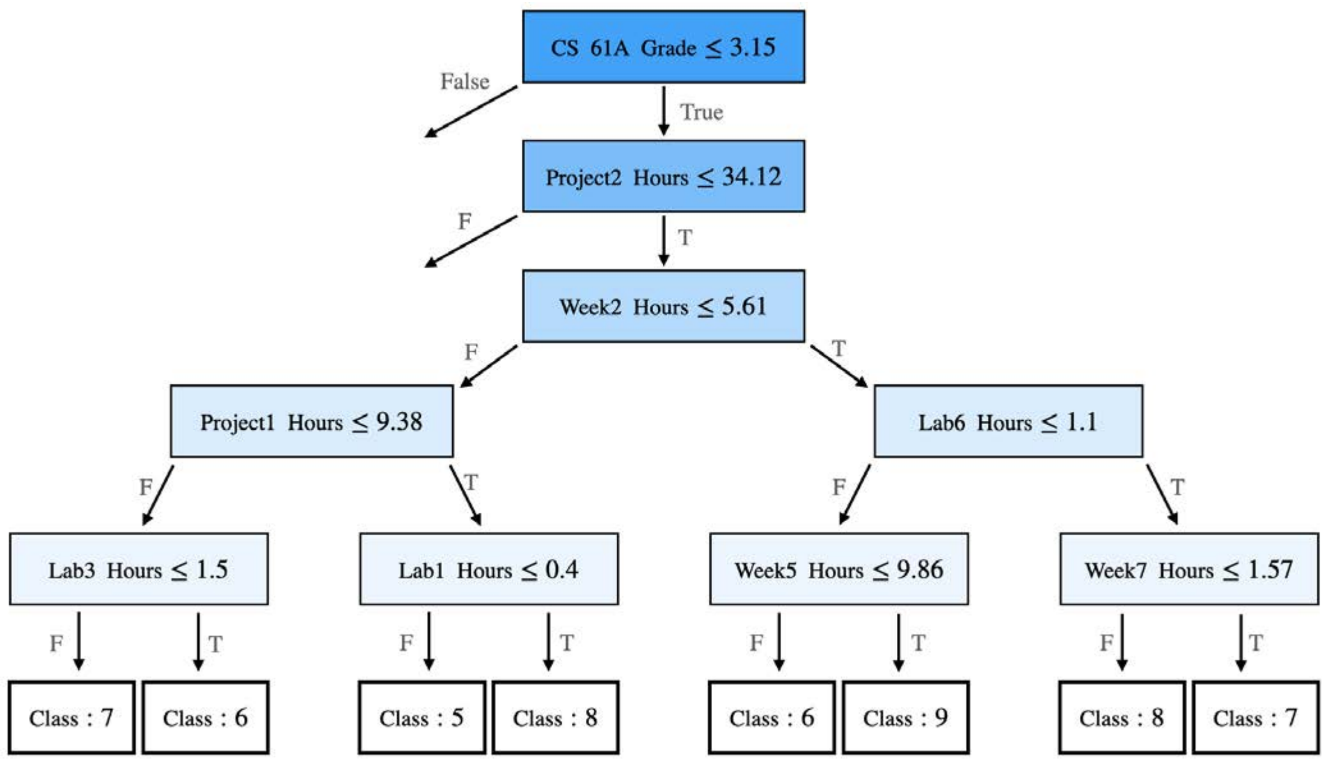


Figure 6: Snippet of the graphical representation of a decision tree classifier trained on Snaps data.

(3) worked on Project 1 for about 9 hours or less, which was less than the class average of 14.32 hours.
(4) invested 25 minutes or less on Lab 1, suggesting that they were stronger programmers coming into CS 61B than their peers (who are in class 5 under the same previous splits). Lab 1 included a short assignment, requiring students to define a Java method to determine whether an input year is a leap year.

While these are mostly conjectures based on our data and anecdotal evidence, we believe that this type of usage of Snaps data can help uncover what makes different students in a large course successful. Additionally, having access to this information during the semester, can help instructors understand why certain students are falling behind and encourage them to improve their studying habits or provide additional resources.

5 FUTURE WORK

For the past year, we invested our efforts in building the Snaps plugin, the infrastructure to support the data collection, and used the data to inform our decisions around workload adjustment. However, as we developed Snaps we recognized that there are many opportunities to expand its functionality and the use of the rich data collected from students.

5.1 Live Streams

This past year, in order to get the snapshots from students we instructed them to push the commits in their Snaps repositories at four designated checkpoints. However, we initially intended on having the Snaps plugin automatically push the commits it creates in the background. In order to do so, the plugin would need access to students' authentication credentials for Github, which we use to host the CS 61B repositories. We felt that requiring students to provide these credentials would be too invasive, as they may be regular users of Github, outside of their CS 61B work. Additionally, there was no easy way to circumvent this requirement without considerably interrupting the course's existing infrastructure.

