FSU COMPUTER SCIENCE DEPARTMENT

Puzzle Technical Report

COSC 310 Section 001

Data Structures and Analysis

April 21, 2013

By William Briggs

1832235

This project will display the two searching algorithms Depth First Search and Breadth First Search traversals. When solving a puzzle by traversing a tree of possible puzzle solutions, the program will put the solution through a series of test conditions to meet all of the puzzle requirements. Each of the searching algorithms will produce a different solution, which disproves that there is only one solution to the puzzle. This technical report will demonstrate how the code was implemented and how different solutions can be obtained through different search methods.

**CONTENTS**

LIST OF PICTURES/TABLES 3

1. INTRODUCTION 4
2. BACKGROUND 4
   1. Breadth First Search 4
   2. Depth First Search 4
3. APPROACH 5

3.1 Puzzle Piece Class 5

3.2 Board Class 5

3.3 Puzzle Class 6

1. RESULTS 6
2. CONCLUSION 7
3. REFERENCES 7
4. APPENDICES 8

**LIST OF PICTURES/TABLES**

**Figure 1. Solution to the puzzle using Breadth First Search**

**Figure 2. Solution to the puzzle using depth First Search**

**Figure 3. Breadth First Search**

**Figure 4. Depth First Search**

1. **INTRODUCTION**

This puzzle project is an attempt to implement different search algorithms in order to find the most efficient way to solve the given puzzle. According to puzzle description, the box claims this 3x3 puzzle offers over 300,000 wrong ways to complete and only one possible correct solution (not including rotations). The conditions to finding a correct solution are using all of the pieces, having all sides compatible with one another, and have 6 tabs facing out along the edges of the puzzle. Each piece consists of four sides with their individual tabs that can be either a tab out or slot in. Out of these two choices, there are four choices of the pattern they can be. This program will output how the algorithm found its solution through a series of printouts that lets the user know what is happening. The code will demonstrate the following:

1. Develop a data structure to store the state of the puzzle. This will be your state description.
2. Develop/use some list structure in which you will be able to add the states of the puzzle.
3. Develop a node data structure. In the node you will store the current state of the puzzle, a pointer (or some reference) to the parent node, the move that led to the current state, and the depth.
4. Develop a set of rules for generating children (successor states).
5. Implement the Breadth-first algorithm to solve the puzzle and the depth-first algorithm to solve the puzzle.
6. Your code will have two modes for each algorithm: normal and verbose. In normal mode, the program will output only the final solution steps. In verbose mode, the program will output all the states it is generating in the order in which they were searched, in addition to the final solution steps. These modes are best implemented through command line arguments.
7. **BACKGROUND**

In order to begin this project, I had to expand my knowledge about the two search algorithms that I would be implementing. Breadth First Search and depth First Search are two different algorithms that either use the queue or stack in their searches [1].

* 1. **Breadth First Search**

Breadth First Search is an uninformed search method that will search through every possible solution by expanding and evaluating all nodes in a graph. It will search for the correct answer without considering what the answer might be; this tends to be an exhausting search method for the system. This algorithm will use the queue to find and compare all children nodes that the puzzle produces. The search method will search all of the children before continuing onto the grandchildren. The worst case scenario the graph will have is a depth of one, meaning all vertices must be stored [2]. This will make Breadth First Search impractical for large problems that require large amounts of memory.

* 1. **Depth First Search**

Depth First Search is more closely related to the preorder traversal of a tree. The overall algorithm initializes a set of markers to verify which vertices are visited, chooses a starting vertex x, initializes the tree, and calls the next depth First Search [3]. Closely related to Breadth First Search, this algorithm can be thought of as a shorter and more complex search. The search method traversal will start at the root node, and will then explore all of the available children nodes until there are none left. After searching the children nodes the algorithm will backtrack to a previous child. This method will know when to backtrack when there is no more children to traverse, the code will check if the child is a solution. If not, it will return to the nearest child and continue its search.

