Using Markov Transition Matrix to Analyze Parsons Puzzle Solutions

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

ABSTRACT

In Parsons puzzles, students are asked to reassemble the scrambled lines of a program in their correct spatial order. The temporal order in which students reassemble the lines of code can provide insight into their puzzle-solving strategies. We applied Markov transition matrix to the puzzle solutions of introductory programming students to find patterns in their puzzle-solving strategies. We analyzed the data of students solving a Parsons puzzle involving selection statements in C++, Java and C#. In this paper, we will visualize the results of our analysis as heat maps. We found that most students assembled the program in the puzzle line by line in the order in which the lines appeared in the program. They discarded distracters either early or late in the puzzle-solving session and back-to-back more often than not. We also found differences between C++ and Java/C# solutions that support the results from prior research that program comprehension of novice procedural students was superior to that of novice object-oriented students.

Keywords

Parsons puzzles, Markov transition matrix, Programming Tutors.

1. INTRODUCTION

In a Parsons puzzle [1], the student is presented a problem statement, and the program written for it. The lines in the program are provided in scrambled order. The student is asked to re-assemble the lines in their correct order. Each puzzle may also contain distracters, which are syntactic or semantic variants of lines of code in the program. The student is asked to discard distracters.

The strategies used by students to solve Parsons puzzles have been of interest to researchers. One approach used to study puzzle-solving strategies has been to use state transition diagrams [5] – wherein, each node is one state in the puzzle: both the nodes and the arcs between them are sized proportional to the number of solutions that traversed them. A drawback of this approach is that the number of states in a Parsons puzzle is combinatorially explosive. So, the resulting graph is typically sparse, making it hard to find patterns. Another approach has been to use think-aloud protocol while students are in the process of solving the puzzles [6]. But, this approach does not scale well and cannot be used after students have solved the puzzles. Yet another approach has been to use Backus Naur Form grammars to represent ideal puzzle-solving strategies [7]. But, this approach can be used to check whether students used a desirable strategy to solve puzzles, not to find the strategies used by students.

We propose to use first order Markov transition matrix to find patterns in student solutions that correspond to their puzzle-solving strategies Our approach is tractable since it considers lines in the puzzle instead of states. It is scalable unlike think-aloud protocol. It can be used to find the strategies used by students unlike BNF grammars.

2. MARKOV TRANSITION MATRIX

In a typical Parsons puzzle tutor, a limited set of actions are provided for the student to solve a puzzle. The actions include inserting a line of code into solution, reordering a line in the solution, and deleting distracters. The data logged by the tutor includes a sequence of <line, action> tuples, wherein, the line refers to the correct line number of the line in the code, and action refers to the action applied to that line of code. We will refer to this as the student's **action sequence**.

Each Parsons puzzle has only one correct solution. So, the correct solution, i.e., the final re-assembled program will be the same for all the students. But, the order in which students go about assembling the lines of code, i.e., the action sequence of <line, action> tuples will vary among students. This order is a manifestation of their puzzle-solving strategy, influenced by their understanding of the syntactic and semantic relationships among the lines of code, for example, that a declaration statement is executed before an assignment statement or that a prompt statement must appear before input statement.

We represent each student's action sequence as a first order Markov transition matrix. In the matrix, the rows and columns are the lines in the program in their correct order. In addition, a first row is added for the start state S before attempting the puzzle and a last column for the end state E after completely solving the puzzle. We will use M as the abbreviation for Markov transition matrix and $M_{i,j}$ to denote the element of the matrix on row i and column j. Initially, all the elements $M_{i,j} = 0$. If a student applies an action to line j after applying an action to line i, we increment $M_{i,j}$ by 1.

	1	**2**	**3**	**4**	**E**
S				1	
1		1		1	
2	1				
3					
4	1				1

	1	**2**	**3**	**4**	**E**
S	1				
1			1		1
2		1	1	1	
3		2			
4	1				

Figure 1. Markov Transition Matrices for solution sequences 4-1-2-1-4 and 1-3-2-2-3-2-4-1 for a puzzle containing 4 lines of code.

