Analyzing Parsons Puzzle Solutions using Modified Levenshtein's Algorithm

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

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

ABSTRACT

We adapted Levenshtein's algorithm to compute edit distance of student solution from the correct solution in Parsons puzzles with the intention of using edit distance as an estimate of the degree of correctness of a student's solution. We modified the algorithm to eliminate substitution operation, which is not allowed in Parsons puzzles, and include reordering operation which is allowed. We used the solution log data from the tutor to reconstruct each step that the student took to solve the puzzle and applied the modified algorithm to compute the edit distance for each step to generate edit distance trails of student solutions. We used these edit distance trails to represent student solutions and applied k-means clustering to find patterns. The analysis was conducted on the data collected by a tutor on `selection statements` over four years. We found interpretable patterns among complete solutions, of optimal versus sub-optimal solutions, based on the inclusion of optional lines of code. Among incomplete solutions, we found patterns of known puzzle-solving behaviors. Edit distance trails helped identify student patterns regardless of the sequence of individual statements manipulated. However, by being puzzle-independent, they lose the ability to identify puzzle-specific information.

Keywords

Edit-Distance, Modified Levenshtein's Algorithm, K-means clustering, Patterns in puzzle solutions

1. INTRODUCTION

Edit distance is a widely used string similarity measure to quantify how dissimilar two strings are based on the number of operations required to convert one string to another. It can be used to compute the degree of correctness of a student's answer in Parsons Puzzle.

In a Parsons puzzle, the lines of a correct program are scrambled and presented to a student, who is tasked with reassembling the code in its correct order. In this scenario, edit distance is the largest when the student starts solving the puzzle, and reduces to 0 when the solution is complete and correct. So, edit distance is an indicator of how close a student's solution is to the correct solution. The series of edit distances computed after each puzzle-solving step taken by the student provides a progress report of the student solving the puzzle.

Since edit distance is computed in terms of the number of operations needed to covert one string to another, our interest was in finding an edit distance algorithm that would consider the operations (and only the operations) allowed when solving a Parsons puzzle: insertion (when a student inserts another line of scrambled code into the solution), deletion (when the student deletes a line of code from the solution) and reordering (when a student reorders lines of code within the solution). We wanted to use this algorithm to generate edit distance trails of student solutions in order to find interpretable patterns from student log data from the tutor by using clustering techniques.

2. MODIFYING LEVENSHTEIN'S ALGORITHM

The edit 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 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 [9]. Levenshtein's algorithm calculates edit distance based on three operations: insertion, deletion and substitution. We modified the algorithm to eliminate substitution and incorporate reordering operation as substitution is not an operation permitted in the Parsons puzzle tutor, but reordering is.

Levenshtein's algorithm starts with a m x n matrix, where m and n are the sizes of the two strings being compared. It initializes the matrix by filling the first row and column by row/column number and filling the cells row by row using the minimum of the following three states [9], given (i, j) is the matrix index:

1. Insertion: $lev_{a,b}(i, j-1)+1$
2. Deletion: $lev_{a,b}(i-1, j)+1$
3. Substitution: $lev_{a,b}(i-1, j-1)+1$, when ($a_i \neq b_j$)

If the characters being compared are not the same, a unit cost is added. If they are the same, matrix(i, j) = matrix(i-1, j-1). For example, in Table 1, the edit distance to convert character 'B' to 'A' is at matrix (1, 1). It is computed as the minimum of three operations: insertion – cost at matrix (1, 0), deletion – cost at matrix (0, 1), and substitution – cost at matrix(0, 0). The minimum of these three costs is at matrix(0, 0). Therefore, matrix

(1, 1) is set to 0 + 1 = 1 after adding unit cost for substitution operation since 'B' ≠ 'A'.

Omitting substitution operation: We modified the algorithm to remove substitution operation and compute the minimum from only two operations - insertion (matrix (i, j-1)) and deletion (matrix (i-1, j)). To convert character 'B' to 'A' without substitution, we require two operations: deletion followed by insertion, giving us an edit distance of 2. We compute this as the minimum of matrix (0, 1) and matrix (1, 0), and add a unit cost since 'B' ≠ 'A', yielding an edit distance of 2 for cell (1, 1). The algorithm repeats this process for all the cells in row-major order.

