Course Friction Explorer

Visualizing and validating indicators of student struggle

Shizuko Akamoto
shizuko@cs.ubc.ca

Noa Heyl
falkirks@cs.ubc.ca

Marie Salomon
mariesal@cs.ubc.ca

ToTo Tokaeo
tokaeo@cs.ubc.ca

Abstract

This section is left blank in the update.

1. Introduction

CPSC 310 is an undergraduate software engineering course offered at UBC Vancouver. The course involves a term-long project consisting of four checkpoints each spanning a period of 2-3 weeks. The project involves teamwork in which the student collaborates with one other (occasionally two) student(s) through GitHub using git version control. AutoTest is an evaluation tool used in this course, in which a suite of private tests are invoked against the team's solution, and feedback in the form of test failure overviews are reported back. Throughout the course of the project, students access resources including labs, office hours, and Piazza, where they seek help from the course staff. CPSC 310 is a large course, with typically more than 300 registered students, and this makes it extremely difficult for the course staff to determine when and why a student is struggling in the course. We define course "friction" loosely as any identifier of student struggles: some examples being ineffective resource allocation, non-standard use of version control, and prolonged period of inactivity. Recognizing these frictions are crucial for the course staff as it makes possible early intervention. Course friction arise from many causes, and previous works have extracted and joined data from the various tools the course employs. For example, data set taken from 2020 Winter Term 2 offering contains 25,488 AutoTest results, 5,801 Piazza contributions, and 2,366 office hour visits, joined with each student. With existing data, one way to solve this problem would be to manually inspect each table row and try to identify suspicious patterns. But tabular representations are oftentimes inefficient for searching patterns over large data sets, and thus, CPSC 310 will benefit from a more effective visualization of their course data.

1.1 Personal Expertise

We chose the domain of course friction in part because all of us are currently TAs for some undergraduate software engineering courses at UBC (CPSC 310, 319, 410). We are often faced with the difficulty of identifying when a student is struggling, which delays timely intervention and has repercussions to overall effectiveness of learning. Most undergraduate level software engineering courses employ some sort of source code management, autograding, Q&A platform, from which we can derive identifiers of friction. In particular, CPSC 310's existing datasets are especially familiar to some of us as we have been working with them previously. This makes it an ideal starting point for a more generalized visualization tool on course friction. Finally, from a software engineering perspective, understanding the pitfalls in collaborative software engineering tasks aligns with our interest as well.

2. Related Work

2.1 Computer Science Education

There is a prevalent belief in CS education research that grades in our courses are bimodal, with some population of "strugglers" and others who do well. In past work this has been explained as a "geek gene", causing some students to be predisposed to better outcomes in Computer Science courses [1]. Robins introduced the concept of "learning edge momentum" which posits that the reason CS1 grades appear the way they do is due to the tight linking of knowledge and how it builds on itself [11]. If a student falls behind the impact will compound itself quickly, which they suggest is unique among fields of study. Patitsas et al find the problem space is more complex [10]. They evaluated many course distributions at UBC CPSC and found that very few are actually bimodal. This indicates that there are not necessarily two separable populations of students but instead a single population that undergo struggles in diverse ways.

Even if we can not split students into two populations based on outcomes, there is value in understanding how and why students struggle. Various authors have used different features to create models for identifying these students. In this paper we call these models "red flags". In "Exploring the Value of Different Data Sources for Predicting Student Performance in Multiple CS Courses" the authors use grade information to predict a final course outcome [9]. They find that prerequisite grade or clicker grade strongly predicts final grade. And in "In Situ Identification of Student Self-Regulated Learning Struggles in Programming Assignments" they use measures of stagnation in grade to indicate struggle on an assignment [2]. Furthermore, Estey et al develop a model using changes in programming behaviour to identify students in need of support [4]. They find that they can identify students who require support in the first few weeks or term and target outreach to them. Neural networks have also been explored in finding students in need of assistance. One paper uses student grades and the number of submissions as features in their model [3]. Together these authors have conceptualized some feature (or "red flag") and demonstrated its relationship to student outcomes. In this paper we contribute a tool that allows quick discovery and validation of features like these in a general purpose way.

2.2 Classroom dashboard visualizations

We build a dashboard style visualisation for representing student "red flags" and understanding their relationship to outcomes. Dash-

boards are a common tool for understanding learner behaviour. For example, Kia and their collaborators created a dashboard for visualising learner attributes in a MOOC class on edX [8]. This dashboard displays attributes such as attendance, gender, and age as bar charts. Ginda and their collaborators also investigate MOOCs, creating conceptual content hierarchies and "learner path" visualisations that show the steps a learner takes through a class [6]. These tools demonstrate the utility of visualisations for educators to understand the experience of their students throughout a course. In this paper, instead of visualising learner attributes directly we visualise learners in terms of cohorts which are defined by models created by the visualisation user.