Consider a puzzle containing 4 lines of code that are provided to the student scrambled. The left side of Figure 1 shows the Markov transition matrix of a student who applies actions to lines in the following order: 4-1-2-1-4. Since the first line acted on by the student is 4, $M_{S,4} = 1$. Thereafter, the matrix entries that are set to 1 are $M_{4,1}$, $M_{1,2}$, $M_{2,1}$, $M_{1,4}$ and finally, $M_{4,E}$ since 4 was the last line to be acted upon. The right side of the figure shows the matrix for a student who applies actions in the following order: 1-3-2-2-3-2-4-1. In particular, note that the student acts upon line 2 after line 3 twice – hence, $M_{3,2} = 2$. The student applies back-to-back actions to line 2, e.g., inserts line 2 in the solution, and immediately reorders it in the solution – hence, $M_{2,2} = 1$. The last line acted upon was line 1 – hence, $M_{1,E} = 1$.

We used Markov transition matrix to find patterns in the puzzle-solving strategy of a cohort of students. Each student may have solved a puzzle one or more times, i.e., the number of solutions ≥ number of students. We combined all the solutions of all the students into a single transition matrix, such that:

$$M_{i,j} = \sum a_{i,j} / \sum s$$

$\sum a_{i,j}$ is the sum of all the actions on line j after line i in all the student solutions;

$\sum s$ is the number of student solutions, i.e., the number of times students solved the puzzle.

So, $M_{i,j}$ is the number of actions on line j after line i per student solution. If all the students apply exactly one action to each line in each solution, $0 \leq M_{i,j} \leq 1$. But, since students are allowed to act upon each line as often as they wish, $M_{i,j}$ can be greater than 1.

Since the puzzles also included two distracters D1 and D2, we added rows and columns in the matrix for D1 and D2 after those for all the lines in the puzzle. $M_{i,D1}$ refers to students acting on the first distracter D1 after line i. In the matrix:

- If each student applies exactly one action to each line of code, the sum of all the entries in a row / column is 1. But, since a student may apply more than one action to a line of code (e.g., insert into the solution, reorder within the solution), the sum of each row / column is at least 1.
- The larger the value of $M_{i,j}$, the larger the number of times students applied an action to line j after line i. So, the larger the number of times students discerned a syntactic or semantic relationship between lines i and j.
- A puzzle that is temporally assembled in the correct spatial order of lines in the code will appear as entries in all the elements $M_{i,i+1}$.
- If the students randomly solve a puzzle, almost all the entries in the matrix of the puzzle will be non-zero.

3. ANALYZING PARSONS PUZZLE SOLUTIONS

For this study, we analyzed the data collected by a Parsons puzzle tutor called epplets (epplets.org) [2] on `if-else` statements. The tutor was used by introductory programming students as an after-class assignment. The tutor was used by C++, Java and C# students during fall 2016 – fall 2020.

In particular, we analyzed student solutions of a puzzle wherein, the program was written to read two numbers and print the smaller value among them. The puzzle contained 14 lines of code and 2 distracters in C++ and Java. In C#, the puzzle contained 15 lines of code and 2 distracters. The pseudocode of the program was as shown in Figure 2, line for line:

```
1    Declare variable for first number
2    Declare variable for second number
3    Prompt for first number
4    Read in first number
5    Prompt for second number
6    Read in second number
7    if( first number < second number )
8    {
9       Print first number
10   }
11   else
12   {
13      Print second number
14   }
```

Figure 2. Pseudocode of the puzzle

In C#, there was an extra line 15, which ended the function main. For analysis purposes, the two distracters were counted as lines 16 and 17, although they were presented to the student paired with the original line of code of which they were a variant. Pseudocode was included as comments in the puzzle before lines 1,2,3,5 and 7, which disambiguated the relative order of lines 1 and 2, and lines 3-4 and 5-6. Students got credit whether they placed an open brace on line 8 or line 12. Similarly for close brace on lines 10 and 14.

For our analysis, we considered only those students who solved the puzzle completely and correctly so that we could find patterns among those who successfully solved the puzzle. Some students may have solved the puzzle more than once. We considered all those solutions. A puzzle with n lines can be solved with n actions. A student who solved a puzzle with no more than 10% extra actions is considered to have solved the puzzle optimally. We also analyzed optimal solutions separately.

4. RESULTS

We present the Markov transition matrix as a heat map, with darker green for larger values. For simplicity, we present the values in each matrix element multiplied by 100 and as whole rounded numbers, e.g., 0.016 as 2.