Table 1. Levenshtein distance matrix with substitution

	Col.(j)	0	1	2	3
Row(i)	-	-	A	B	C
0	-	0	1	2	3
1	B	1	1	1	2
2	E	2	2	2	2
3	A	3	2	3	3

Table 2. Levenshtein distance matrix without substitution (with trace back)

	Col.(j)	0	1	2	3
Row(i)	-	-	A	B	C
0	-	0	1	2	3
1	B	1	2	1	2
2	E	2	3	2	3
3	A	3	2	3	4

Adding reordering operation using trace back: A reordering operation in Parsons puzzle can be broken down into two consecutive operations – insertion and deletion:

- Insertion followed by a deletion of the same character later in the string is a moving operation towards the front of the string.
- Deletion followed by an insertion of the same character later in the string is a moving operation towards the end of the string.

To identify reordering operations, we traced back from the end of the matrix (m, n), to the initial position (0, 0) and used a hash map to determine that the insertion and deletion operations had been applied back to back to the same line. If they were, we counted the insertion and deletion operations as one reordering operation.

For insertion operation, the character from the target string (row character) at the current position is used as the key in the hash map and its value is incremented. For a deletion operation, the character from the source string (column character) of which edit distance is to be calculated is used as the key in the hash map and its value is decremented. A constant unit cost is used for each operation.

Table 2 shows the trace back from (m, n) to (0, 0), where the length of both strings is 3. At (3, 3), the minimum value among matrix (3,2) and matrix (2, 3) is 3. Since both cells have the minimum value, either cell can be chosen to visit next. In this example, assume that the cell to the left, i.e., matrix (3, 2) is visited next. This highlights an insertion operation of character 'C'. The hash map is updated with key C and value '1', followed by another insertion of 'B'. At index (3, 1), the character is 'A' in both (row and column) strings. Therefore, the algorithm moves diagonally without any cost. Next, consecutive deletion operations are carried out to trace back to (0, 0). The resulting hash map has the following values:

Table 3. Hash map record of trace back of (Table 2.)

Key	C	B	E	B
Value	1	1	-1	(-1)

Upon reaching matrix (1, 0) (Table 2), a deletion operation of character 'B' is recorded to reach position matrix (0, 0). Since the hash map already has an entry for 'B', and the value for it is positive corresponding to insertion of 'B', the subsequent deletion of 'B' implies a reordering operation of 'B'. Since insertion is followed by deletion, character 'B' is reordered to a later index in the string. The string transformation can be summarized as:

Table 4. Transformation of string "BEA" to "ABC"

Operation	String
(Source string)	BEA
Insertion of C	BEAC
Insertion of B	BEABC
Deletion of E	BABC
Deletion of B (Target string)	ABC

In every case where the entry is positive for a particular key in the hash map and a deletion operation is performed or where the entry is negative and an insertion operation is performed, a reordering operation is identified and the edit distance is decremented by one. As seen in Table 3, operations do not have to be back-to-back. When the operations on a character are back-to-back, it signifies a transposition of the character, i.e., reordering by a single position.

The algorithm has two boundary conditions, when either 'i' or 'j' reaches 0. When j reduces to 0, we have a left column boundary condition: since no insertion operations are possible, the hash map is updated with deletion operation based on the character from the source (row) string. Table 2 exemplifies a column boundary condition. Similarly, when i reduces to 0, we have an upper row boundary condition: since no deletion operations are possible, the hash map is updated by recording insertion operation using the characters in the target (column) string as the key.

Several other extensions of Levenshtein's algorithm have been attempted before. Damerau-Levenshtein algorithm [4] extends Levenshtein's algorithm by considering adjacent character transpositions as another operation. Transposition is a special case of reordering, where the reordering is done by just one character. We needed an algorithm that treated reordering by any number of characters as a single-cost operation.

Several modifications have dealt with the general case of reordering an entire substring. Shapira *et al.* provide a polynomial time greedy algorithm to move substrings [3]. They identify this problem as NP-complete. Since it uses a greedy strategy to handle move operations, the algorithm is unable identify all move cases; it rather gives an approximation.

Comrode and Muthukrishnan provide a general approach that is subquadratic and deterministic called *edit-sensitive-parsing* (ESP) [5]. The algorithm approximates the edit distance with moves in *O(n logn)*. Takabatake *et al.* [8] further optimize the index structure used in the ESP technique to make the algorithm near linear time.