2.3 Visualization techniques and idioms

In "Quality based guidance for exploratory dimensionality reduction" the authors create a general tool and process for reducing a high dimensionality dataset into a single attribute one, allowing a user to pull out interesting elements for further inspection [5]. A user does this by selecting interesting variables and then inspecting their correlations. The problem of identifying struggling students also maps onto dimensionality reduction. However, in our work we are concerned about the correlation between each potential red flag and the true struggling students. This simplifies the problem considerably because we don't care about correlation between every pair of attributes. We also distinguish our work in that we identify correlations between membership in the set of struggling students instead of between quantitative attributes themselves. LineUp introduces a method for visualising multidimensional data in a tabular format. They facilitate the task of ranking based on a user-specified model [7]. LineUp also allows for the comparison of multiple models by displaying them side by side. This idiom is essential to identifying struggling students because users of our tool need to compare their candidate models to decide which is most useful. However, LineUp stops somewhat short of the idiom we need as they don't incorporate temporal data. We need a user to be able to understand how their model for identifying struggling students varies in accuracy and sensitivity over time. LineUp does not treat attributes as time-varying.

We also use the idiom of circle packing where students are represented by smaller circles nested in larger circles representing their groups. Circle packing was previously described in "Visualization of Large Hierarchical Data by Circle Packing" [12]. The authors of this paper use circle packing to represent tree data where multiple levels of nesting exist. In our use of the idiom, however, we only ever pack exactly one level of nesting. However, we do make use of similar techniques to layout the nested circles within the containing circle.

3. Data Abstraction

The given data abstraction consists of a series of tables representing various aspects of our dataset. We then develop a new abstraction which we expose to users of the visualization.

3.1 Input table abstraction

The input data is a series of tables. Additional information about attributes for each table is in Appendix A.

3.1.1 Table autotest_results

Throughout the project, students make incremental submissions of their code by committing and pushing to branches on their git repositories. Each team's repository consists of a single master branch and a number of development branches which are oftentimes per member or per feature. Each project checkpoint has a suite of associated tests that AutoTest runs on each push to these git branches. Students can see the result of the AutoTest run on a specific commit by explicitly requesting AutoBot [1]. AutoTest result requests are rate-limited across branches, for example, one request on any branch per six hours per student.

Autotest results are identified by their feedback_id which is a unique identifier for each result entry. They also have several attributes. They have categorical attributes representing the deliverable the result is for, the branch the result is from, the user who requested the feedback, and the user who committed the code change. They have ordered attributes for the score, the current visible score, time of the request and time feedback was given.

3.1.2 Table contributions

Piazza is the most active resource where students seek and receive assistance from not only the course staff, but also other fellow CPSC 310 students. Students can create posts which can be either a note or an answer-wanted question, categorizing them using tags. A Piazza contribution includes every action from creating a post, replying to a post, creating a new followup to an existing post, etc, all of which are recorded with timestamps in this table.

Piazza contributions are identified by their cid which is a unique identifier for each contribution entry. They also have several attributes. They have categorical attributes representing whether or not the contribution was made anonymously, the kind of contribution, whether the post was tagged as "project", the user who made the contribution, and the post where the contribution was made. There is also a single ordered attribute, the time at which the contribution was made.

3.1.3 Table queue_visits

Aside from Piazza, TA-held office hours are also one resource students use for issues that benefit from more synchronous, one-to-one interaction. Access to TA assistance in office hours are regulated by Queue@UBC, an online queue service simulating "lining-up" for help. Each student seeking assistance would enqueue and wait for a notification for their turn. A TA can view all the students currently on the queue, and would pick one to "start answering" thereby dequeuing them. Upon addressing the student's question, the TA would "finish answering", recording answer_finish. Note that contrary to a conventional queue, the TA need not follow FIFO order strictly; this is to prioritize help for the students requiring more immediate assistance.

Queue visits are identified by their qid which is a unique identifier for each queue entry. They also have several attributes. These include the user who asked the question, and the TA who answered it. Ordered attributes are the time of enqueue, dequeue, and the time at which the TA started and finished answering the question.

3.1.4 Table users

A deidentified user hash, anon_id, corresponds to each student and TA. Users have a categorical attribute representing whether or not the user is a withdrawn student. They have a single ordered attribute, the time at which their first lab of the term started.