Figure 3 presents the heat map of complete C++ solutions (N=98). In the figure, S stands for Start State and E for End State. D1 and D2 are distracters, listed after the 14 lines of code.

We observe the following with regard to the puzzle-solving strategies of students:

1. Most students started by assembling the two variable declaration statements. They assembled the two statements back to back.
2. Most students assembled the program in the puzzle line by line in the order in which the lines appeared in the program. So, the largest values are all along the diagonal. For example, $M_{3,4}$ of students who acted upon input statement

after prompt statement is far greater than $M_{4,3}$ of students who acted upon prompt statement after input statement. Similarly, $M_{5,6}$ is far greater than $M_{6,5}$.

3. Most students tried to discard distracters either early in the puzzle-solving session or late (columns D1 and D2). They also acted upon distracters back-to-back more often than not.

4. Even though shell or frame-first coding [3] is encouraged, i.e., students are advised to write `if()` followed by `else`, and close brace after the corresponding open brace, students did not seem to follow this advice. Hardly anyone assembled `else` (line 11) after `if` (line 7), i.e., $M_{7,11}$ is very small. Similarly, $M_{8,10}$ of students acting upon closing brace after open brace is smaller than $M_{8,9}$ of students acting upon the content of if-clause after open brace of if-clause. Similarly for else-clause, i.e., $M_{12,14}$ is smaller than $M_{12,13}$.

Figure 4 shows the heat map of complete solutions in Java (N=146). Most of the patterns observed for complete C++ solutions can also be observed for complete Java solutions. Figure 5 shows the heat map of complete C# solutions (N=43). We see the trend that Java heat map is more dispersed than C++ heat map and C# heat map is even more dispersed than Java heat map, i.e., more off-diagonal elements have larger values in Java/C# than in C++. The column E (for End State) is reached in C++ by most students after the last three lines in the puzzle, viz., 12-14 or the two distracters. In Java, several students reached the end state after lines 5 and 6 deep within the program. In C#, students reached the end state from many more lines in the program than either in Java or C++. One explanation is that this may be due to the paradigm of programming used in the languages: object-oriented in Java/C# versus procedural in C++. Prior research found that program comprehension of novice procedural students was superior to that of novice object-oriented students, possibly because of longer learning curve for object-oriented programming [4].

Figure 6 presents the heat map of the differences between C++ (Figure 4) and Java (Figure 5) solutions. The difference can be calculated because C++ and Java programs have exactly the same code on each line. We find two noticeable differences:

1. Java students applied back-to-back actions to the same line more often than C++ students, e.g., to lines 1, 4 and 6. So, for example, difference $M_{1,1}$ is large.

2. Java students preferred to act upon the two input statements back-to-back and act upon the two prompt statements back-to-back unlike C++ students who chose to assemble each input statement immediately after its corresponding prompt statement. So, difference matrix $M_{3,5}$ and $M_{4,6}$ are large. One explanation is that the syntax of input and output statements is larger in Java compared to that in C++, e.g.,

   `firstNum = stdin.nextInt();` in Java compared to

   `cin >> firstNum;` in C++ and

   `System.out.println( "Enter the first value");` in Java versus

   `cout << "Enter the first value";` in C++.

   So, students are more likely to notice the two Java input statements as being similar, prompting them to act upon them back-to-back.

Figures 7 and 8 present the heat map of the optimal solutions in C++ (N=33) and Java (N=23). Note that optimal solutions are more tightly spun around the diagonal, i.e., students who solved the puzzles with the fewest unnecessary actions did so in backward reasoning fashion, i.e., starting from a visualization of the final program and assembling the lines of code in the order in which they appear in the program, and not in an opportunistic forward-reasoning fashion.

In summary, Markov transition matrix is a useful tool to analyze the strategies used by students when solving Parsons puzzles. When visualized as a heat map, it succinctly summarizes patterns in their puzzle-solving behavior and highlights the differences between groups such as C++ versus Java students, and complete versus optimal solutions.

5. DISCUSSION

In our analysis, we considered only line numbers and not actions in action sequence, the sequence of <line, action> tuples. So, matrix element $M_{i,j}$ was a number and not the action taken on line j after line i. This coding lost some data available in action sequences. For example, $M_{i,i}$ represents back-to-back actions applied to line i. These could be actions that cancel each other out, such as deleting a line followed by undeleting it. In such a case, the two actions could be ignored. Similarly, two actions applied back-to-back to a line could signal issues with the user interface, e.g., when a line is inserted into solution and immediately moved up or down in the solution by just one line: when the actions are drag-and-drop as in the case of epplets, it may not have been clear to the student where to drop a line so that it is inserted in its intended location. Including the nature of action in the Markov transition matrix may lead to richer results.