We believe that given more research and/or consideration of diverting the course's infrastructure away from Github, we can enable live streams of snapshots from students. This will allow the teaching staff to take action in a timely manner based on Snaps data. For example, if a student spends excessive time on an assignment, or exhibits multiple misconceptions in their code, they could be targeted for additional support. While similar action can be taken based on the Snaps data that comes after projects' completion, targeting struggling students early can aid in minimizing frustration and stress, and provide timely help for those who need it the most.

5.2 Plagiarism Detection

In the original work by Yan et al. [4], TMOSS, a snapshotting tool was used to improve the detection of excessive collaboration between students. For example, if a student copied code from another student, or from a solution found online, there would be a snapshot of the copied work, before the student adds "noise" to evade traditional plagiarism detectors. In TMOSS, every snapshot from every student would be compared to all final submissions. The researchers showed that using intermediate snapshots for plagiarism detection, yields nearly twice as many hits compared with traditional methods, which only compare final submissions. The traditional tool, MOSS, found 35 students out of 1420, and TMOSS, using intermediate snapshots, found 61 students. However, while MOSS ran in 0.03 hours, TMOSS ran for 9.77 hours.

We believe that since the Snaps snapshots are taken at a very high frequency, the runtime for plagiarism detection can be improved significantly, by targeting snapshots where the amount of code added to the editor by a student is large. Such snapshots would suggest that code was copied and pasted. From our inspection, we have seen that snapshots taken by Snaps mostly showcase line-by-line modifications made by students, and snapshots with many lines of code added can be easily detected. For example, consider the snapshot in Figure 7, taken after intelliJ automatically implemented the Deque interface methods for a student. Compared with the snapshots included above, it is clear that dozens of lines of code were added all at once.

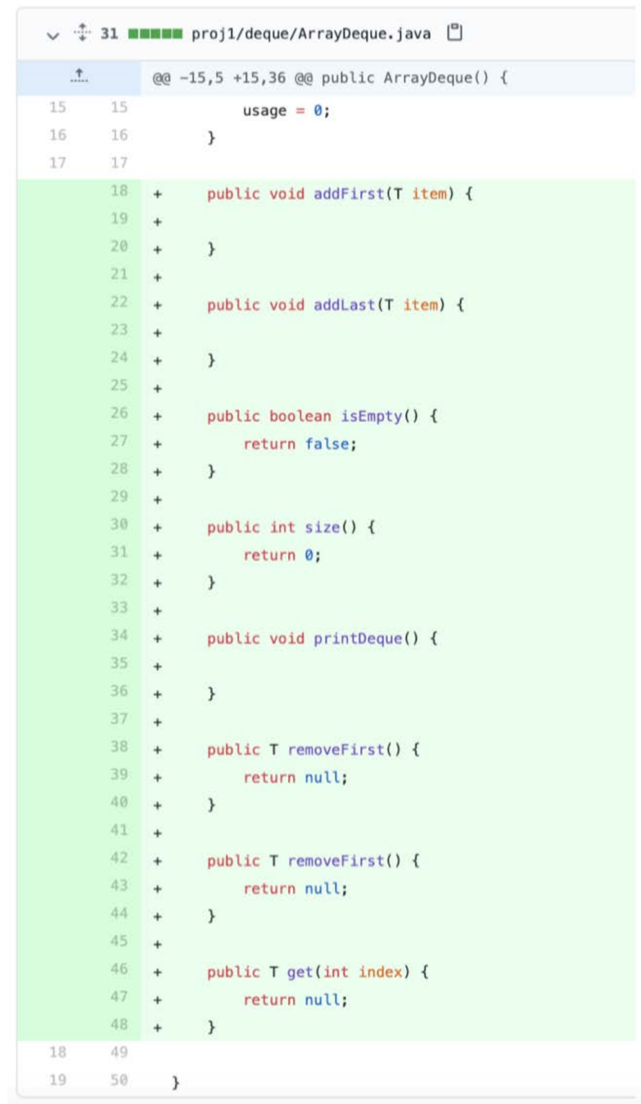


Figure 7: A snapshot showing a large amount of code added at once.