These modifications to accommodate reordering of substrings are more general than what we need in Parsons puzzles, where, only one character is reordered at a time. Because of the NP-complete nature of the problem of reordering substrings, these algorithms approximate edit distance calculations. For our problem, we were interested in exact calculation of edit distance, while restricting the moved substring to a single character, i.e., line of code.

So we simply looked at ways to identify move operations using the dynamically filled matrix generated by Levenshtein algorithm. We incorporate backtracking to the algorithm to effectively track reordering operations in a Parsons's puzzle. Dynamic programming and backtracking are more expensive in terms of time and memory than the approaches mentioned above but we were able to correctly identify reordering operations as single-cost operations in the scope of our problem.

3. COMPUTING EDIT DISTANCE TRAILS

A student's solution of a Parsons puzzle is logged as a sequence of actions such as:

1. Moved from problem to solution at line 7:
   `short firstNum`
2. Moved from problem to solution at line 9:
   `short secondValue;`
3. Moved from problem to solution at line 10:
   `cout << "Enter the first value";`
4. Reordered from line 10 to 12:
   `cout << "Enter the first value";`

The tutor logs the sequence of actions taken by students to solve each puzzle. Student are tasked with solving Parsons puzzles using drag-and-drop actions. We wrote a program to reconstruct the partial solution of the student after each action, the partial solution being the program the student had assembled so far for the puzzle. Next, we computed the edit distance of each partial solution from the correct solution for the puzzle. If a puzzle had multiple correct solutions, we computed the distance of each partial solution from the specific correct solution eventually reached by the student. The resulting edit distance trail of a student for a puzzle with 6 lines of code might look like this:

[6, 5, 5, 4, 5, 4, 3, 2, 3, 2, 1, 0]

Since not everyone solved each puzzle with the same number of actions, the length of the edit distance trail varied from student to student. But, the minimum length of the trail was n + 1 where n was the number of lines in the correct solution of the puzzle.

Figure 1 shows a graphical representation of two edit distance trails. These trails show the progress report of two different students who attempted the 2005 template problem on `if-else` statements. The 2005 template problem asks the student to read two numbers and print the smaller value among them. The student is expected to arrange the scrambled lines of code that contains if-else statements in the correct order.

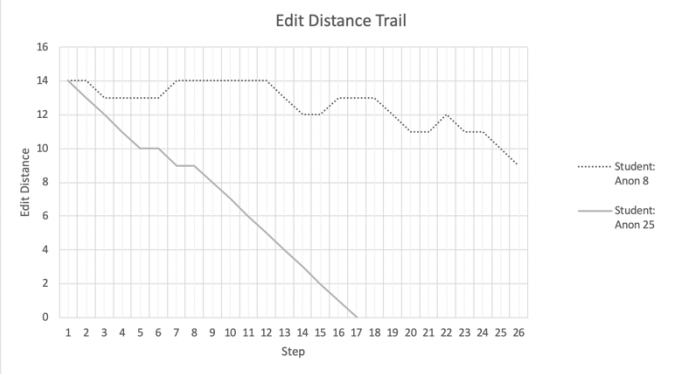

Figure 1. Edit distance trail graphical representation.

The edit distance trail of student "Anon 25" is [14, 13, 12, 11, 10, 10, 9, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] which has a length of 17. Similarly, student "Anon 8" has a longer trail of length 26 that does not converge to an edit distance of value 0. This shows that student Anon25 was able to solve the puzzle in 17 steps whereas student Anon8 took a total of 26 steps but was unable to solve the puzzle.

Using this edit distance trail, we are able to identify the correctness of each action taken by the student. By doing so, we are able to plot the problem solving strategy of various students for different types of problems.

4. AN APPLICATION OF EDIT DISTANCE TRAILS

For this study, we used data collected by a Parsons puzzle tutor on `if-else` statements from a suite of such tutors available online called epplets (epplets.org) [1]. In the tutor, the student was tasked with solving Parsons puzzles using drag-and-drop actions. The student was required to solve each puzzle completely and correctly before going on to the next puzzle. If the student took a lot of redundant actions to solve a puzzle, the tutor scheduled additional similar puzzles for the student to solve. If the student took more than twice as many actions as necessary to solve a puzzle, the tutor offered the student the option to bail out. If a student bailed out, the solution was marked as incomplete and the student was presented additional similar puzzles.

