Visualizing Novice Approaches to Programming

Using BlueJ Blackbox Data

Chili Johnson
University of Puget Sound

Robert Shelton
University of Puget Sound

Xeno Fish
University of Puget Sound

ABSTRACT

This paper provides a sample of a LaTeX document which conforms, somewhat loosely, to the formatting guidelines for ACM SIG Proceedings.

1 INTRODUCTION

We sought to create a visualization tool or for mapping how novice computer science students approached and solved a given assignment. Such a mapping seems intuitive for early problems which have a relatively small number of truly distinct solutions. One can only print "Hello World" so many ways. Beyond that once a set of students is trying to make a specific problem with a specific output as one might remember from introduction courses we could categorize the different end solutions turned in by the class. They might differ in the control flow and objectification, thus solutions that seem to take different approaches might still accomplish the same end goal. A professor grading thirty or forty different solutions could recognize similarities and patterns across them. What if one solution was preferable? how do students find that solution? How is this changed making the students do pair programing? Taking this categorization a step further we could categorize their relative progresses to map out what brought them to one path or another. Such a mapping would require not just the end states but also the start states and a series of intermediate states across many students completing the same assignment. The BlueJ Blackbox database gave us exactly that.

BlueJ, an integrated development environment for Java aimed at elucidating the nature of object-oriented programing to students starting out, has an optional opting-in that allows them to record the student's code after anonymization for academic use. Developed by the university of Kent, also distributed was an intro textbook to accompany the software. [3] Their software records to high granularity the changes in a students code. This gave us the data set we needed as well as unified the assignments needed to be held constant.

With this data in hand we set out to build a tool to visualize how the students moved through the solution space. Such a mapping could be used to see the thought processes of the students, common pitfalls, and just how varied the solutions could be.

2 BACKGROUND

Everything we did started with the blackbox database. This collection of student programming assignments, maintained by the University of Kent, records every time a student moves in between lines with their cursor as an edit event. This gave us a fairly fine granularity of edits to look at when we were comparing different source codes. The database also tracks different sessions of user activity, and contains all the edit, file open, file rename, and any other events they might perform within those sessions. Edit events are recorded as either complete sources or diffs. This forced us to consider which different kind of source we were looking at, find a complete source, and then apply the diffs up to the first successful compile to plug into the java parser.

The previous work we looked at for this program either didn't generalize the collection of information from the blackbox database or didn't use the database itself. "Modeling How Students Learn to Program" by Piech, Sahami, Koller, Cooper, and Blikstein was a similar study that also used abstract syntax trees to parse code, but their system wasn't applicable over a large dataset like the one we wanted to work with. They had a set of "computer science students with teaching experience [5]" evaluate 90 pairs of source codes taken from a modified version of eclipse to determine distance. First of all we wanted to automate the distance function, so manually coming up with distance metrics wasn't feasible. They also used a hidden markov model to evaluate the states, but that also wasn't feasible for our project because our automatically generated data turned out to be difficult to label.

The other paper we looked at in depth, "37 Million Compilations: Investigating Novice Programming mistakes in Large-Scale Student Data" used the blackbox database specifically. [2] They were primarily concerned with errors and mistakes made by students. They used compiler errors to catch 4 different kinds of student mistakes, and the rest of the logical errors were caught with a custom parser. This study focused on looking just at mistakes rather than a time series of student code, and we were more interested in how code changed over time.

While deciphering and incorporating common error states into our visual map was something we initially wanted to delve into, we found it just was not feasible with our approach. We ended up using a Java abstract syntax tree parser in our abstraction of student code. Unfortunately the parser we used understandably could not parse broken code.

The work done by Hovemeyer, Hellas, Petersen and Spacco deals with where students tend to make the most mistakes.[4] They also performed analysis on student code through the use of abstract syntax trees, but they limited their focus specifically to control flow. They found that fiLoop problems feature prominently in research on student understanding of programming concepts, since they are a major, challenging component of CSI courses.fi However, we decided that rather than looking exclusively at control flow we could look at how students factor their code into different sections

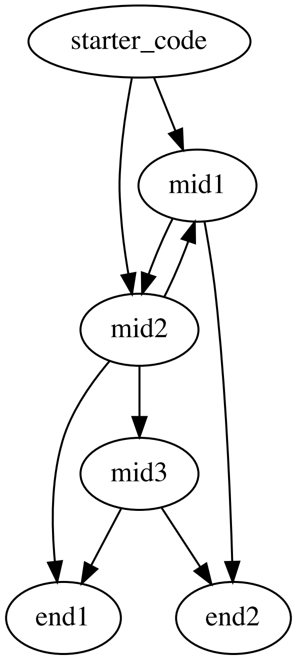

Figure 1: A sample of the desired output of our visualization tool. Vertices represent clusters of similar project states and edges represent transitions between different project states.

to get a good idea of how they were thinking through a problem. We looked at method and class declarations as well as method calls to differentiate between student code.

3 METHOD & IMPLEMENTATION

The output of our visualization tool is a finite state machine diagram which is the condensed representation of many students' paths through a particular programming assignment (see section 3.1). Figure 1 is a toy example of our desired output. This contrived example shows students whose projects begin in the same `starter_code` state, transition through various intermediate states, and ultimately end up in two different end states.

To produce this diagram, we needed to define not only what a student's project's state is at any given time, but also define a coordinate space in which these states exist in order to allow states to be clustered by "similarity" (see section 3.2 Project State).

3.1 Model & Terminology

The highest-level data model in our tool is the *assignment*. An assignment is a single programming problem (*e.g.* assigned homework problem or textbook exercise) which students attempt to complete. An assignment is a collection of individual projects each which represent a single student's attempt to complete the assignment. Projects are assumed to be within the same assignment if they have identical names, *e.g.* all projects with the name `CSCI161-Hwk1`, or `book-exercise` are considered to be in the same assignment.