3.2 Revised data abstraction

We want users to be able to write queries over our data to answer questions about students. We find that the complexity of joining multiple tables and filter operations is not needed for our visualization. Therefore, we refine the existing abstraction to a single table for "students" which we generate automatically. This table starts

[1] Autobot is an autograding system built at UBC, students request Autobot by commenting on a Github commit, but we run tests regardless of whether they request it.

as a filtered version of the "users" table to remove course staff and then is populated with data from the other tables.

Students are identified by their anon_id which is a deidentified hash. They have the same attributes as the users table, plus additional attributes which are generated synthetically to facilitate user queries

- num_visits - count of visits to Office Hours so far
- num_contributions - count of Piazza contributions so far
- $score$ - current project score
- *...and additional attributes we devise*

4. Task Abstraction

We want to build a visualization dashboard that will help instructors and TAs detect struggling students early on in a course (CPSC 310). To do so, they can use certain indicators correlated with some outcome, for example low final grades, as red flags to identify the students. These indicators may be some patterns dependent on the following:

- Office hour visits
- Piazza contributions
- Auto-grading results

If a student happens to fulfill a red flag, then some intervention may be helpful in keeping the student on track. Although these red flags can be any arbitrary condition, we can also do some prior analysis using previous years data to discover meaningful red flags. To simulate making a prediction, there should also be a way to restrict available data to a certain time-frame. We will also consider calculating some statistics that may describe how well the indicator predicts the outcome.

In order to accomplish this goal we introduce the following high-level tasks

- **T1**: Discover indicators that identify students who are struggling.
- **T2**: Evaluate and compare candidate indicators based on their sensitivity, accuracy, stability, and speed (at identifying struggling students).

5. Solution

Our solution allows instructors and TAs to perform rapid verification on their hypothesized indicators of student struggles. It should provide an intuitive means to visually confirm a certain pattern in student data as a potential red flag leading to some unfavourable outcome, thereby enabling course staff to step in at an earlier stage. The instructors and TAs can use pre-proposed indicators of student friction from previous studies to verify against their own course dataset, but the visualization should also guide them to explore their dataset and discover more red flags.

We propose a solution consisting of two main idioms

5.1 Indicators Board (Circular Packing)

An example of this view is shown in Figure 1. Outcome of interest is represented as a central circle, whee each indicator being verified is positioned as circles around the outcome. Students identified by each indicator (ie. belonging to the cluster described by the indicator) are represented as sub-circles and packed inside. The size of an indicator circle encodes the number of students contained in each cluster. In addition, each indicator circle is colour-coded with linearly varying saturation in the red hue, encoding its similarity to the outcome circle. Similarity is defined by member containment: the indicator cluster contains a student also present in the Outcome cluster. This value is be binned into several saturation ranges to avoid the problem of indistinguishable colour encoding. Interaction in the collapsible side panel allows configuring the outcome and identifiers to be verified. Other interactions include zooming into a single cluster, dragging (useful when many indicators are simultaneously visualized), and potentially hovering to display more detailed information about the cluster. The user can also interact with the time slider to step through the evolution of each cluster in a specified time interval.

5.2 Table

An example of this view is shown in Figure 2. The user can navigate to a per indicator table visualization by selecting the indicator circle in the circle packing of visualization 1. Students in the selected indicator cluster are on the vertical axis and the different indicators on the horizontal axis. (NOTE: these indicators need not be part of what was selected in visualization 1). In each cell is the derived value of student data based on the indicator. Rows with students also contained in the outcome cluster should be colour-coded with red colour. Interactions include sorting on the indicator columns and filtering on the rows. Further details of this visualization is currently TBD, but the goal is providing opportunities for explorations to identify identifiers other than the set from visualization 1.

5.3 Handling complexity

Our solution uses common techniques in order to handle complexity. These include navigation, juxtaposition, and dimensionality reduction.

5.3.1 Views and Manipulation

We implement a time slider than will select a time range for the data. As the slider shifts, marks representing data time-stamped outside of this range will disappear from the vis, and likewise as the timerange expands, marks will appear. These animations may be considered a Change manipulation, and selecting a time range can be a Navigation to reduce the number of visible items or attributes.

5.3.2 Juxtapose

Using different circles representing different indicators, we can also juxtapose multiple views, which will allow users to compare the predictive powers of these indicators. The set of marks contained in each circle may be subsets or disjoint sets from each other, but each view should share the same encoding.