The first 3 puzzles presented by the tutor were on the following programs, listed here along with a unique puzzle id associated with each puzzle:

1. A program to read two numbers and print the smaller value among them (puzzle id 2005).
2. A program to read numerical grade, convert it to letter grade - A (90 and up), B (80-89), C (70-79), D (55-69) and F otherwise - and print it (puzzle id 2105).
3. A program to read a number and print whether it is odd or even (puzzle id 2000).

The first and third puzzle was on if-else statement and the second puzzle was on nested if-else statements. If a student solved the first puzzle with too many redundant actions, the tutor presented

additional puzzles, the first of which was the third puzzle – it was similar to the first puzzle. Each puzzle had two distracters – lines of code that did not belong in the solution.

The tutor was used by our introductory programming students as after-class assignments. For this study, we used the data collected by the tutor over eight semesters: Fall 2015 – Spring 2019. We included data from only the students who gave permission for their data to be used for research purposes. Students could use the tutor as often as they wished. Students used the tutor in four different languages: C, C++, Java and C#. We combined the data from all four languages in our analysis. 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. 275 students withheld permission for use of their data during at least one session (but, may have given permission during other sessions).

In order to find patterns among edit distance trails, we used k-means clustering in scikit-learn Python package. Since edit distance trails were not all of the same length, we padded trails at the end with -1 so that all the trails were of the length of the longest trail for the puzzle. We used elbow method to determine the optimum number of clusters k. k-means algorithm clustered the trails in n-dimensional hyperspace, wherein n was the uniform length of all the edit distance trails. So, the algorithm clustered edit distance trails, not individual edit distances in the trails. After grouping edit distance trails into clusters, we computed centroid curve of each cluster, which represents the pattern or archetype trail of the cluster. For the calculation of centroid curve, we ignored the -1 values used to right-pad the trails so that they would not affect the shape of the curve.

We analyzed the edit distance trails of each puzzle 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 are listed in Table 5. We did not apply clustering algorithm if the number of trails was less than 50.

Table 5: 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)	785	4	371	4
2 (2105)	376	4	356	3
3 (2000)	358	4	26	N/A

4.1 Puzzle 1

The clusters found for complete solutions of the first puzzle are shown in Figure 2 along with their centroids, which are themselves trails. We found that *clustering separated edit distance trails of completed solutions by how optimally students solved the puzzle, i.e., by the slope of edit distance trails*. Table 6 lists the four clusters, number of solutions in each cluster, and the minimum, maximum and mean number of actions taken in those solutions to solve the puzzle. The puzzle contained 14 lines of code and 2 distracters. The 14 lines included two pairs of braces around if-clause and else-clause. Both the pairs of braces were optional since both the clauses contained a single statement. So, the puzzle could have been solved with 14 (both pairs included), 12 (only one pair included) or 10 (neither pair included) lines - All three versions were accepted as correct. Therefore, all the trails start at a value between 10 and 14 in Figure 2. Since a puzzle with n lines can be optimally solved with n actions, cluster 1 (leftmost centroid in Figure 2) with a mean of 17.26 actions included all the optimal solutions.

In the figure, data points at 15 or 16 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. Each data point is part of one of more trails – when a data point is shared among trails of different clusters, the colors of the different clusters have blended. Since our interest was in finding patterns in the trails, i.e., centroid curves, and not distribution of data, we used a regular graph rather than a bubble chart.

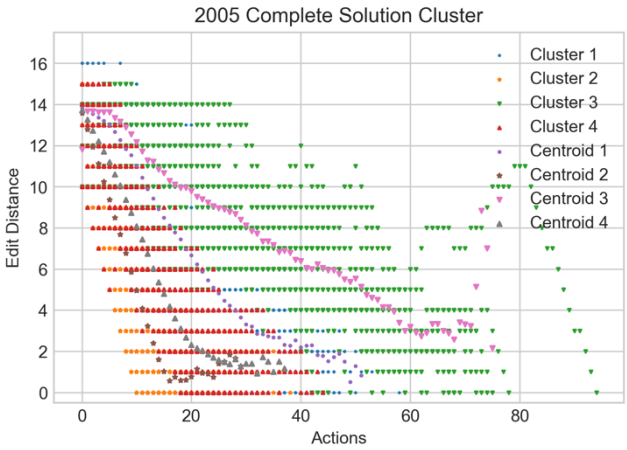

Figure 2. Clusters of Complete Solutions of the First Puzzle

Table 6: Complete Solution Clusters of the First Puzzle: 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	533	16	38	17.26
2	157	18	44	24.66
3	77	28	58	38.81
4	18	42	94	63.88

The clusters found for incomplete solutions of the first puzzle are shown in Figure 3. Table 7 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 would have been necessary to complete the solution.