Each project consists of a sequence of sessions. Sessions are collections of events which occur during the time between when a project is opened, and when it is closed. Events are collected for many user actions including compilations, changes in source code, debugging, file renaming, invocations, *etc.* Figure 2 Illustrates the hierarchical structure of our data model.

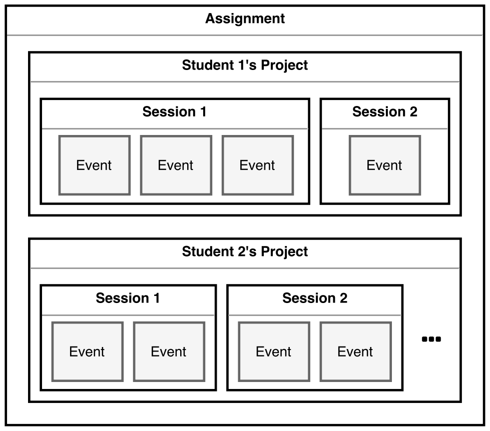

Figure 2: Our data model derived from the Blackbox model. A single assignment is the collection of projects with the same name.

3.2 Project State

There are two components to our concept of project state. The first is our state representation in which we use abstract syntax trees (AST) to represent the structure of a project's source code at any given time. The second is the coordinate transformation which allows us to place these ASTs within a coordinate space.

3.2.1 Abstract Syntax Trees. Java code can be represented as a tree where any vertex's children are either statements within the vertex's block, or subcomponents of a statement. Figure 3 is a simplified example of such a tree. We represent a project's state as the abstract syntax tree generated directly from the Java source code of the project.

3.2.2 Coordinate Transformation. In order to afford clustering "similar" project states together, we require a concept of distance or space. Determining the distance, or edit distance, between two trees is a computationally expensive problem. Using tree edit distance as a distance metric is further complicated by the fact that vertices—*e.g.* method declarations—in certain levels of an abstract syntax tree can be reordered without affect on the overall program, whereas some vertices—*e.g.* control structures—cannot be reordered without semantically changing the program.

We ultimately decided to implement a relatively naïve approach to the coordinate transformation. To convert an abstract syntax tree into its coordinate, we first define the space in which we are interested; in our case we are interested in class declarations, method declarations, method calls, variable declarations, and assignments statements, but our method is easily modifiable to consider more or different features of a given program. We take these interesting features and traverse the abstract syntax tree, counting the number of occurrences of each feature. We then take these counts as

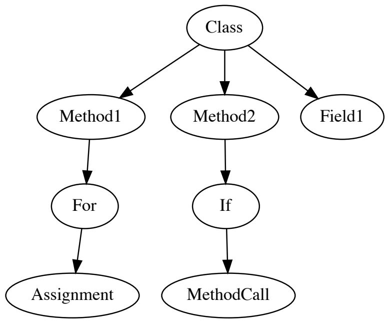

Figure 3: A simplified example of a Java abstract syntax tree.

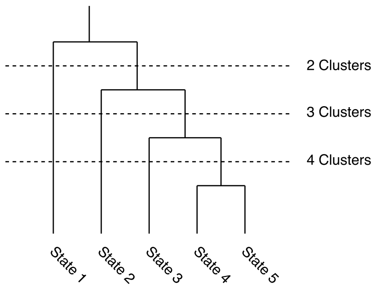

Figure 4: A hierarchical clustering tree. Project states are clustered by similarity, and the resulting hierarchy can be retroactively cut to visualize states at different levels of granularity.

an n-tuple and use that tuple as our coordinate in space. The distance between two states/trees is then only the Euclidean distance between their computed coordinates.

For example, if a given project state includes 8 classes, 4 method declarations, 6 method calls, 8 variable declarations, and 5 assignments, our naïve transformation would produce the coordinate $(8, 4, 6, 8, 5)$.

3.3 Clustering

After defining the coordinate space into which our project states can be transformed, as well as defining the transformation itself, we are able to employ a clustering algorithm to determine which projects are most similar in terms of program structure. We chose to employ an agglomerative hierarchical clustering algorithm. The advantage of using the particular algorithm is that it produces a hierarchical tree of similarity which we can retroactively cut at different levels in order to visualize student progress at coarser or finer granularity. Figure 4 illustrates this property.

3.4 Analysis Pipeline

We implemented our visualization tool as a linear pipeline of processing stages as illustrated in Figure 5. Our pipeline was written in the Ruby programming language. Each stage saves output incrementally to disk to allow the analysis to be tolerant to interruptions. Each stage—except the first stage—reads the previous stage's output from disk as it performs its computations. There are 3 stages to our analysis pipeline: **query**, **source reconstruction**, and **AST/clustering**.

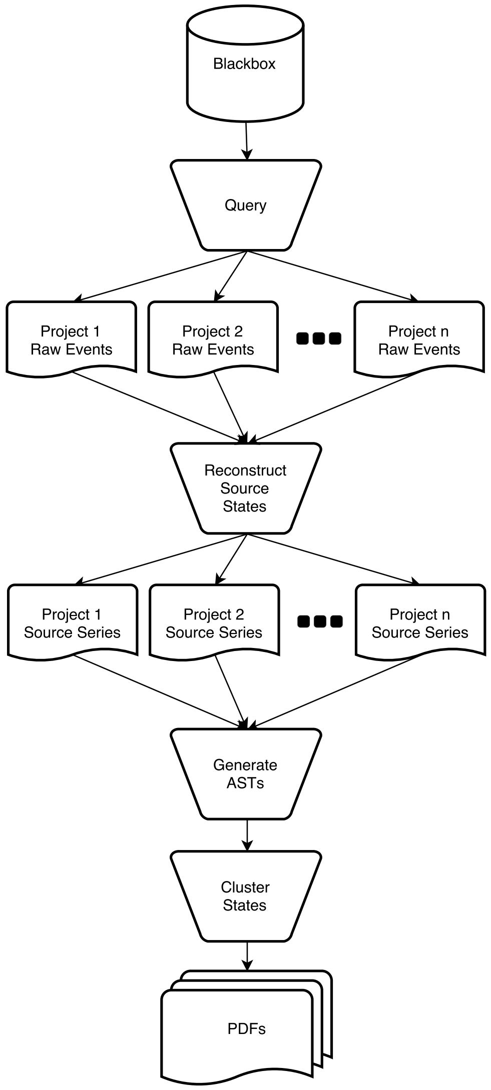

Figure 5: The flow of data through our analysis pipeline, including intermediate states saved to disk.