5.3.3 Dimensionality reduction

We consider our tasks **T1** and **T2** as a simple version of a dimensionaility reduction problem. Our dataset contains a myriad of dimensions about each student which altogether may have some degree of predictive power over their final outcome. We allow a vis user to select some of these dimensions to generate a new synthetic dimension and then explore how well this dimension partitions students. Our reductions are much simpler than literature as our synthetic dimensions are always binary, either a student is in them or not, they don't have a magnitude.

5.4 Scenario

Imagine the user of this course friction explorer as an instructor for CPSC 310. It is currently mid-term and the students have just passed their deadline for submitting code for checkpoint 1 of the project, and AutoTest executed test suites against each of their codebase to give an interim checkpoint grade. It is a good time to evaluate how the students are progressing in the course, and if any of them are showing signs of struggle. The user would want

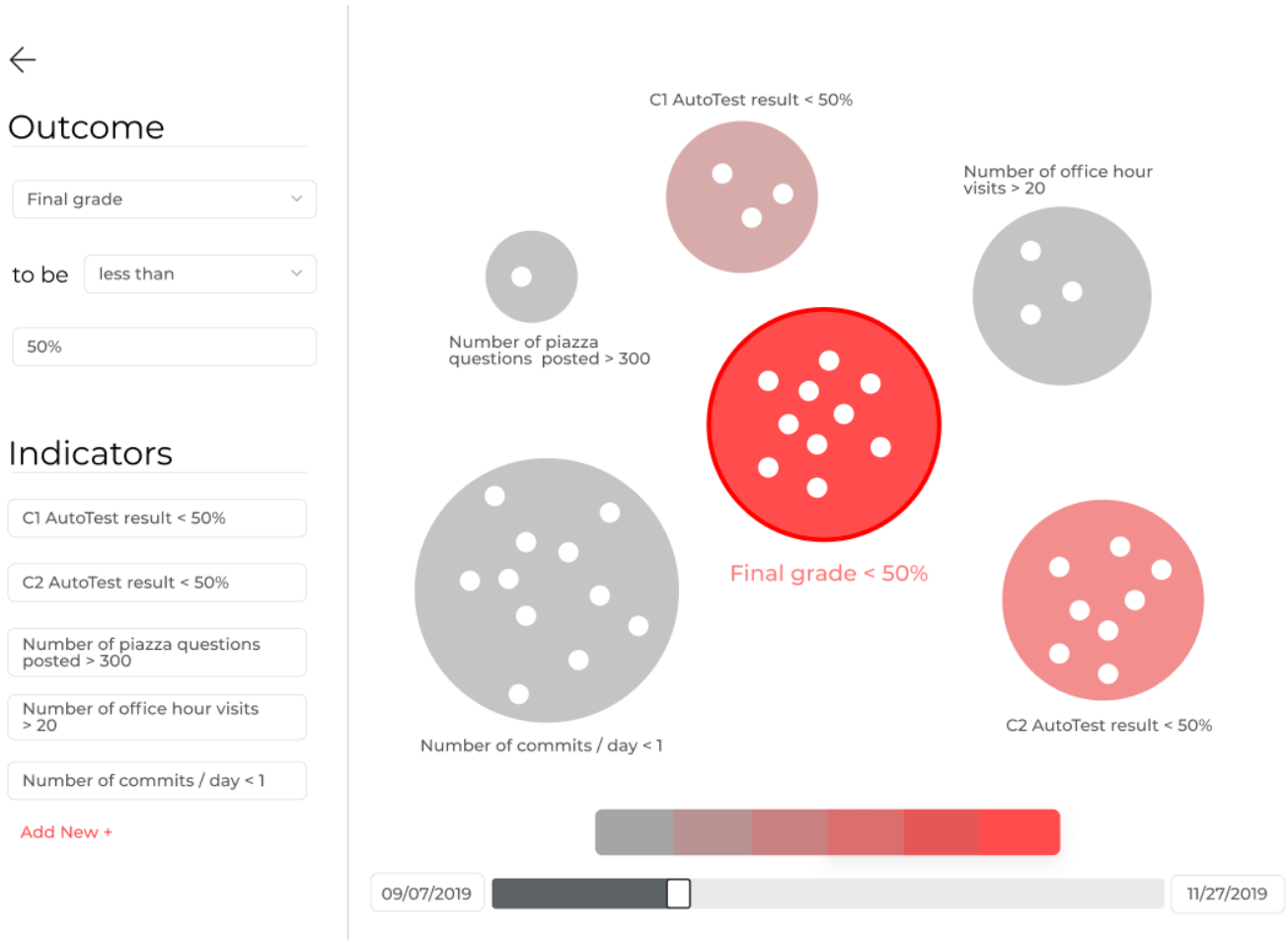


Figure 1: Circular packing view of 5 possible indicators of a failing final grade.

