Using Edit Distance Trails to Analyze Path Solutions of Parsons Puzzles

Salil Maharjan
Ramapo College of New Jersey
smaharj3@ramapo.edu

Amruth N. Kumar
Ramapo College of New Jersey
amruth@ramapo.edu

ABSTRACT

We propose edit distance trail as a representation for analyzing the behavior of students solving Parsons puzzles. The edit distance of a student's solution from the correct solution of a Parsons puzzle gives the degree of correctness of the student's solution. Edit distance trail of a student's solution is the chronological sequence of edit distances of the student's solution from correct solution after each puzzle-solving action. We used edit distance trail representation to analyze the puzzle-solving behavior of students who used a Parsons puzzle tutor on `while` loops. In order to find patterns in student solutions, we applied k-means clustering with elbow method. We found that the centroid curves of the clusters of complete solutions differed by slope, corresponding to the degree of optimality of student solutions. Students found the final few steps of the solution to be more challenging. Centroid curves of clusters of incomplete solutions separated informed attempts from uninformed attempts and identified when students hit a dead-end. We discuss the advantages and drawbacks of our representation as compared to aggregate graphs used in literature and how edit distance trails can provide insight not afforded by descriptive statistics.

Keywords

Edit-Distance, K-means clustering, Patterns in puzzle solutions

1. INTRODUCTION

In a Parsons puzzle [4], the student is given a program with its lines scrambled and asked to reassemble the lines in their correct order. The student is also asked to delete one or more distracters – lines of code that do not belong in the program. These puzzles are rapidly gaining popularity in introductory programming courses. Students preferred solving Parsons puzzles to answering multiple choice questions or writing code in electronic books [3]. Educators like Parsons puzzles because solving puzzles takes significantly less time than debugging code or writing equivalent code, but in one study, it resulted in the same learning performance and retention [2]. Scores on the puzzles were found to correlate with scores on code-writing exercises in another study [8]. Software to administer Parsons puzzles have been developed for programming languages such as Turbo Pascal [4], Python (e.g., [7,10]) and C++/Java/C# [1].

The sequence of actions taken by students to solve Parsons puzzles can potentially yield insight into their puzzle-solving strategies, just as similar analysis has been proposed for code-writing tasks (e.g., [5]). If patterns can be found in how students go about solving the puzzles, the patterns may in turn be used to predict the likelihood that a student can successfully solve a puzzle, and to provide customized feedback that helps a struggling student get back on track to correctly solve a puzzle. In other words, analyzing how students solve these puzzles could be beneficial to educators, students and researchers.

Whereas the solution to a Parsons puzzle is a state, the sequence of actions taken to solve a puzzle is a path. To date, to the best of our knowledge, only one study has been carried out to analyze the path taken by students to solve Parsons puzzles [6]. In the study, researchers built a visualization of the solution paths used by students and found wide variance among student solutions. They built aggregate graphs of all the solution paths for each puzzle. In the graphs, nodes were puzzle states, the size of each node being proportional to the number of students who had visited that state. The nodes were color-coded based on correctness. Similarly, edges represented state transitions in student solutions, with the width of each edge proportional to the number of student solutions that included the transition. The researchers found sub-optimal puzzle-solving behaviors such as backtracking and circular looping that could be targeted with customized feedback.

This analysis based on puzzle-states can yield puzzle-specific patterns, such as the statement(s) in a puzzle that students have the most trouble assembling. But, since a puzzle with n statements can have n! states, the aggregate graph of the puzzle can be sparse, making it harder to find patterns in student solutions. For the same reason, this approach does not scale well to larger puzzles, i.e., puzzles with more lines of code.

2. EDIT DISTANCE TRAILS

As an alternative to aggregate graphs, we propose to use **edit distance trail**. An edit distance trail is the sequence of edit distances of student's solution from correct solution, one edit distance per action taken by the student to solve the puzzle. In other words, it is a record of edit distances of the student's partial solution from the correct solution from start to finish. In order to find patterns in student solutions of a puzzle, we propose to use **k-means clustering** of edit distance trails with elbow method for determining the value of k.

The operations allowed in a Parsons puzzle are 1) insertion of a statement into the solution 2) deletion of a statement from the solution and 3) reordering of a statement within the solution. The edit distance of a student's solution from the correct solution is

Salil Maharjan and Amruth Kumar "Using Edit Distance Trails to Analyze Path Solutions of Parsons Puzzles" In: *Proceedings of The 13th International Conference on Educational Data Mining (EDM 2020)*, Anna N. Rafferty, Jacob Whitehill, Violetta Cavalli-Sforza, and Cristobal Romero (eds.) 2020, pp. 638 - 642

the number of these actions necessary to reach the correct solution from the student solution.