3.4.1 Query Stage. The query stage is responsible for interfacing with the Blackbox database and is the only stage which doesn't read any intermediate data from disk. This stage uses ActiveRecord, a Ruby Object-Relational Mapping, with a MySQL adapter to make complex queries against the Blackbox database with relatively simple and readable syntax.

The intent of the query stage is to retrieve all events relevant to our analysis and save them to disk, organized by project. A project is the representation of a real BlueJ project created by a user. Every project contains a collection of sessions, and each session contains a sequence of events representing various actions by the user during that session. For each project and each session within that project, the query stage filters all contained events and only

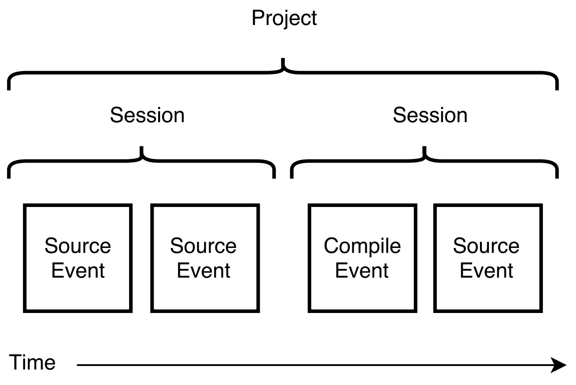

Figure 6: The flow of data through our analysis pipeline, including intermediate states saved to disk.

keeps events representing changes in source code as well as events representing attempted compilations. The raw attributes of these events are saved to disk in individual YAML files by project, with the structure shown in Figure 6. Source events are either complete snapshots of source code, or code differences in the unified diff format. Compile events only contain a boolean indicating whether or not the project code was successfully compiled in its current state.

The query stage executes a large number of database operations on a relatively slow database server, so this stage can take quite a while to complete. Processing a single project can involve hundreds of foreign key lookups, each empirically requiring 100–200ms per query. To shorten the execution time, the query stage is multi-threaded, making use of many simultaneous database connections to perform queries in parallel. Even with the parallelism, the query stage can take quite a bit of time to execute. For example, the query stage took 50 hours to process 11,000 user projects for a relatively-simple assignment using 8 threads / simultaneous connections.

3.4.2 Source Reconstruction Stage. The source reconstruction stage reads in the event series produced by the query stage and produces a time series of snapshots of the state of the source at every successful compile. Project histories are reconstructed by traversing the event series for each file in the project and patching the unified diff events into a complete snapshot of the project state at any given point in the traversal. The project state snapshots are saved to an output file whenever the traversal encounters a successful compile event. The final output of this stage is a collection of files each containing the series of project snapshots taken at each successful compile event.

3.4.3 AST Generation and Clustering. The final stage of our pipeline can be divided logically into two sub-stages with no intermediate output saved to disk between the stages. The first stage is responsible for converting the saved series of project states from the previous stage into a series of abstract syntax trees. The second stage is responsible for clustering the converted project states as well as tracing each student's path through the different clusters of states.

The first stage calls upon a small Java utility to convert each of the input source states into a series of abstract syntax trees. As these trees are read back into our Ruby environment, their coordinates are calculated using the naïve method described in section 3.2.2. Once all syntax trees have been converted to coordinates, the trees are passed to the second stage.

The second stage uses the agglomerative clustering algorithm to generate a hierarchical similarity tree of *all* project states. This tree is then cut at various levels (see Figure 4) to produce various sizes of sets of project state clusters. For each set of project state clusters, we use information about each student's temporal path through each of their project states to generate a digraph (see Figure 1) with edges weighted by the number of times a student made any particular state transition.

Finally these digraphs are converted to the Graphviz DOT graph description language, and then converted into PDF files using the Graphviz utility.

4 RESULTS & ANALYSIS

The biggest problem we discovered towards the end of this project was the runtime of the agglomerative clustering algorithm we had selected to cluster the states of student code. We already knew the algorithm was $O(n^3)$, but when we actually started running the code it became apparent that we had underestimated what runtime we would actually be cubing in that equation. We had pulled a little over 11,000 projects for the "Auction" assignment detailed in the BlueJ textbook, but running our code on just 100 of those took just over two hours. This lead to an equation for runtime looking like this:

$$\begin{aligned} f(100) &= 2 \text{ hours}, \\ 11000/100 &= 110, \\ 110^3 &= 1,331,000, \\ 2 \text{ hours} * 1,331,000 &= 2,662,000 \text{ hours}, \\ &= 303 \text{ years}. \end{aligned}$$

Needless to say we didn't exactly have 303 years to run this program. In fact, we had about a week, so we came to the realization that we needed to pick a smaller dataset to run on. We chose 300 projects, which was about as big as we could get without leaving instructions for our grandchildren to handle the results.