to identify struggling students and reach out to them earlier so that they can make improvements toward the end of the term. As in previous studies shown there are several factors that often are the causes of friction:

- Starting course assignment late
- Performing suboptimally on one of the earlier assignment
- Seeking assistance repeatedly in a short span of time

Now, if these indicators were verified to also be applicable to CPSC 310, then a bulk of the user's work is complete. The rest of the work involves identifying students with these particular characteristics, which is comparably straightforward. The user would turn to Course Friction Explorer for performing the work of friction verification against datasets from the most recent course offering: 2019 Winter Term. The user is most likely interested in knowing whether these proposed friction indicators can actually identify students that have failed the course in that term. So, in the circular packing visualization, the user can set the "Outcome" circle to be "Students who received final grades less than 50%". Then, the user can add each one of the aforementioned characteristics as an indicator circle. These circles are, "Students who started checkpoint 1 after Oct 1. 2019", "Students who received AutoTest results of less than 50% on checkpoint 1", and "Students who create more than 5 piazza posts daily on average". After all the interested indicators are added, the visualization displays a central red circle corresponding to the Outcome cluster, and three surrounding Indicator circles of different sizes.

In particular, the circle corresponding to the indicator "Students who received AutoTest results of less than 50% on checkpoint 1" is coloured with saturated red, very much similar to the colour of the Outcome circle. The user is able to click into the particular indicator circle to inspect the member students. The user is able to perform the same verification on datasets from multiple different past offerings, all leading to this convergent result.

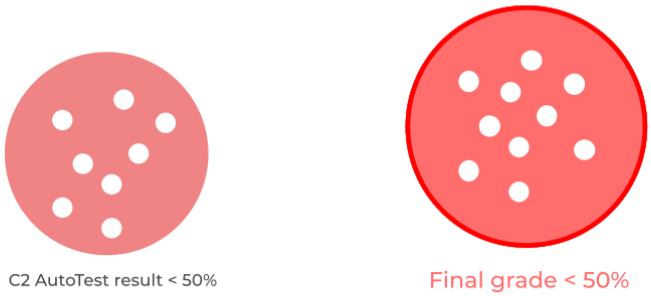
Learning from this verification, the user can make a preemptive move of reaching out to the students from the current term whose checkpoint 1 AutoTest result reported a failing grade.

5.5 Implementation Approach

We implement Course Friction Explorer as a web application. We use a backend written in Python [2] and using FastAPI [3]. The backend assumes responsibility for loading and processing the input dataset. The dataset is then made available for querying using a DSL we designed over a REST endpoint. This allows us to perform the majority of the analysis using Python libraries which we are more familiar with.

[2] https://www.python.org/

[3] https://fastapi.tiangolo.com/



Add New +

CWL	Number of commits / day	Number of piazza questions posted	C1 AutoTest result	C2 AutoTest result
cwl1	1	5	65%	50%
cwl2	3	0	80%	93%
cwl3	2	20	99%	95%
cwl4	5	6	72%	81%
cwl5	10	4	100%	100%
cwl6	15	0	85%	84%
cwl7	2	0	44%	53%

Figure 2: Tabular view of the results of a specific indicator.

The backend endpoints are used by our React [4] and d3.js [5] frontend.

5.5.1 Loading dataset for querying

We load the datasets which are stored in a SQLite database into a memory representation which exposes a time varying interface. This means that the frontend can provide arbitrary timestamps and the backend can sample attributes at that time. The process of loading datasets also maps from the input data abstraction to our new ”users” based abstraction.

5.5.2 DSL: Writing queries

As part of our backend, we have designed a domain-specific language (DSL) (see Appendix B) to help process different queries in our vis tool. Interacting with the front-end interface would translate to constructing a query in our DSL in the backend, and our vis tool will evaluate this query, producing the desired output. This DSL should be able to express all possible interactions with our tool. Though the DSL will still morph as we decide on the possible user interactions with the vis, we have a parser that will construct an abstract syntax tree from any query written in this DSL. We might also allow the vis user to directly write their queries in our DSL in addition to using the graphical interface.

5.5.3 Frontend

Our React frontend makes use of a number of libraries to help expedite development. In particular, we use React components from Materials UI, applying customized CSS via styled components. Redux is the backbone for our frontend state management, giving all our individual components access to a consistent global state to which they can adapt their views on. The overall structure of the frontend is depicted in Figure 3.