In the current analysis, we considered only complete and correct solutions as well as optimal solutions. Analyzing incomplete and incorrect solutions may yield patterns in puzzle-solving behavior that unearth common misconceptions among programming students.

This search for patterns can be extended to more than back-to-back operations: element $M_{i,j}$ in nth order Markov transition matrix will yield a measure of students acting upon line j in the nth action after line i. This could be used to answer questions such as how quickly after assembling an open brace do students get around to assembling its matching closing brace in the program.

We have accumulated log data from multiple epplets – on sequence, selection and loops, and on multiple puzzles, including those involving nested control statements. In the future, we plan to apply Markov transition matrices to analyze this log data.

6. ACKNOWLEDGMENTS

Partial support for this work was provided by the National Science Foundation under grants DUE-1432190 and DUE-1502564.

7. REFERENCES

[1] Parsons, D and Haden, P.: Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proc. 8th Australasian Conference on Computing Education (ACE '06), Vol. 52. pp 157-163. Australian Computer Society, Inc. (2006)

[2] Kumar, A.N.: Epplets: A Tool for Solving Parsons Puzzles. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18), pp. 527-532. ACM, New York, NY, USA (2018)

[3] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15). ACM, New York, NY, USA, 29-38. DOI: https://doi.org/10.1145/2818314.2818331

[4] Susan Wiedenbeck, Vennila Ramalingam, Suseela Sarasamma, Cynthia L. Corritore. A Comparison of the Comprehension of Object-oriented and Procedural Programs by Novice Programmers. Interacting with Computers, 11 (3). January 1999, Pages 255–282, https://doi.org/10.1016/S0953-5438(98)00029-0

[5] 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.

[6] Fabic, G., Mitrovic, A., Neshatian, K.: Towards a Mobile Python Tutor: Understanding Differences in Strategies used by Novices and Experts. In: Proceedings of the 13th International Conference on Intelligent Tutoring Systems, LNCS, vol. 9684, pp. 447–448. Springer Heidelberg (2016)

[7] Amruth N. Kumar. 2019. Representing and Evaluating Strategies for Solving Parsons Puzzles. In Proceedings of Intelligent Tutoring Systems (ITS 2019), Kingston, Jamaica. Springer LNCS 11528, 193-203

C++	1	2	3	4	5	6	7	8	9	10	11	12	13	14	D1	D2	E
S	70	3	4	3	2	0	0	0	0	1	2	0	2	0	9	3	0
1	28	96	5	2	1	2	3	0	2	1	0	0	0	0	7	0	0
2	10	15	57	16	2	1	4	1	3	2	2	0	0	0	10	7	1
3	3	1	32	63	26	5	2	2	4	3	2	1	1	1	18	4	0
4	8	2	12	22	65	17	0	0	6	9	0	1	6	1	14	5	1
5	1	1	7	8	22	89	8	2	1	2	0	0	3	2	9	6	3
6	4	2	6	7	14	14	75	3	5	1	0	1	4	2	4	6	0
7	0	0	1	2	3	1	15	68	22	3	2	2	1	0	4	7	1
8	1	0	3	0	0	1	4	15	52	27	4	11	1	7	3	3	2
9	3	1	4	6	3	0	1	9	29	51	26	4	11	2	3	5	3
10	2	1	3	6	0	1	1	10	8	32	61	16	3	14	4	6	3
11	0	1	0	1	1	0	2	4	7	3	21	59	33	4	2	4	1
12	1	2	3	1	2	0	2	2	0	12	6	18	48	32	2	5	10
13	0	0	1	4	8	5	2	6	4	7	4	10	15	57	6	12	8
14	2	0	8	2	0	0	1	4	7	13	4	9	13	6	25	24	22
D1	9	5	15	21	12	7	7	4	5	1	2	6	3	3	12	24	25
D2	4	2	6	5	3	5	6	3	5	4	7	7	6	9	29	5	19

Figure 3. Heat Map of Complete C++ Solutions (N=98): S is Start state, E is End state, D1 and D2 are distracters