In order to calculate edit distances, we modified Levenshtein's algorithm [11]. Levenshtein's algorithm calculates edit distance based on three operations: insertion, deletion and substitution. Since substitution is not an operation permitted in Parsons puzzles, but reordering is, we modified the algorithm to eliminate substitution and incorporate reordering operation.

Edit distance trail of a student on a puzzle is the chronological list of edit distances of the student's partial solution from the correct solution after each action taken by the student to solve the puzzle.

- The starting edit distance of an empty student solution from the correct solution is equal to the number of lines in the puzzle. So, every edit distance trail starts with a value equal to the number of lines in the puzzle.
- When the student's solution is complete and correct, its edit distance from the correct solution is 0. So, every edit distance trail of a completed solution ends with 0.
- The length of the trail is one more than the number of actions taken by the student to solve the puzzle, the extra element corresponding to the start edit distance before the student has taken the first action to solve the puzzle.
- Since our modified Levenshtein's algorithm to compute edit distance treats insertion, deletion and reordering as single-cost operations, each insertion and deletion action increases or decreases the edit distance by exactly 1. A reordering action may change the edit distance by 0 if reordering is incorrect, or 1 if correct.
- If the student inserts a distracter into the solution, edit distance increases by 1.

Unlike the combinatorially explosive number of states in aggregate graphs [6], the length of edit distance trail is linear in the number of actions taken by the student to solve the puzzle. The result of this smaller state space is greater overlap among student solutions, making patterns in student solutions easier to find. Since edit distances abstract away puzzle-specific details such as program states and individual lines in a puzzle, edit distance trails are also amenable to comparison across puzzles.

3. A STUDY OF EDIT DISTANCE TRAILS

We used edit distance trails to analyze the data generated by a suite of tutors on Parsons puzzles called epplets (epplets.org) [1]. The tutors are adaptive and use pretest-practice-post-test protocol – every student solved all the pretest puzzles, but the tutors adaptively selected practice and post-test puzzles based on the learning needs of the student. Students used a drag-and-drop interface to solve puzzles.

For this study, we analyzed the data collected by an epplet on `while` loops. In the epplet, during pretest, students solved Parsons puzzles on the following concepts:

1. A puzzle containing a single while loop. The problem (id 2005) on which the puzzle was based was: "A program that reads numbers till the same number appears back to back. It prints the first number to appear twice back to back (e.g., 4 appears back to back in 3,7,5,7,4,4,5 and is printed)."
2. A puzzle containing nested while loops, the inside while loop's condition dependent on the execution of the outside while loop, The problem (id 2105) on which the puzzle was based was: "A program that repeatedly reads a positive number, reads additional numbers till its multiple is found, and prints the number and its multiple. It repeats this until 0 or negative value is entered for the number. For example, while reading the sequence 3,2,4,6,2,5,4,0 it prints 3,6 and 2,4."

The tutor was used by introductory programming students as after-class assignments. For this study, we used the data collected by the tutor over eight semesters: Spring 2016 – Fall 2019. We included data from only the students who gave permission for their data to be used for research purposes. Students used the tutor in four different languages: C, C++, Java and C#. We combined the data from all four languages in our analysis. Students could use the tutor as often as they wished. When a student used the tutor multiple times, data from all the sessions was included in the study. In all, 1068 students used the tutor during those eight semesters.

Epplets log the sequence of puzzle-solving actions taken by each student. We processed these logs to reconstruct the partial solution after each action and compute the edit distance of the partial solution from the correct solution using modified Levenshtein's algorithm. After computing the edit distance trail corresponding to each sequence of puzzle-solving actions, we used k-means clustering to find patterns in the edit distance trails of the two puzzles (ids 2005 and 2105) separately. Within each puzzle, we analyzed edit distance trails of complete and incomplete solutions separately. The number of edit distance trails available for each puzzle and the optimal number of clusters found for each puzzle for complete and incomplete solutions are listed in Table 1.

Table 1. Number of Edit Distance Trails Available and Optimal Number of Clusters Found for each Puzzle

Puzzle No. (Id)	Complete Solutions		Incomplete Solutions	
	Trails	Clusters	Trails	Clusters
1 (2005)	532	3	239	4
2 (2105)	180	3	153	4

3.1 Puzzle 2005

The clusters found for complete solutions of puzzle 2005 are shown in Figure 1, along with their centroids, which are themselves trails. Table 2 lists the three clusters, number of solutions in each cluster, and the minimum, maximum and mean number of actions taken in those clusters to solve the puzzle.