The first cluster corresponded to students bailing out after just two actions. Since this was the first puzzle presented by the tutor, it is likely that 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 line) comprised of

students who made steady progress (mean of 11.94 actions), but reached a plateau at the end before bailing out. Cluster 2 (second centroid curve from the left) comprised of students who made gradual progress towards the solution (mean of 24.44 actions) before bailing out. Both the clusters bailed out about 8 actions away from solving the puzzle, i.e., they bailed out about halfway through the solution to the puzzle that contained 14 lines. Cluster 4 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 solutions in the cluster*: solutions that were informed (steep slope) versus those that were not informed and included a lot of redundant actions (shallow slope), and the point at which a solution hit a dead-end (plateau).

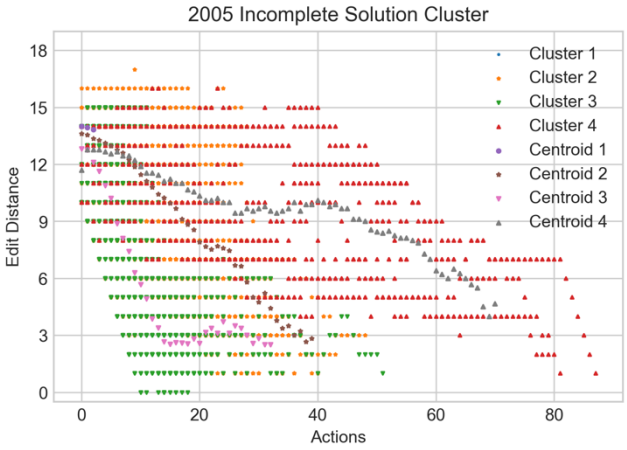

Figure 3. Clusters of Incomplete Solutions of the First Puzzle

Table 7: Incomplete Solution Clusters of the First Puzzle: Number of trails, minimum, maximum and mean actions taken to solve the puzzle and mean final edit distance

Cluster Number	N	Actions to Solve the Puzzle			Mean final distance
		Min	Max	Mean	
1	171	0	2	0.43	13.98
2	92	12	48	24.44	8.29
3	99	3	51	11.94	8.12
4	9	50	87	67.11	4.66

4.2 Puzzle 2

Figure 4 and Table 8 show the clusters found among complete solutions of the second puzzle, which contained 34 lines of code and 2 distracters. 16 of the 34 lines were open or close braces which were optional as explained earlier. So, complete solution edit distance trails started with a value in the range 18-34 and ended with 0.

The main difference between the four clusters was the number of optional braces that were included in the final solution: cluster 1 (third centroid from the left in Figure 4) included all the braces, cluster 2 (leftmost centroid), none of the braces, and cluster 3 and 4 (second and right-most centroid from the left), some of the braces. In this case, *clustering separated edit distance trails of completed solutions by the number of optional lines that were included in the final solution (i.e., the y intercept of edit distance trails at x = 0)*. Cluster 4 comprised of students who took a lot more actions to solve the puzzle than necessary.

Table 8: Complete Solution Clusters of the Second Puzzle: 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	130	36	67	40.75
2	44	36	57	41.45
3	171	36	74	40.67
4	31	50	123	71.48