The service layer is responsible for communicating with our backend using REST API queries: the `QueryService` and `StudentService` makes request to `POST /query` and `GET /students/id` endpoints surfaced by the backend server respectively.

The state management layer lie in between our service layer and the frontend React components. We choose to offload state retrieval and update logic from the individual components into this state layer so to ensure states are accessed and modified in consist way by all components and to minimize code duplication. Redux is used heavily for this purpose. We have defined a global redux store, consisting of multiple slices (subsets of the global state accessible as a single unit from frontend components). For example, the `indicatorsSlice` maintains the current working set of indicators configured by the user, and surfaces the `addIndicator`,

[4] https://reactjs.org/

[5] https://d3js.org/

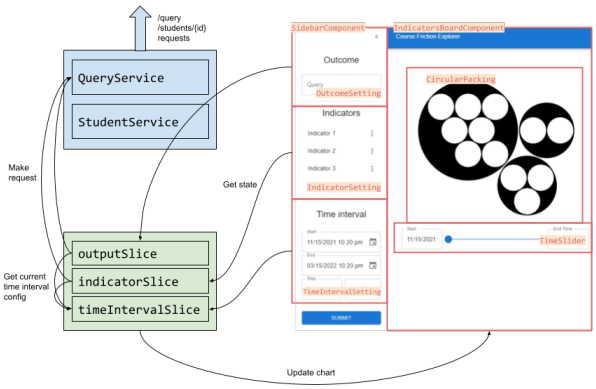


Figure 3: Interaction between frontend services, states, and components

`editIndicator`, `removeIndicator`, and `queryAllIndicators` as common access methods callable from all frontend components as needed.

Finally, the React components serve as our presentation layer. Our visualization dashboard is broken down into two major page components: `IndicatorsBoard` and `IndicatorTable` components. Each composes smaller sub-components like the `Sidebar`, `TimeSlider`, `IndicatorEditorDialog`. Despite the many components accessing and modifying shared state, our components remain moderately decoupled from each other, thanks to the state management layer moderating state accesses. Integrating React with d3 came with some complexity, as both attempts to assume full control of the DOMs in view. In order to ensure they work in combination, we need to define a clear separation in ways they can perform DOM manipulation. We wish to utilize d3's strong set of data manipulation and visualization methods, including animation and transitions. On the other hand, React's ease in receiving user input events (eg. zoom, pan, click) and compatibility with our state layer are crucial. Therefore, our approach involves using React component as a wrapper around each chart DOM manipulated entirely by d3. For example, our circular packing chart in the Indicators Board has a wrapper `CircularPacking` component which renders a single DOM node `<vis-circular-packing>`. The component feeds all data needed by d3 to manipulate `<vis-circular-packing>` to properly compute and visualize the chart. Upon updates from the state layer and input events from user, it is `CircularPacking` component's responsibility to trigger appropriate d3 methods to update the chart with new data. We have found this architecture gives us the benefits from both libraries with minimal conflicts.

6. Milestones and Schedule

Table 1 displays the proposed development schedule of the Course Friction Explorer from the initial planning to writing and submitting the final paper. We are estimating a total of 81 person-hours on task, with the main hours spent at implementing the Course Friction Explorer followed by writing the paper. However, estimating and planning person hours on task tends to be very difficult and error-prone. Therefore, we will be iteratively revising the allocation of hours, to confirm their accuracy or adapt where needed. After revising Table 1, the deadline for the backend of the course friction explorer got pushed back due to the fact that the development of the backend and frontend started simultaneously. Moreover, the current status of each task and actual hours spend on completed task is displayed in Table 1. Table 2 and Table 3 provide a detailed breakdown of the tasks including the responsible team members for each task, estimated workload, actual hours spend on each task and the status of each task. The estimated workload is equally divided among the four team members.

7. Discussion

This section is left blank in the update.