The puzzle contained 13 lines of code and 2 distracter lines. So, all the centroid curves in Figure 1 start at 13. Data points in the figure at 14 or 15 correspond to the start of trails in which students inserted one or both distracters into the solution before inserting any lines of code that actually belonged in the solution. In the figure, each data point is part of one or more trails – when a data point is shared among trails of different clusters, the colors of the different clusters have blended.

Since a puzzle with n lines can be optimally solved with n actions, cluster 1 (leftmost centroid curve in Figure 1) with a mean of 17.20 actions included all the optimal solutions. The centroid curves of the other two clusters have shallower slopes, corresponding to the use of more actions than necessary to solve the puzzles.

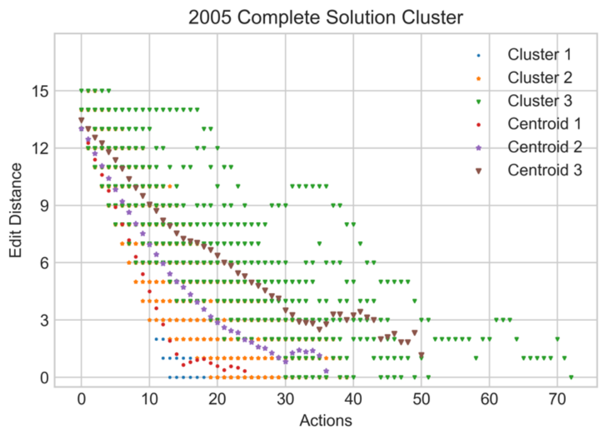

Figure 1. Clusters of Complete Solutions of Puzzle 2005.

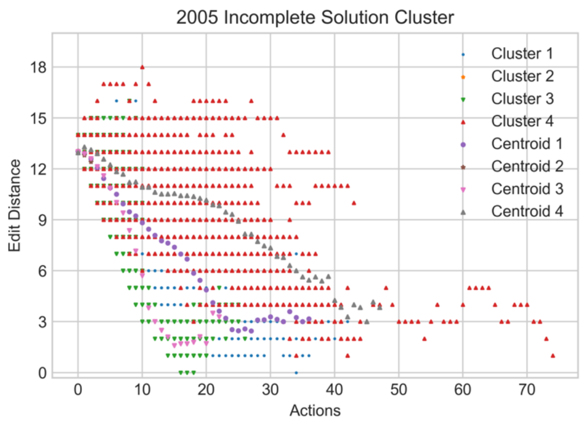

Figure 2. Clusters of Incomplete Solutions of Puzzle 2005.

Table 2. Complete Solution Clusters of Puzzle 2005 (13 lines): Number of trails, minimum, maximum and mean actions taken to solve the puzzle

Cluster Number	N	Actions to Solve the Puzzle		
		Minimum	Maximum	Mean
1	383	16	28	17.20
2	112	20	40	27.25
3	24	31	73	40.12

Table 3. Incomplete Solution Clusters of Puzzle 2005 (13 lines): Number of trails, minimum, maximum and mean actions taken to solve the puzzle

Cluster Number	N	Actions to Solve the Puzzle			Mean final distance
		Min	Max	Mean	
1	78	12	43	21.05	7.05
2	65	1	3	1.50	12.95
3	53	4	27	9.90	9.58
4	38	23	75	32.18	8.63

The clusters found for incomplete solutions of the first puzzle are shown in Figure 2. Table 3 lists the number of incomplete solutions in each of the four clusters, the minimum, maximum and mean number of actions taken in the solutions of the clusters and the mean of the final edit distance of all the solutions in the cluster. The final edit distance shows how many more actions were necessary to complete the solution.

Cluster 2 corresponds to student who bailed out after a maximum of 3 actions. It is likely that these students were familiarizing themselves with the user interface of the puzzle and planned to return to use it in seriousness later. Cluster 3 (leftmost centroid curve) comprised of students who made quick progress (mean of 9.90 actions), but reached a plateau at the end before bailing out. They had an average of 9.58 steps left to complete the puzzle. Cluster 1 (second centroid curve from the left) comprised of students who made gradual progress towards the solution (mean of 21.05 actions) before bailing out. The students in this cluster took more actions to solve the puzzle than students in cluster 3, but got closer to the complete solution. Cluster 4 (rightmost centroid curve) was comprised of students who were lost from the beginning. Note that the *slopes of the centroid curves of incomplete solution clusters provide qualitative information about incomplete solution attempts in the cluster*: attempts that were informed (steep slope) versus those that were uninformed and included a lot of redundant actions (shallow slope), and the point at which attempts in a cluster hit a dead-end (plateau).