Our algorithm let us choose how many clusters we wanted to view for a given project. This produced a series of PDF files in between the bounds we selected for the number of clusters. For the 300 project run, we looked at all series between 2 and 40 clusters. Some of the lower numbers of clusters weren't exactly useful, because the agglomerative clustering algorithm shows only outliers from the most common path. At the low levels the extreme outliers are basically only the students that saved the wrong project under this name. This means that the more clusters there are, the more compiles we see come out of the one big cluster.

First off we can see the far outlier clusters in the 21 cluster graph in "M" and in the 36 cluster graph in "N" and "W." The large cluster "S" that appears in the 21 cluster graph has over 1300 compiles in it, where as that same large base cluster "BF" in the 35 cluster graph

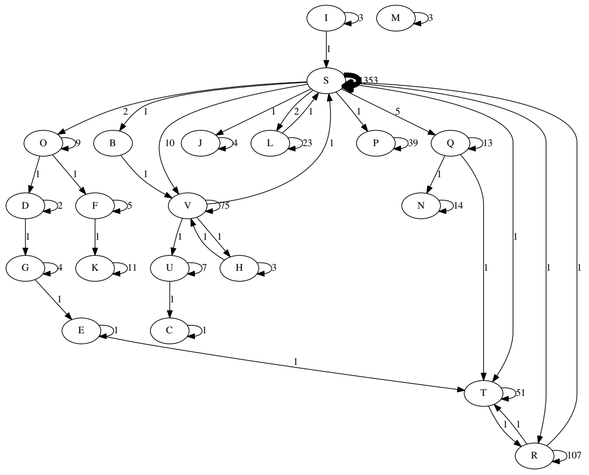

Figure 7: This graph has one large cluster, (S), containing the majority of the compiles.

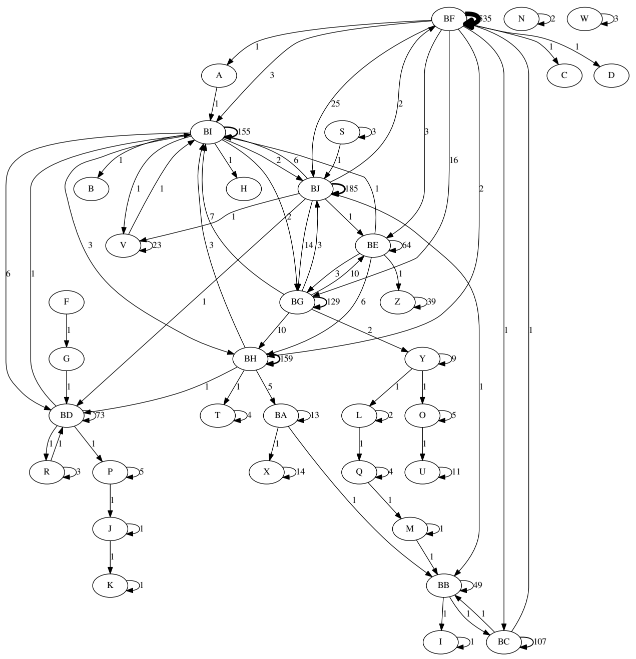

Figure 8: Here the one large cluster in the previous graph has been extended into multiple (BI, BJ, BE, BH, BG).

has been extruded into several other significant clusters ”BI” ”BJ” ”BE” ”BH” and ”BG.” These clusters represent more specific looks at how students proceed through the assignment. Due to the way we calculate the distance between states, often these transitions between clusters represent the addition of a new method because we heavily weight that function towards class and method declarations. The framework we created would allow for different weightings if future researchers wanted to instead look at creation of variables or method calls more closely.

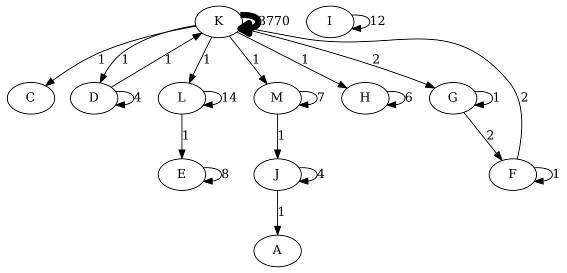

Figure 9: Book Exercise, 12 Clusters. This shows how the majority of student compiles appear in the start state, ”K”.

n each of these digraphs, the groupfis progression through the code is represented by a few different pieces. The arrows from a node back to itself are the number of times a student compiled in a state that fit in that cluster, and the arrows in between clusters show us the number of students that moved between a state in one cluster and a state in another. It is important to note that there is no representation of time in this graph. We broke the data down this way in order to give equal space to a student who compiled only a few times while making large edits to their code and a student who compiled many times but made smaller edits. Both of these hypothetical students would see progression through various clusters, even if they took more time to make it in between them. This leaves us with a collection of different series of student progressions. They all start in a large cluster, representing the starter code provided by the book. Because the agglomerative clustering algorithm focuses on outliers, the first end states we see are the ones that are vastly different from the starter code. For example, a student in a cluster like fiKfi might have added numerous additional methods to their code that werenfit required by the book but may have been required by a specific teacher or that they decided to add for their own personal education or use. In a simpler exercise like fiBook Exercisefi from the BlueJ textbook, even though the factoring of student code is more straightforward than it might be in the more complicated fiAuction,fi it is easier to see the clear diverging paths that students took. In (figure number here for book 13), we can see a large collection of students beginning in cluster fiKfi with the starter code provided by the textbook.

```
class Book
{
   // The fields.
   private String author;
   private String title;

   /**
    * Set the author and title fields when this object
    * is constructed.
    */
   public Book(String bookAuthor, String bookTitle)
   {
       author = bookAuthor;
       title = bookTitle;
    }

    // Add the methods here ...
}
```