Java	1	2	3	4	5	6	7	8	9	10	11	12	13	14	D1	D2	E
S	57	7	3	13	3	0	4	0	0	1	1	1	0	2	6	2	0
1	78	123	8	19	3	8	3	1	1	1	0	1	1	1	9	2	1
2	32	44	50	39	4	19	8	1	5	0	1	1	1	1	9	9	1
3	8	6	45	67	61	8	2	3	6	3	1	0	1	0	17	11	1
4	17	16	19	60	53	82	7	1	6	2	0	0	4	1	16	7	1
5	6	2	23	14	41	60	13	3	8	3	2	3	7	3	15	19	10
6	9	12	18	25	14	53	69	3	8	1	1	2	1	4	14	14	10
7	1	1	8	4	1	3	30	77	38	3	2	8	5	0	3	4	2
8	2	1	7	3	4	3	14	34	58	26	6	13	5	3	2	7	3

9	3	2	4	6	3	3	4	15	27	64	34	5	14	9	3	8	1
10	2	0	4	5	2	3	1	8	12	27	72	16	8	12	3	7	3
11	3	0	3	1	3	0	3	8	6	9	16	65	36	6	2	2	3
12	5	1	3	3	3	1	1	10	6	8	8	20	62	24	5	8	8
13	3	3	8	3	8	2	3	8	10	14	6	12	26	71	4	10	3
14	4	1	12	1	5	2	3	3	4	16	6	14	12	13	23	18	19
D1	17	8	17	18	10	8	14	7	4	3	4	5	2	2	46	30	11
D2	12	1	10	10	13	6	10	7	4	6	6	10	9	4	28	20	23

Figure 4. Heat Map of Complete Java Solutions (N=146): S is Start state, E is End state, D1 and D2 are distracters

C#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	D1	D2	E
S	51	2	0	5	0	2	2	2	0	0	0	0	0	2	21	9	2	0
1	19	77	0	7	2	2	2	0	0	0	0	0	0	0	0	21	0	0
2	9	12	28	21	2	7	7	2	16	0	0	0	0	0	7	7	5	0
3	0	0	26	37	54	12	0	2	21	2	12	5	5	5	7	9	7	0
4	7	2	16	49	23	63	12	2	19	5	5	0	14	0	0	12	7	5
5	2	0	33	19	33	40	9	0	9	9	0	7	16	2	2	2	14	2
6	2	0	7	23	12	19	35	5	23	5	7	5	7	5	2	12	19	9
7	5	0	12	7	5	2	19	51	19	5	5	0	5	0	2	2	2	0
8	0	0	5	7	0	7	2	33	42	26	12	12	7	5	2	5	5	0
9	2	0	14	16	14	12	9	14	30	30	23	7	28	5	5	0	2	9
10	2	2	0	0	2	0	2	14	7	30	42	12	12	19	9	2	9	14
11	0	2	9	7	7	2	2	12	5	2	28	49	23	5	7	2	9	2
12	2	0	5	2	14	2	0	7	5	19	9	30	33	21	2	7	7	9
13	0	0	2	9	14	7	12	9	5	9	9	19	26	47	14	5	9	9
14	0	0	9	2	2	2	5	2	2	16	12	16	12	33	23	5	14	12
15	16	0	9	5	2	2	7	2	5	9	5	2	7	19	9	7	2	12
D1	7	26	14	9	7	12	2	2	7	9	2	5	2	2	2	14	12	5
D2	5	0	14	14	7	2	12	7	7	2	5	7	9	0	5	19	5	12

Figure 5. Heat Map of Complete C# Solutions (N=43): S is Start state, E is End state, D1 and D2 are distracters

Difference	1	2	3	4	5	6	7	8	9	10	11	12	13	14	D1	D2	E
S	14	4	1	10	1	0	4	0	0	0	1	1	2	2	3	1	0
1	51	27	3	17	2	6	0	1	1	0	0	1	1	1	2	2	1
2	22	29	7	23	2	18	4	0	2	2	1	1	1	1	1	2	0
3	5	5	14	4	36	2	0	1	2	0	1	1	0	1	1	7	1
4	9	14	7	38	12	64	7	1	1	7	0	1	2	0	2	2	0
5	5	1	16	6	19	29	5	1	7	1	2	3	4	1	6	12	7
6	5	10	12	18	0	39	6	0	3	0	1	1	3	2	10	8	10
7	1	1	7	2	2	2	15	9	16	0	0	6	4	0	1	3	1
8	1	1	4	3	4	2	10	19	6	1	1	2	4	4	1	4	1
9	0	1	0	1	0	3	3	6	1	13	8	1	3	7	0	2	2
10	0	1	1	1	2	2	0	2	4	4	11	1	4	2	1	1	0