3.2 Puzzle 2105

Figure 3 and Table 4 show the clusters found among complete solutions of Puzzle 2105, which contained 16 lines of code and 2 lines of distracters. So, complete solution edit distance trails started with a value in the range 16-18 and ended with 0.

The leftmost centroid curve corresponds to cluster 1, which contains all the optimal solutions, and yet, has an average of 22.2 actions. The rightmost centroid curve corresponds to cluster 2, wherein, students were able to solve the puzzle but took almost twice as many actions as cluster 1. The middle centroid curve corresponds to cluster 3, which included students between the other two clusters in terms of the mean number of actions taken. *Clustering of complete solutions resulted in centroid curves with varying slopes, corresponding to solutions at different levels of optimality.*

Table 4. Complete Solution Clusters of Puzzle 2105 (16 lines): Number of trails, minimum, maximum and mean actions taken to solve the puzzle

Cluster Number	N	Actions to Solve the Puzzle		
		Minimum	Maximum	Mean
1	112	20	33	22.22
2	36	33	66	44.58
3	42	26	42	33.71

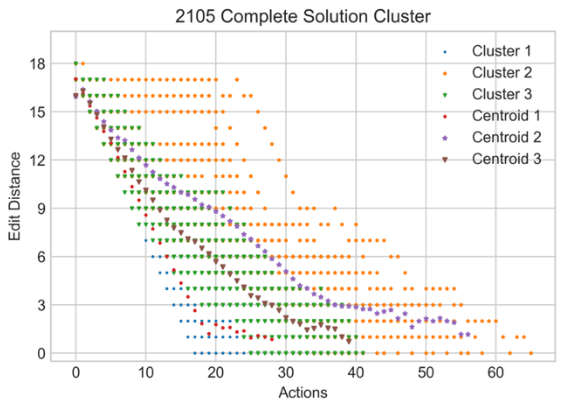

Figure 3. Clusters of Complete Solutions of Puzzle 2105.

Figure 4 and Table 5 show the clusters found among incomplete solutions of puzzle 2105. Cluster 1 represents students who bailed out early, with a maximum of 4 actions. Students in cluster 4 (middle centroid curve) got closest to the correct solution while taking more than twice the number of actions needed to optimally solve the puzzle. Students in cluster 2 (rightmost centroid curve) were less-prepared than those in cluster 4: they took nearly 50% more actions, but were still not as close to the final solution when they bailed out.

Table 5. Incomplete Solution Clusters of Puzzle 2105 (16 lines): Number of trails, minimum, maximum and mean actions taken to solve the puzzle

Cluster Number	N	Actions to Solve the Puzzle			Mean final distance
		Min	Max	Mean	
1	54	1	4	1.24	17.05
2	18	34	84	48.50	10.11
3	26	5	27	10.30	13.34
4	53	18	51	33.88	7.07

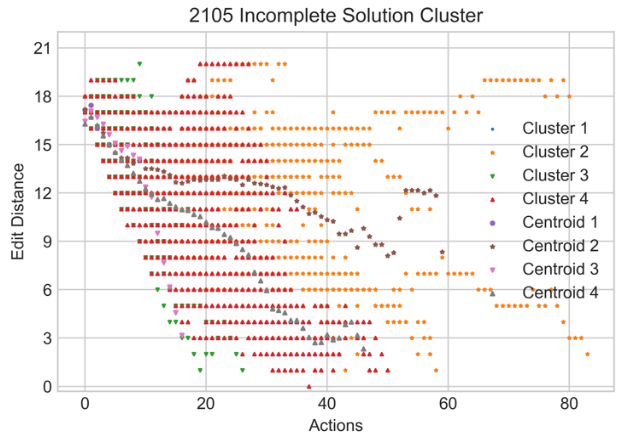

Figure 4. Clusters of Incomplete Solutions of Puzzle 2105.