Then in cluster ”M” the one student that has arrived in this cluster has added a few other methods and fields to their code.

```
class Book
{
    // The fields.
    private String author;
    private String title;
    private int pages;
    private String refNumber;

    /**
     * Set the author and title fields when this object
     * is constructed.
     */
    public Book(String bookAuthor, String bookTitle)
    {
        author = bookAuthor;
        title = bookTitle;
        pages = 0;
        refNumber = " ";
    }

    // Add the methods here ...
    public void getAuthor()
    {
        String author;
    }

    public void getTitle()
    {
        String title;
    }
```

Then at the end of the path that student took, they arrive in cluster ”A” where they have added and fleshed out additional methods even more.

```
    public void printDetails()
    {
        if(refNumber.length() > 0)
        {
            System.out.println("title: " + "author " + pages
                + " refNumber "
            + "borrowed. ");
        }
        else
        {
            System.out.println("title: " + "author " + pages
                + " refNumber "
            + "borrowed. ");
        }
    }

    public void setRefNumber (String ref)
    {
        if(ref.length()>=3)
        {
            refNumber = ref;
        }
        else
        {
            System.out.println("Error.");
        }
    }
```

5 DISCUSSION

We took a different approach to trying to extract meaning from the BlueJ data than the previous researchers investigating novice programmers. While we didn't achieve the same kind of results, we were not trying to. As they were trying to crunch raw data and achieve metrics, our unique clustering approach was trying to group code, something very different. We attempted to extract a more intuitive structure of student movement and development without combing the data for specific instances or using our own personally devised approximations of student code. We feel this visualization tool would be useful in showing how students tackle a problem and move through the solution space.

These differences in approach highlight just how little we know about novice computer science as a whole. The mind uninitiated in computer science is an interesting thing. As computer scientists, we have already trained our brains to think in the common and well-trodden pathways; it is hard for us to imagine not to use them. Though our field defined and paved the way for data sciences in every other field, we have yet to really solve the issue of developing better data for pedagogical uses.The BlueJ database in terms of size and breadth is one of the larges repositories of novice programing data collections in existence, and yet there is not much interest or drive from those in the field to truly make use of it, or replicate it's scheme anywhere else. Computer science compared to its peer fields is a new one. We do not have the same kind of history of pedagogy on par with other fields. That's not to say research hasn't been done, nor that the BlueJ undertaking isn't worthwhile. Great strides have been made in recent years in the terms of researching novice programmers, however we have not bee able to truly apply great data science to unveil their mysteries. It is of our opinion that such an endeavour to create an equally sized and more accessible database for easier collaboration amongst computer scientists. It is unclear to us why BlueJ appears so unpopular, although we feel the accessibility of the data could be at fault. The size and scope of the BlueJ blackbox database schema is truly incomprehensible at first glance[1]. Although the software distribution and accumulation is expertly handled, massaging of the data or certain data sets such that they were in a more readily available and manipulable format might prove useful to researchers working in the subject.

6 FUTURE WORK

One major area of possible future work is in data collection and data filtering. By the nature of our tool in its current state, the output it produces only visualizes student paths which stray the farthest from the global trends, the outliers of any assignment. Creating an scheme to reject outlier states before they are clustered would allow researchers to more easily identify popular paths through an assignment's solution space. Removing outliers before the clustering stage would make the visualizations produced show higher volumes of movement between key states, unlike the single-student paths shown on Figure 13 for example.

Outlier rejection presents a significant challenge because any quantitative outlier rejection scheme will involve human knowledge of the assignment, and likely an arbitrary threshold. One approach could be to analyze the distribution of all states and reject those which are d deviations away from the mean. Event his simple scheme however, requires a human to assess how the assignment's typical solutions are transformed by the chosen coordinate transformation function, and what an acceptable "distance" in state space is. The distribution of states is also highly dependent on the coordinate transformation function as well, so the scheme for rejecting outliers using functions which take into account the number of method declarations will likely be much different than schemes using coordinate transformations which do not take declarations into account.

The coordinate transformation function itself (see section 3.2.2) presents another entry point for possible future work. The coordinate transformation we created used a 5-dimensional space, with a dimension for class declarations, method declarations, method calls, variable declarations, and assignment statements. However, this can be easily modified to work in a different-dimensional space, or with different program features. While our transformation function biased our analyses to consider differences in code factoring, one could swap out our function for a function with dimensions for while and for loops, if and switch control structures, and comparisons to bias the analysis towards program logic rather than source structure.

Our concept of the coordinate transformation function itself can also be built upon. Rather than using a naïve counting approach to generating coordinates, one could look into alternative methods of calculating the distance between two source code states. Integrating a performant AST edit distance algorithm instead of our naïve Euclidean distance is an interesting thread to follow.

The agglomerative clustering algorithm itself is a large weakness of our approach in terms of execution time. Looking into a divisive rather than agglomerative clustering algorithm may lead faster execution time as our analysis is only concerned with the largest state clusters.

7 CONCLUSION

We collected and abstracted student code from the BlueJ database. We clustered and organized the result into a series of state-based-diagrams informing the movement of the students. We explored the possibilities of code abstraction and visualization. The BlueJ blackbox database is a powerful collection of student data, perhaps the most powerful one in existence. The possibilities for illustrating and determining the behaviors of novice students from it is seemingly endless. Alas so much of its potential is untapped.We made a noble attempt to sift through that data and produce a coherent mapping of student progress and development on a given assignment. While we ran into our fair share of problems, our theory we believe is sound. With more time and drive, and more selective datasets, labeling and deciphering such student motion would be possible.

ACKNOWLEDGMENTS

The authors would like to thank the University of Kent for providing access to their BlueJ Blackbox database.