	Task	Due date	(Total/Per person)	Actual hours spent	Description	Status
1	Pitch	Sep. 29	8/2	8	Create content and rehearse pitch	Complete
2	Proposal	Oct. 21	28/9	28	Discuss the project, create illustrations and write the project proposal	Complete
3	Learning and understanding the tools	Oct. 28	40/10	36	Read the documentations and examples. Learn how to use the tools and how they can interact.	Complete
	- d3.js	Oct. 24	16/4		16	Complete
	- Fast API	Oct. 26	12/3		8	Complete
	- Python	Oct. 28	12/3		12	Complete
4	Project Update I	Nov. 16	12/3	16	Prepare and provide updated paper for Peer Reviews	Complete
5	Project Update II	Nov. 24	16/4		Prepare for the Post-Update Meeting and demonstrate the prototype	To do
6	Implementation	Dec. 6	160/40		Implement and complete the Course Friction Explorer	In progress
	- Backend: Setup, Data, Configs, Querying, DSL	Nov. 26	60/15		Setup the environment, clean and work with the data in the backend, do the configs and create queries.	In progress
	- Frontend: Circular packing, Table	Nov. 26	60/15		Implement the frontend by i.a. creating circular packing and table models.	In progress
	- Analysis	Dec, 6	40/10		Generating additional properties people might use	To do
7	Draft of the final paper	Dec. 8	20/5		Write draft of the final paper	To do
8	Presentation	Dec. 15	16/4		Prepare slides for the final presentation and talking points	To do
9	Final paper	Dec. 17	24/6		Finish the paper and include final changes and conclusion	To do

Table 1: Overview of project milestones and person hours allocated to each

Task	Assignees	(Total Hr. / Per person)	Actual hours spent	Status
Environment setup	Marie Salomon, ToTo Tokaeo	6/3	5	Complete
Clean and load the data	Noa Heyl	6		In progress
Configs	Shizuko Akamoto	3	3	Complete
DSL Features	Noa Heyl, ToTo Tokaeo	12/6	12	Complete
EBNF and parsing	ToTo Tokaeo	18		In progress
Queries	Noa Heyl	15		In progress

Table 2: Breakdown of project tasks for the backend of our project

Task	Assignees	(Total Hr. / Per person)	Actual hours spent	Status
Environment setup	Shizuko Akamoto, Marie Salomon	4/2	4	Complete
Redux integration	Shizuko Akamoto	3	3	Completed
Circular Packing	Marie Salomon	15		In progress
Time slider	Shizuko Akamoto	11		In progress
Table models	Shizuko Akamoto, Marie Salomon	20/10		In progress
Additional visualizations	Shizuko Akamoto, Marie Salomon	6/3		In progress

Table 3: Breakdown of project tasks for the frontend of our project

References

[1] Alireza Ahadi and Raymond Lister. 2013. Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant?. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 123–128. https://doi.org/10.1145/2493394.2493416

[2] Kai Arakawa, Qiang Hao, Tyler Greer, Lu Ding, Christopher D. Hundhausen, and Abigayle Peterson. 2021. In Situ Identification of Student Self-Regulated Learning Struggles in Programming Assignments. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 467–473. https://doi.org/10.1145/3408877.3432357

[3] Karo Castro-Wunsch, Alireza Ahadi, and Andrew Petersen. 2017. Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 111–116. https://doi.org/10.1145/3017680.3017792

[4] Anthony Estey, Hieke Keuning, and Yvonne Coady. 2017. Automatically Classifying Students in Need of Support by Detecting Changes in Programming Behaviour. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 189–194. https://doi.org/10.1145/3017680.3017790

[5] Sara Johansson Fernstad, Jane Shaw, and Jimmy Johansson. 2013. Quality-based guidance for exploratory dimensionality reduction. *Information Visualization* 12, 1 (2013), 44–64. https://doi.org/10.1177/1473871612460526 arXiv:https://doi.org/10.1177/1473871612460526

[6] Michael Ginda, Michael C. Richey, Mark Cousino, and Katy Börner. 2019. Visualizing learner engagement, performance, and trajectories to evaluate and optimize online course design. *PLOS ONE* 14, 5 (05 2019), 1–19. https://doi.org/10.1371/journal.pone.0215964

[7] Samuel Gratzl, Alexander Lex, Nils Gehlenborg, Hanspeter Pfister, and Marc Streit. 2013. LineUp: Visual Analysis of Multi-Attribute Rankings. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2277–2286. https://doi.org/10.1109/TVCG.2013.173

[8] Fatemeh Salehian Kia and Simon Fraser. 2016. Learning Dashboard: Bringing Student Background and Performance Online.