4. DISCUSSION

Analysis of complete solutions of both the puzzles yielded three clusters corresponding to different levels of optimality of the solution: one cluster corresponding to optimal solutions, and the other two differing in the number of unnecessary actions taken by students to solve the puzzle. Analysis of incomplete solutions yielded four clusters, one of them corresponding to "lurkers" – students who just tried a few actions before bailing out. Lurkers are similar to "stoppers" identified in literature [9] – students who do not take any actions once they encounter a problem, although we believe lurkers were probably just testing the interface. "Movers" identified in literature [9] were all the students in complete solution clusters who were able to solve the puzzle by gradually taking steps towards the correct solution. "Tinkerers" [9] were students in incomplete solution clusters who tried to solve the problem by making small changes in the hopes of making it work.

Edit distance trails can be used to further analyze the behavior of movers and tinkerers. For instance, all the centroid curves in complete solution clusters show a tail at the end of a steep slope. This suggests that even movers who make steady progress solving a puzzle find the last few steps to be more challenging. One possible explanation is that at the end of solving a Parsons puzzle, the student is left with only one or two lines to insert, but the number of locations where they can be inserted are the most ever, making the final steps more challenging.

This illustrates an example of when edit distance trails are more informative than descriptive statistics such as the number of steps taken to solve a puzzle: even in a monotonically decreasing edit distance trail, a change in slope may hint at a moment of frustration (transition from slope to plateau) or insight (transition from plateau to slope). A non-monotonic trail with frequent up- and down-swings may suggest the use of trial-and-error approach. Such differences may be found in the trails of two different students even when they may have taken the same number of steps to solve a puzzle.

Edit distance trail representation is at a more abstract level than aggregate graph representation reported in literature [6]: using edit distances eliminates puzzle-specific details such as the specific line of code acted upon at each instant by a student. So, aggregate graph representation is better at unearthing puzzle-specific patterns such as determining the specific lines of code that most students might have problems assembling correctly. Edit distance trail representation on the other hand makes it easier to identify patterns among solutions – optimal versus sub-optimal complete solutions, lurking behavior, etc. because of its smaller state space. In the future, with the accumulation of additional data, we hope to find more patterns among complete and incomplete solutions that will provide more qualitative information about the types of solutions.

5. ACKNOWLEDGMENTS

Partial support for this work was provided by the National Science Foundation under grant DUE-1432190.

6. REFERENCES

[1] Amruth N. Kumar. 2018. Epplets: A Tool for Solving Parsons Puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18).*

ACM, New York, NY, USA, 527-532. DOI: https://doi.org/10.1145/3159450.3159576.

[2] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. 2017. Solving Parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17).* ACM, New York, NY, USA, 20-29. DOI: https://doi.org/10.1145/3141880.3141895.

[3] Barbara J. Ericson, Mark J. Guzdial, and Briana B. Morrison. 2015. Analysis of Interactive Features Designed to Enhance Learning in an Ebook. In *Proceedings of the eleventh annual International Conference on International Computing Education Research (ICER '15).* ACM, New York, NY, USA, 169-178. DOI: https://doi.org/10.1145/2787622.2787731.

[4] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06), Denise Tolhurst and Samuel Mann (Eds.), Vol. 52. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157-163.

[5] Hosseini, Roya & Hellas, Arto & Brusilovsky, Peter. (2014). Exploring Problem Solving Paths in a Java Programming Course.

[6] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How do students solve parsons programming problems?: an analysis of interaction traces. In *Proceedings of the ninth annual international conference on International computing education research* (ICER '12). ACM, New York, NY, USA, 119-126. DOI: https://doi.org/10.1145/2361276.2361300

[7] Nick Cheng and Brian Harrington. 2017. The Code Mangler: Evaluating Coding Ability Without Writing any Code. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17).* ACM, New York, NY, USA, 123-128. DOI: https://doi.org/10.1145/3017680.3017704.

[8] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08).* ACM, New York, NY, USA, 113-124. DOI=http://dx.doi.org/10.1145/1404520.1404532.

[9] Perkins, D. & Hancock, Chris & Hobbs, Renee & Martin, Fay & Simmons, Rebecca. 1986. Conditions of Learning in Novice Programmers. Journal of Educational Computing Research. 2. 10.2190/GUJT-JCBJ-Q6QU-Q9PL.

[10] Petri Ihantola and Ville Karavirta. 2010. Open source widget for parson's puzzles. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education (ITiCSE '10).* ACM, New York, NY, USA, 302-302. DOI: https://doi.org/10.1145/1822090.1822178

[11] V.I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *In Soviet physics doklady*, *10*, 8, 707-710.