11	3	1	3	0	2	0	1	3	2	6	6	6	3	1	0	2	2
12	4	1	0	2	1	1	1	8	6	4	2	2	14	8	3	2	3
13	3	3	7	1	1	3	1	1	6	7	1	2	11	14	2	2	5
14	2	1	3	1	5	2	2	1	3	3	2	5	1	7	2	6	4
D1	8	2	2	4	3	0	7	3	1	2	2	1	1	1	34	7	14
D2	8	1	4	5	10	0	4	4	1	2	2	3	3	5	0	15	3

Figure 6. Heat Map of Difference Between Complete C++ and Java Solutions

C++	1	2	3	4	5	6	7	8	9	10	11	12	13	14	D1	D2	E
S	88	0	0	0	0	0	0	0	0	0	0	0	0	0	9	3	0
1	0	97	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0
2	0	0	76	6	0	0	0	0	0	3	0	0	0	0	9	6	0
3	0	0	0	82	12	3	0	0	0	0	0	0	0	0	3	0	0
4	0	0	6	0	79	6	0	0	0	0	0	0	0	0	3	6	0
5	0	0	0	0	6	85	6	0	0	0	0	0	0	0	3	6	0
6	0	0	0	0	3	0	88	0	0	3	0	0	0	0	3	3	0
7	0	0	0	0	0	0	0	82	12	3	0	0	0	0	0	3	0
8	0	0	0	0	0	0	0	0	70	15	0	6	0	6	3	0	0
9	0	0	0	0	0	0	0	9	0	70	12	3	3	0	3	0	3
10	0	0	0	0	0	0	0	0	9	0	85	3	0	3	0	3	0
11	0	0	0	0	0	0	0	3	3	0	0	73	21	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	3	3	70	9	0	6	12
13	0	0	0	0	0	0	0	0	6	3	0	0	0	82	6	0	3
14	0	0	3	0	0	0	0	3	3	6	0	6	6	0	24	36	12
D1	12	0	12	9	3	0	3	0	0	0	0	3	0	0	3	27	30
D2	0	3	3	3	3	6	3	3	0	0	0	6	0	0	30	0	39

Figure 7. Heat Map of Optimal C++ Solutions (N=33): S is Start state, E is End state, D1 and D2 are distracters

Java	1	2	3	4	5	6	7	8	9	10	11	12	13	14	D1	D2	E
S	70	9	0	0	0	0	0	0	0	0	0	4	0	0	9	9	0
1	0	91	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	9	0	74	9	4	0	0	0	0	0	0	0	0	0	4	0	0
3	0	0	0	78	13	0	0	0	0	0	0	0	0	0	9	0	0
4	0	0	4	0	74	17	0	0	0	0	0	0	0	0	4	0	0
5	0	0	4	0	0	74	4	0	0	0	0	0	0	0	9	9	0
6	0	0	0	0	4	0	78	0	0	0	0	0	0	0	13	4	0
7	0	0	0	0	0	0	0	83	17	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	70	17	0	4	0	0	4	4	0
9	0	0	0	0	0	0	0	4	0	65	26	0	0	0	0	0	4
10	0	0	0	0	0	0	0	4	9	0	74	4	0	4	0	4	9
11	0	0	0	0	0	0	0	0	0	4	0	65	30	0	0	0	0
12	4	0	0	0	0	0	0	0	0	4	0	4	61	22	0	9	0
13	0	0	0	0	0	0	0	4	4	4	0	9	0	70	4	4	0

14	0	0	4	0	0	0	0	0	0	13	0	9	9	0	30	17	17
D1	9	0	4	9	4	9	13	0	0	0	0	4	0	0	0	39	9
D2	9	0	0	4	0	0	4	4	0	0	0	0	0	4	13	0	61

Figure 8. Heat Map of Optimal Java Solutions (N=23): S is Start state, E is End state, D1 and D2 are distracters