[9] Soohyun Nam Liao, Daniel Zingaro, Christine Alvarado, William G. Griswold, and Leo Porter. 2019. Exploring the Value of Different Data Sources for Predicting Student Performance in Multiple CS Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 112–118. https://doi.org/10.1145/3287324.3287407

[10] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. 2016. Evidence That Computer Science Grades Are Not Bimodal. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 113–121. https://doi.org/10.1145/2960310.2960312

[11] Anthony Robins. 2010. Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education* 20, 1 (2010), 37–71. https://doi.org/10.1080/08993401003612167 arXiv:https://doi.org/10.1080/08993401003612167

[12] Weixin Wang, Hui Wang, Guozhong Dai, and Hongan Wang. 2006. Visualization of Large Hierarchical Data by Circle Packing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*. Association for Computing Machinery, New York, NY, USA, 517–520. https://doi.org/10.1145/1124772.1124851

A.
Input dataset attributes

A.1 Table autotest_results

A.1.1 Categorical attributes

- deliv - Checkpoint number
  - 4 categories (example: `"c0"`)
- ref - Git branch AutoTest was called on
  - 1665 categories (example: `"ref/tags/c2-rc5"`)
- is_master - True if ref is the master branch
  - 2 categories (example: `1`)
- feedback_requester - Deidentified user hash of the requester
  - 409 categories (example: `"j+TyZUe34/c1mZOH9tppky9C..."`[6])
- committer - Deidentified user hash of the committer
  - 409 categories (example: `"F+CFt8v9oVAaHZppCNKYTSN..."`)

A.1.2 Ordered attributes

- score - Test score of this AutoTest run
  - Ranges from 0.00 to 100.00 (example: `100.00`)
- visible_score - Test score currently available to the student
  - Ranges from 0.00 to 100.00 (example: `90.00`)
- request_time - Timestamp of when a student requests the result
  - Ranges from 1610400675825 to 1622674908773 (example: `1610401163054`)
- feedback_time - Timestamp of when result is returned
  - Ranges from 1610400675825 to 1619628202579 (example: `1610521081669`)

A.2 Table contributions

A.2.1 Categorical attributes

- is_anonymous - True if post is created by anonymous contributor
  - 2 categories (example: `0`)
- kind - Contribution kind
  - 12 categories (example: `"followup"`)
- is_project - True if contribution is tagged as project
  - 2 categories (example: `1`)
- anon_id - De-identified id of the contributor
  - 409 categories (example: `"07e7yyUGH4zoF+i5UF3PH9d..."`)
- post_id - Unique identifier of the post the contribution was on
  - 1436 categories (example: `"Ks43Y68znhtzXws8zNnDG..."`)

A.2.2 Ordered attributes

- created_at - Timestamp of contribution
  - Ranges from 1610149993000 to 1620093785000 (example: `1619632354000`)

A.3 Table queue_visits

A.3.1 Categorical attributes

- anon_id - Deidentified identifier of student asking question
  - 409 categories (example: `"DjN2/LSrZHkxfxAk/ka8gIigB6..."`)
- answerer_id - Deidentified identifier of TA answering question
  - 27 categories (example: `"Nxw7gaFw+d2v0moktJ1dGKzd4Ix2..."`)

A.3.2 Ordered attributes

- enqueue - Timestamp of enqueue
  - Ranges from 1600386235000 to 1618015298000 (example: `1618005226000`)
- dequeue - Timestamp of dequeue
  - Ranges from 1600386253000 to 1619307862000 (example: `1619307843000`)
- answer_start - Timestamp of when the TA starts answering
  - Ranges from 1600386239000 to 1618013723000 (example: `1618008486000`)
- answer_finish - Timestamp of when the TA finishing answering
  - Ranges from 1600386239000 to 1618016558000 (example: `1618004632000`)

A.4 Table users

A.4.1 Categorical attributes

- withdrawn - True if the user withdrew from the course
  - 2 categories (example: `1`)

A.4.2 Ordered attributes

- first_lab_time - Timestamp of the user's first lab. Null if user is a TA
  - Ranges from 1610384400000 to 1610751600000 (example: `1610751600000`)

[6] Identifier hashes have been truncated to save line space in this paper.

B.
DSL Grammar

```
query : filter;
filter : logic | binary | time_op;
logic : '('filter ('AND'|'OR') filter')' | '(''NOT' filter')';
time_op : ('BEFORE'|'AFTER') time | 'BETWEEN' time time;
binary : '('comparable ('>'|'<'|'<='|'>='|'!='|'==') comparable')';
comparable : number | time | string;
number : Number | arithmetic | granularity_result | student_attribute;
granularity_result : ('daily'|'weekly'|'monthly'|'final'|'sofar')
                     '('student_attribute')';
arithmetic : '('number ('+'|'-'|'*'|'/') number')';
string : student_attribute | String;
time : time_lit | student_attribute;
student_attribute : 'student.'attribute;
attribute : String;
time_lit : 'time' '(' String ')';

Number : [0-9]+;
String : [a-zA-Z0-9_\-]+;
```