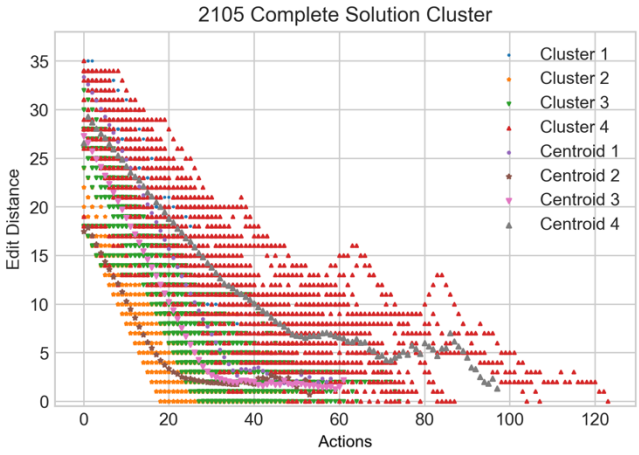

Figure 4. Clusters of Complete Solutions of the Second Puzzle

The clusters found among incomplete solutions of the second puzzle are shown in Figure 5 and Table 9. Cluster 2 comprised of students who bailed out early. Cluster 1 students made rapid progress (leftmost centroid curve in Figure 5), but abandoned the solution about 9 actions short. Cluster 3 students struggled to solve the puzzle (right-most centroid curve in the figure) and bailed out 14 actions short. Once again, we see steep versus shallow slope and plateau – features of the centroid curves that *provide qualitative information* about the solutions in the clusters. The analysis so far supports our hypothesis that *patterns could be found in student solutions of Parsons puzzles that were interpretable.*

Table 9: Incomplete Solution Clusters of the Second Puzzle: Number of trails, minimum, maximum and mean actions taken to solve the puzzle and mean final edit distance

Cluster Number	N	Actions to Solve the Puzzle			Mean final distance
		Min	Max	Mean	
1	130	9	78	43.53	8.89
2	156	0	9	1.05	27.33
3	70	32	138	61.22	14.28

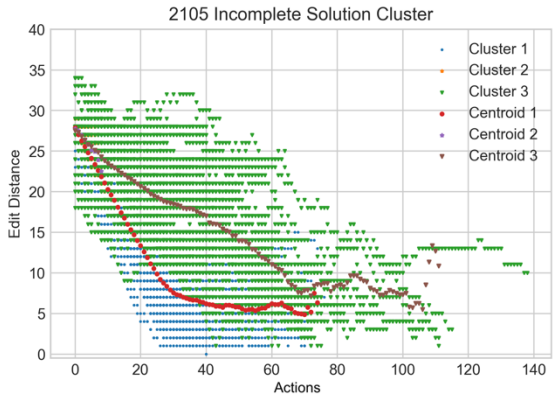

Figure 5. Clusters of Incomplete Solutions of the Second Puzzle

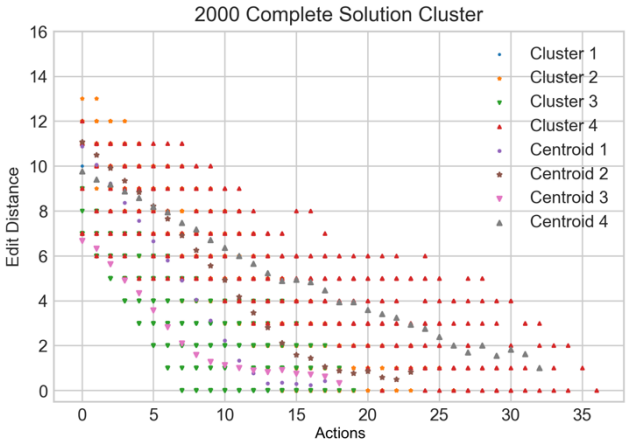

Figure 6. Clusters of Complete Solutions of the Third Puzzle

4.3 Puzzle 3

Figure 6 show the clusters found among complete solutions of the third puzzle. Table 10 lists the minimum, maximum and mean number of actions taken to solve the third puzzle for each complete solution cluster.

The puzzle contained 11 lines of code and 2 distracters. Similar to the first puzzle, puzzle 3 also includes two optional pairs of braces around the if-clause and else-clause. All centroid curves start at a value between 7 and 13 in Figure 6 because of the four optional code lines and two additional distractors. The puzzle could be optimally solved with 11 actions.

In puzzle three, we found that *clustering separated edit distance trails of completed solutions by how optimally students solved the puzzle and also by the number of optional lines that were included in the final solution.* Cluster 3 (left-most centroid curve) grouped students that did not use any optional braces. Figure 6 showed that this centroid curve starts at an edit distance value of 7. Clusters 1 and 2 (second and third centroid from the left) grouped students who use both pairs of optional braces to solve the puzzle. The centroid curve for both these clusters starts at an edit distance value of 11. Furthermore, cluster 1 grouped students who solved the puzzle more optimally with an average of 13.75 actions and cluster 2 grouped students who took more moves, an average of 17.51 actions, to complete the puzzle. Cluster 4 (right most centroid curve) grouped students who used only a single pair of optional braces since their centroid curve starts at a value of 10. Figure 6 showed that this group of students took the most actions to complete the puzzle.