REFERENCES

[1] 2015. *BlueJ Blackbox Data Collection Researchers' Handbook.*
[2] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15).* ACM, New York, NY, USA, 522–527. https://doi.org/10.1145/2676723.2677258
[3] David J. Barnes and Michael Kolling. 2008. *Objects First With Java: A Practical Introduction Using BlueJ* (4 ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.
[4] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. 2016. Control-Flow-Only Abstract Syntax Trees for Analyzing Students' Programming Progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16).* ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/2960310.2960326
[5] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12).* ACM, New York, NY, USA, 153–160. https://doi.org/10.1145/2157136.2157182

A APPENDICIES

A.1 Output from `book-exercise`

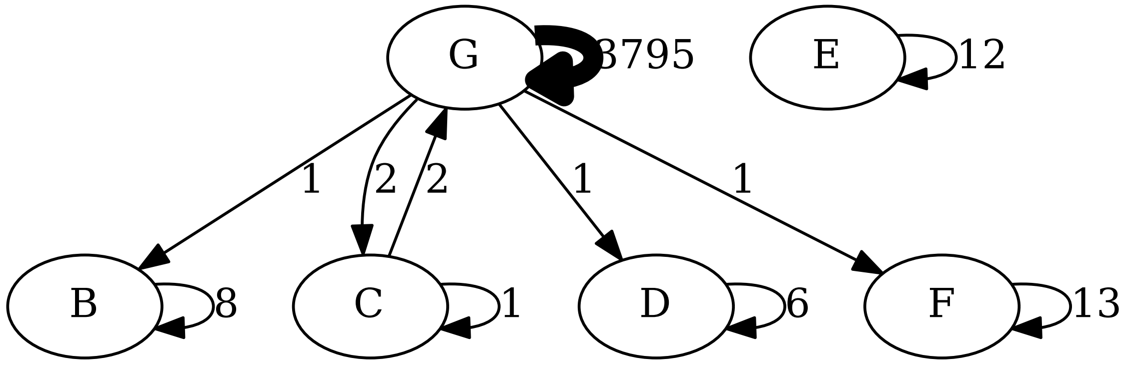

Figure 10: Output visualizing 369 user projects named `book-exercise`

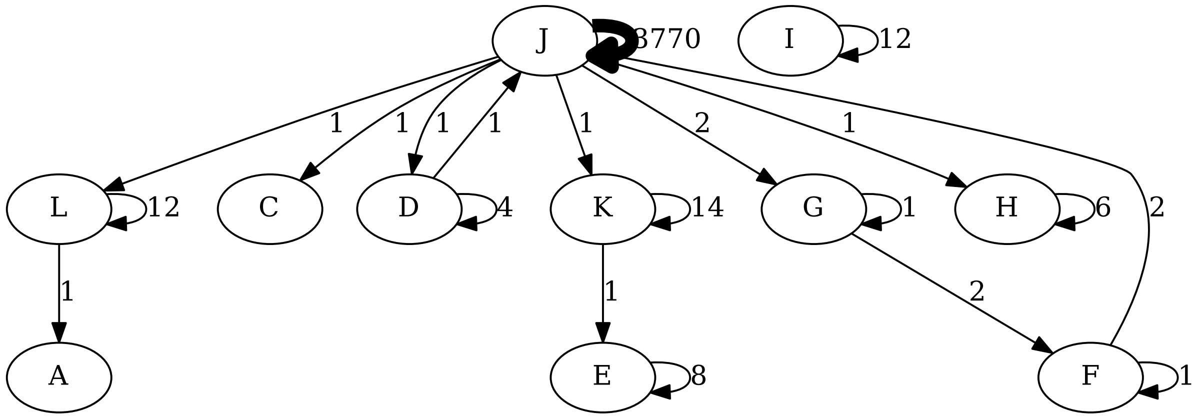