5.3 Other Uses of Snaps Data

In the past year, we performed bulk analysis on Snaps data as a first step in exploring the usefulness of the snapshots. However, we believe that there are many exciting ways to analyze Snaps data to uncover what makes students successful in CS 61B. Some ideas to explore are:

- Analyzing students' code for semantic style. De Ruvo et al. [15] introduced semantic style indicators that may be manifestations of poor knowledge of some programming concepts. Using Snaps

data, we can explore how many of these indicators appear and are rectified over time. The appearance of these indicators can also help us understand what course concepts we don't teach effectively.
- Understanding student methods for testing code. From our analysis of the Snaps data, we couldn't find strong correlation between the amount of JUnit tests (the formal testing method taught in CS 61B) written by students and performance in the course. This suggests that the effectiveness of our current teaching of JUnit testing is questionable, and that students test their code using methods like ad-hoc testing, print statements, and others. Further investigation into students' testing methods can help us improve this important part of our syllabus.
- Uncover more metrics for success. In our results from this study, we demonstrated how Snaps data can be used to obtain a better understanding of the study habits of different types of students in CS 61B. We believe that by extracting more metrics from the Snaps data, we can keep uncovering the parameters that help different students achieve success.

6 CONCLUSION

As computer science classes continue to grow in size, being able to reason about student behavior in large classes becomes a difficult challenge. In this paper we presented Snaps, a tool for capturing the intermediate steps students take when completing programming assignments. We have also shown how we used Snaps data to assess the effectiveness of syllabus changes, and inform future design choices. Finally, we have discussed how Snaps data can be used to identify struggling students, uncover what factors contribute to student success, and lay out a path for more efficient and effective plagiarism detection.

We are also excited to keep expanding Snaps' functionality to allow real time feedback for students. We believe that by catching misconceptions on time, or detecting students burdened by our workload, we can put many more students on the path to success.

7 ACKNOWLEDGEMENTS

This project would not have been possible without the incredible mentorship, support, and friendships I was fortunate to find in my 5 years in UC Berkeley. I would like to thank

- Josh Hug, my advisor, for all his guidance, invaluable mentorship, and encouragement throughout the past 4 years.
- Paul Hilfinger, teaching with whom was a great honor and a great learning experience.
- Matthew Owen, Michelle Hwang, Connor Lafferty, Omar Khan, and Henry Maier, for their friendship and support throughout the completion of this project.
- The CS 61B staff, for their hard work and dedication to students, and allowing Snaps to become a reality.

8 REFERENCES

[1] Berkeley Student Information Systems, https://sis.berkeley.edu

[2] Jeffrey Forbes, David J. Malan, Heather Pon-Barry, Stuart Reges, and Mehran Sahami. 2017. Scaling Introductory Courses Using Undergraduate Teaching Assistants. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). Association for Computing Machinery, New York, NY, USA, 657–658. DOI:https://doi.org/10.1145/3017680.3017694

[3] Nathaniel Titterton, & M. Clancy (2007). Adding Some Lab Time is Good, Adding More Must be Better: the Benefits and Barriers to Lab-Centric Courses. In FECS.

[4] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18). Association for Computing Machinery, New York, NY, USA, 110–115. DOI:https://doi.org/10.1145/3159450.3159490

[5] IDEs based on the IntelliJ platform, IntelliJ Platform SDK, JetBrains Docs. https://plugins.jetbrains.com/docs/intellij/intellij-platform.html#ides-based-on-the-intellij-platform

[6] CS 61B at Berkeley, Spring 2021, https://sp21.datastructur.es.

[7] Designation of Unit Value, Berkeley Academic Senate, https://academic-senate.berkeley.edu/coci-handbook/2.3.1

[8] File Document Manager Listener, IntelliJ API, https://upsource.jetbrains.com/idea-ce/file/idea-ce-7b9b8cc138bbd90aec26433f82cd2c6838694003/platform/platform-api/src/com/intellij/openapi/fileEditor/FileDocumentManagerListener.java
[9] Optimizing Performance, IntelliJ Platform SDK, JetBrains Docs, https://plugins.jetbrains.com/docs/intellij/performance.html.
[10] Recording Changes to the Repository, Git Documentation, https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository.
[11] The JGit Project, The Eclipse Foundation, https://www.eclipse.org/jgit.
[12] JetBrains Plugin Marketplace, https://plugins.jetbrains.com.
[13] Allen Guo .Analysis of Factors and Interventions Relating to Student Performance in CS1 and CS2. EECS Department, University of California, Berkeley. May 2020. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-22.html
[14] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A Survey of Methods for Explaining Black Box Models. ACM Comput. Surv. 51, 5, Article 93 (January 2019), 42 pages. DOI: https://doi.org/10.1145/3236009
[15] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In Proceedings of the 20th Australasian Computing Education Conference (ACE ’18). Association for Computing Machinery, New York, NY, USA, 73–82. DOI:https://doi.org/10.1145/3160489.3160500