Table 10: Complete Solution Clusters of the Third Puzzle: 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	202	13	18	13.75
2	87	14	26	17.51
3	52	13	19	14.69
4	17	21	36	28.64

The incomplete solutions of the third puzzle contained only 26 trails. So, clustering was not performed on incomplete trails.

5. DISCUSSION

In order to be able to objectively contrast the clusters of a puzzle as well as compare the clusters of different puzzles, we computed the **degree of optimality** of the solutions included in each cluster. A puzzle with n lines needs no more than n actions to solve completely and correctly. So, an optimal solution of the puzzle has the same number of actions as the number of lines in the puzzle. The degree of optimality (O) of the solutions in a cluster is calculated as μ / n, wherein, μ is the mean of the number of puzzle- solving actions taken by all the solutions in the cluster and n is the number of lines in the puzzle.

Table 11 lists the degree of optimality of all complete solution clusters of all three puzzles. Different puzzles may have different number of lines of code. The degree of optimality abstracts away this difference, thereby enabling us to compare clusters of different puzzles. Note that on the first puzzle, the first cluster comprised of the most optimal solutions (Table 6). In Table 10, the first cluster had the lowest degree of optimality for puzzle 3. Clusters 1, 2 and 3 were all optimal for puzzle 2 (Table 8), differing only in the number of optional statements that were included in the final solution. All three clusters of puzzle 2 have similar degrees of optimality in Table 11.

Table 11: The Number of Solutions (N) and Degree of Optimality (O) in each Cluster of each Puzzle

No	Complete Solution Cluster Number							
	1		2		3		4	
	N	O	N	O	N	O	N	O
1	533	1.14	157	1.6	77	2.48	18	4.05
2	130	1.15	44	1.17	171	1.15	31	2.01
3	202	1.13	87	1.42	52	1.2	17	2.28

The complete solutions of all three puzzles yielded four clusters. These clusters corresponded to either various levels of optimality or the number of optional lines that were included in the final solution. The first puzzle required students to assemble a single

block of if-else statement and was the first puzzle that was presented to students. Figure 2 showed that the majority of the trails start at an edit distance value of 14 as all the centroid curves start at the same value. The first puzzle had twice the number of complete solution trails compared to the second and third puzzles (785 vs 376 and 358 respectively). We do not see major variations in the use of optional braces in the first puzzle even though the sample size is much larger. This highlights how most students in the first puzzle used all the optional braces to assemble the solution. Beginner programmers might not be aware that clauses enclosing single statements do not necessarily require braces. Students might be including the optional braces as it is a good practice, or they might just be unaware that the tutor considers braces as optional. This is why the first puzzle is clustered solely based on various levels of optimality.

The third puzzle, which is a follow-up puzzle for students who struggled with the first puzzle showed more variety with the use of optional braces. In the third puzzle, students who used all the optional braces are clustered in clusters 1 and 2, students who used none of the optional braces are clustered in cluster 3, and students who used one of the two optional pairs are clustered in cluster 4. This variety in the use of optional braces might be accounted for by the student's experience in programming or using the tutor. Interestingly, Table 10 showed that a majority of students who used only one of the two pairs of optional braces take the greatest number of actions to solve the puzzle. These students are clustered in cluster 4 and used an average of 28.64 actions to solve the puzzle. Table 11 showed that this cluster had the most non-optimal solutions with a degree of optimality of 2.28.

The second puzzle covered nested if-else statements and is more complex than both the first and third puzzles. This puzzle contained eight pairs of optional braces. Table 11 showed that the most non-optimal solutions were clustered in cluster 4 with a degree of optimality of 2.01. Similar to the third puzzle, cluster 4 in the second puzzle corresponded to students who used some of the optional braces. This observation asserts that students who do not follow one of the two practices (i.e., either including all optional braces or eliminating all optional braces) struggle the most with assembling `if-else` statements.

Furthermore, in the second puzzle, the cluster with the second most non-optimal solutions was cluster 2 with a degree of optimality of 1.17. This cluster grouped students who did not use any of the optional braces to construct the final solution. As this puzzle used nested if-else statements, students might had confused some of the lines of the problem since they did not use any of the optional braces. Optional braces are not necessary, but it improves the readability of the code. This might be the reason why students who did not use any braces took more actions to assemble the puzzle than students who used optional braces. This shows that the use of optional braces helped in solving more complex tasks. Cluster 1 grouped students who used all the optional braces and it had a degree of optimality of 1.15. Even though they assemble a larger puzzle (34 lines vs 18 lines), the solutions were more optimal compared to students who did not use optional braces.