Figure 11: Output visualizing 369 user projects named `book-exercise`

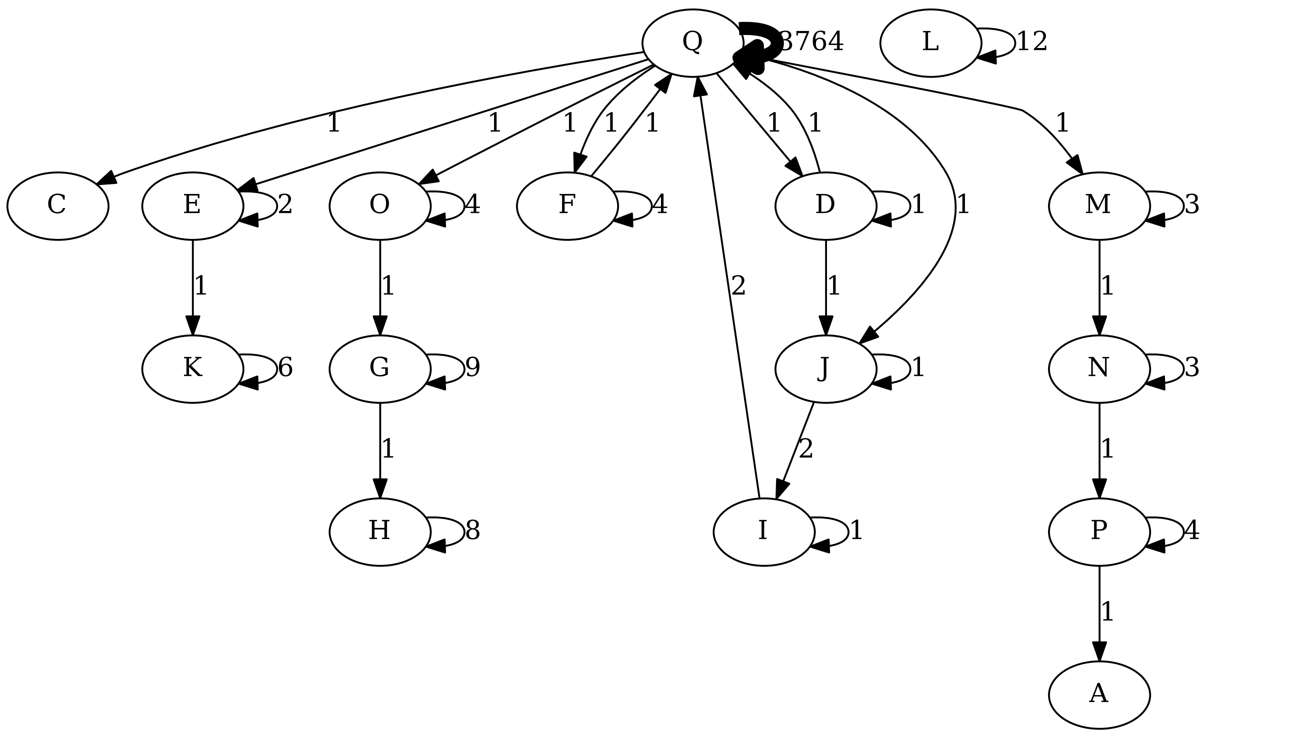

Figure 12: Output visualizing 369 user projects named `book-exercise`

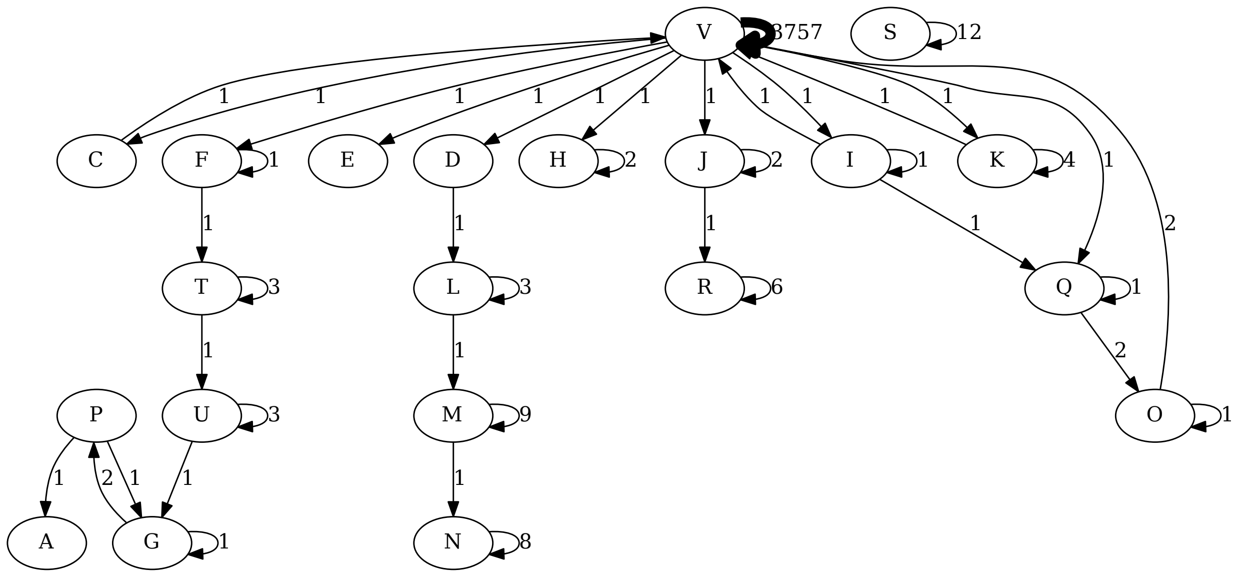

Figure 13: Output visualizing 369 user projects named `book-exercise`

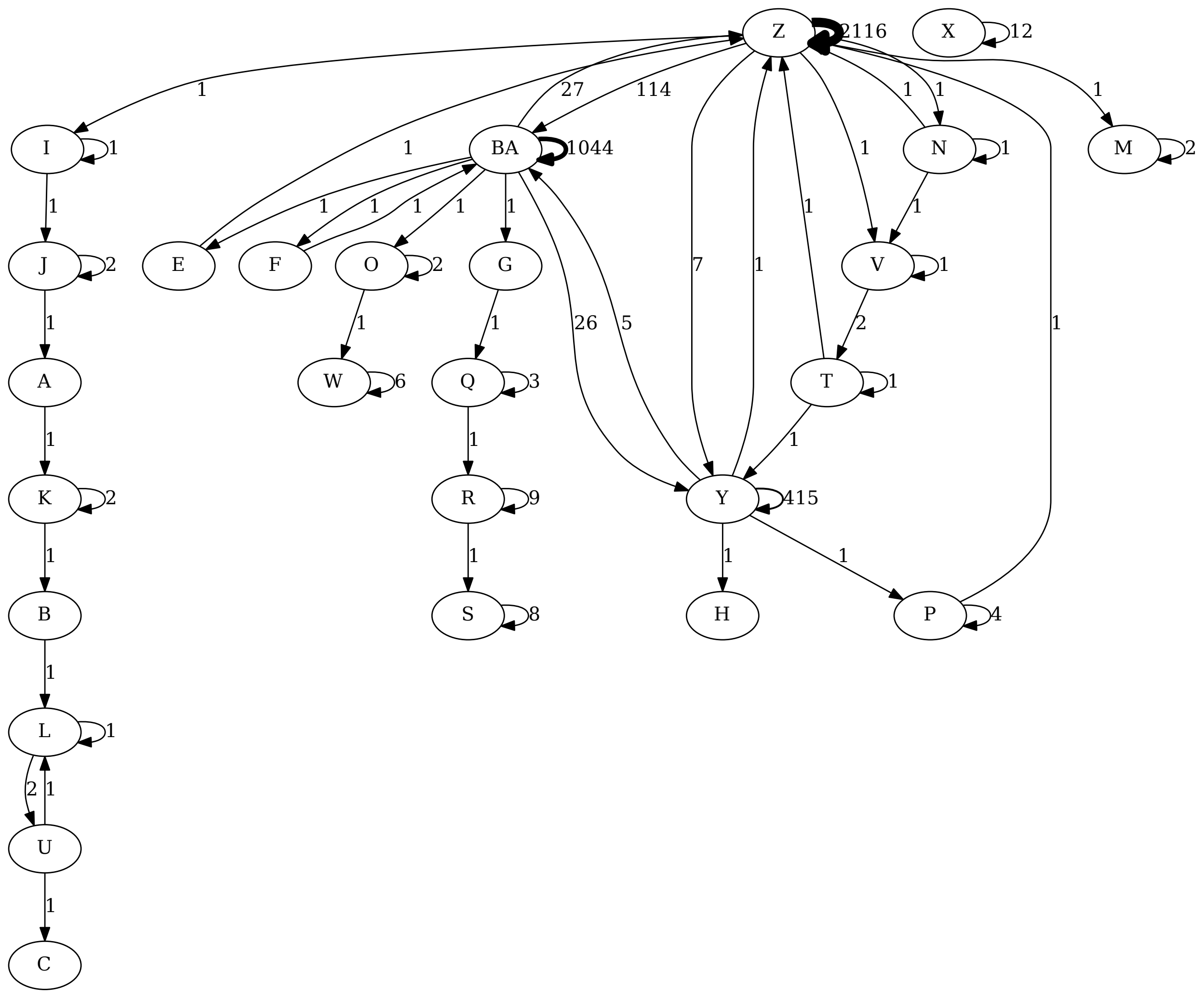

Figure 14: Output visualizing 369 user projects named `book-exercise`

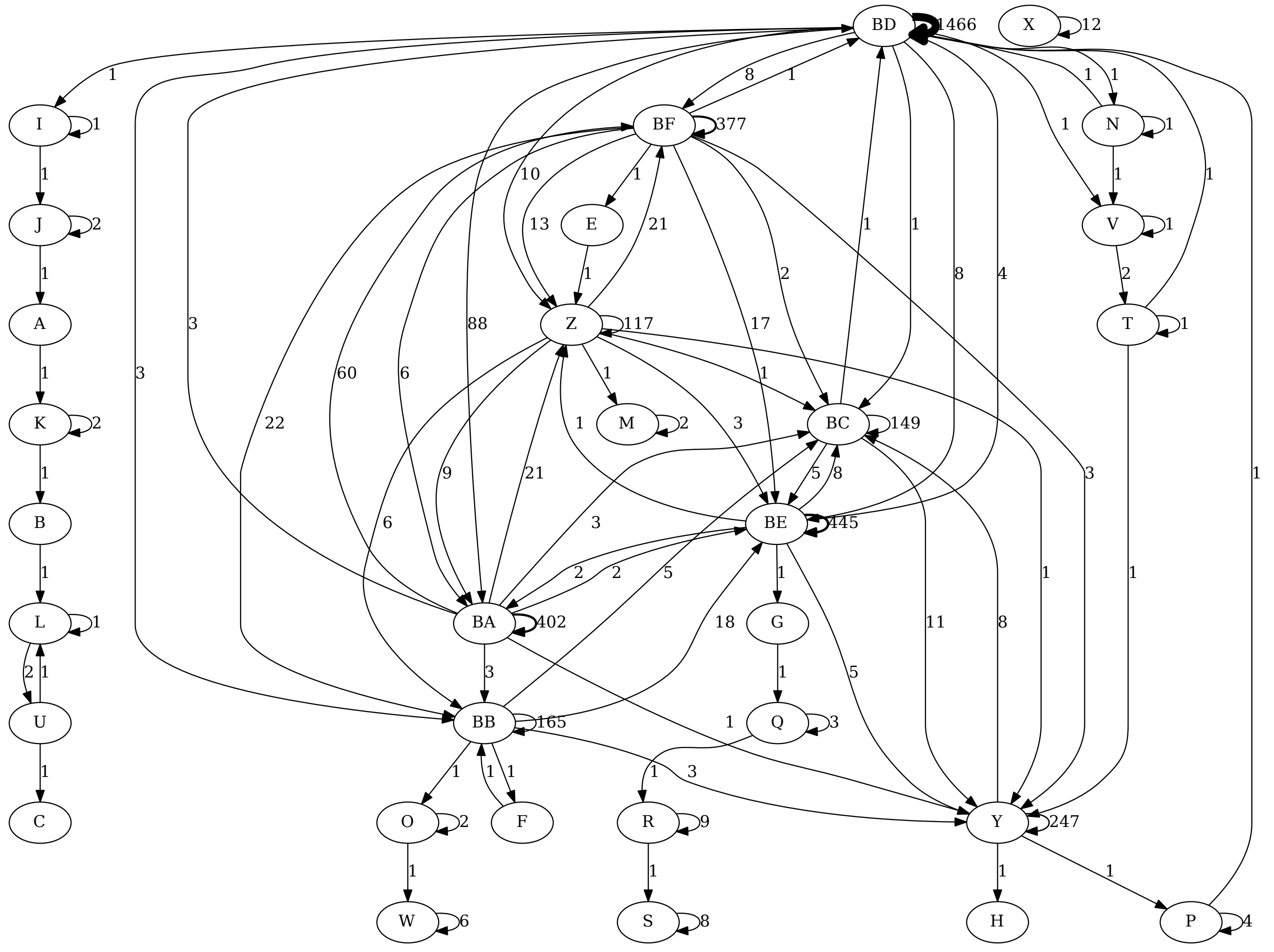

Figure 15: Output visualizing 369 user projects named `book-exercise`