We expect that we may find more clusters if we gather more data for the second puzzle. This might also show optimal and sub-optimal solutions for various types of solutions: 1) solutions with no optional lines included, 2) solutions with all optional lines included, and 3) solutions with some optional lines included.

From a visual inspection of complete clusters of the three puzzles, we find that the centroids in the second puzzle had a longer tail than the centroids in the first puzzle. This showed that the students in all the clusters of the second puzzle faced more difficulty completing the last few steps in the puzzle compared to the students who solved the first puzzle. All three puzzles cover `if-else` statements, but the second puzzle is more difficult than the first and third puzzles because it involves the concept of nesting, which is harder for novice programmers. We have found similar tails in edit distance trails of harder puzzles during the analysis of other concepts [7].

Additionally, Table 11 showed that the second puzzle had a maximum optimality 2.01. This means that the students took at most twice the optimal number of moves to try and solve the second puzzle. The first puzzle included a group of students who took about four times the required moves to solve the puzzle (cluster 4 with a degree of optimality of 4.05). This highlights that students were more motivated to solve the first puzzle. Students most likely gave up more quickly on the second puzzle because of the added nesting complexity.

Among the incomplete solutions, the first cluster in the first puzzle and the second cluster in the second puzzle, shown in Table 7 and Table 9, identifies "lurkers" [6]. Lurkers are students who take a couple of actions and bail out quickly. Hosseini in her literature identifies lurkers as "stoppers", who do not take any actions after encountering a problem. We use the term "lurkers" because we believe that these students were probably just testing the interface to gain familiarity with the tutor. "Movers" identified in the literature [6] corresponded to all the students grouped in the complete clusters. These students were gradually able to solve the puzzle by taking steps towards the correct solution. "Tinkerers" [6] were students who took several actions to solve the puzzle by making small changes but were ultimately unable to solve the puzzle. All the other clusters in the incomplete solutions excluding the "lurkers" identify as "tinkerers". Edit distance trails in this case helped identify the known problem-solving behaviors of students.

Edit distance space has been used to generate hints in code-writing tasks [2]. We use edit distance to track progress of students, not provide hints; we addressed Parsons puzzle solutions which have a finite search space and a single correct solution compared to code-writing exercises which must accommodate any code written by the student and can have multiple possible solutions; and we computed edit distance from the final and only solution to the puzzle, not a weighted average of all possible nearby paths to the solution.

Edit distance trails helped to identify patterns by clustering student solutions regardless of the sequence of individual statements manipulated by them. One shortcoming of using edit distance trails is that since they abstract away puzzle-specific information, they cannot be used to determine the specific lines of code that most students might have problems assembling correctly.

In the future, we plan to analyze data from Parsons puzzle tutors on other topics to see if we can generalize the results of this study across topics.

6. ACKNOWLEDGEMENTS

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

7. 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] B. Paaßen, T.W. Price, S. Gross, B. Hammer, T. Barnes, N. Pinkwart. 2018. The Continuous Hint Factory – Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *Journal of Educational Data Mining*, Volume 10, No 1.

[3] D. Shapira and J. A. Storer. 2007. Edit distance with move operations. *Journal of Discrete Algorithms*, 5, 2, 380–392.

[4] F.J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7, 3, 171-176.

[5] G. Cormode and S. Muthukrishnan. 2007. The String Edit Distance Matching Problem with Moves. ACM Trans. Algor. 3, 1, Article 2 (Feb. 2007), 19 pages. DOI= http://doi.acm.org/10.1145/1186810.1186812.

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

[7] S. Maharjan and A.N. Kumar. 2020. Using Edit Distance Trails to Analyze Path Solutions of Parsons Puzzles. *13th International Conference on Educational Data Mining (EDM) 2020* (forthcoming).

[8] Takabatake, Yoshimasa & Nakashima, Kenta & Kuboyama, Tetsuji & Tabei, Yasuo & Sakamoto, Hiroshi. 2016. siEDM: An Efficient String Index and Search Algorithm for Edit Distance with Moves. Algorithms. 9. 26. 10.3390/a9020026.

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