1. **APPROACH**

**3.1 PUZZLE PIECE CLASS**

In order to approach this puzzle problem, there needs to be a way to represent all of the possible sides as well as rotations/flips of the pieces. Considering these factors I implemented a PuzzlePiece class that has 4 parameters upon creation. The piece will set the first parameter as the top, second as the right, third as the bottom, and the fourth as the left. I used enums to represent the possible sides (HeartIN, HeartOUT, etc) so comparisons can be made between the sides.

The piece node should also have methods that will return the possible combinations the piece can have; this represents the possibility of turning/flipping all of the pieces. In order to accomplish this I had to create a method for each rotation that is possible. The method turnPieceRight will take an integer value as a parameter and rotate the piece right. After making a method for rotating the piece one time, I made a second method that allowed for multiple rotations using an integer as a parameter. In addition to the rotation methods I had to create a flip method that would flip the piece. I have 2 flip methods that will flip horizontally and flip upwards; all of these methods are what I need to produce the possible orientations of each puzzle piece.

The next step was to create a method that will generate a list to hold all of my orientations to the puzzle piece. To do this, I created a PuzzlePiece array that will hold 8 pieces. To create an array of all the same pieces, I needed to create a copy method. The copy method will return a new puzzle piece of the same parameters, in other words copying it. With this method I was able to copy the original piece 8 times and fill the array. Using this array I can use my flip and rotate methods to generate the possible orientations of the piece. After my array is filled with the different orientations of the puzzle piece I can traverse this array when trying to produce a solution.

With a full array of possible orientations of the puzzle piece, I needed a way to compare the pieces’ sides and determine whether they are compatible. Using four different Boolean fit methods, I was able to return true if the sides were compatible. With a puzzle piece as a parameter, I was able to have a switch/case condition for each possible side that would return true if the sides matched. This is important to the project because this method controls if the pieces can be linked for a possible solution.

**3.2 Board Class**

The board class needs to hold all values of the solution in a 3x3 board, and check if the conditions are met to be considered a solution. To solve the 3x3 board problem, I used an array to hold 9 puzzle pieces and set up pointers to create a 3x3 board. Each of the puzzle pieces will have to use the fit methods to meet the requirements of being added to the array. This will ensure that when the board is full, the result will have to be a solution.

In order for the progress of board completion, the board will have to test the valid continuations of the board progress. There needs to be a method to link pieces together and add them to the board. This will allow for further traversing to get closer to the solution, otherwise it would just be comparing two pieces the entire time. A method is needed to return an array with possible solutions; this will test a puzzle piece with all orientations of another piece. This will also output what orientations are being tested to the user, for when the program runs in verbose. Each orientation tested will return if the pieces are valid together, and then return the valid possibilities. To test if each piece is valid the board class will require its own Boolean valid method. In this method the parameter (puzzle piece) will have a switch/case that returns valid possibilities.

The board’s toString method will return the 3x3 board that the pieces are represented by. It will return the positions of letters A-I so when the puzzle prints out the solution, the user will know the position of each piece. This will be useful to the user when the solution is displayed.

**3.3 Puzzle Class**

The puzzle class runs my main method that will input the puzzle pieces and then search for the solution using both Breadth First Search and depth First Search. The project will take arguments to know what search algorithm to use. Depth First Search will be executed if the argument is “dfs”, Breadth First Search will be executed if the argument is “bfs”. Each of these will also search for a second argument of “v”, which will tell the program to run in verbose mode. Verbose mode will print out to the user how the algorithm is searching for a solution. If the program does not read a second argument, then verbose will be set to false. Before an algorithm can be started the puzzle pieces will be generated into an array, each being created with the proper sides of the puzzle. At this point is when the program will call one of the two searching algorithms to find a solution.

1. **RESULTS**

Each search algorithm produced different solutions that met of the requirements when executed. The difference between the two algorithms was the execution time it took to produce the solution. Breadth First Search with verbose had an average of 20 minutes before a solution could be obtained. This is because the search had to test every possible attempt, which takes time. Depth First Search found a solution in about a minute, because it traversed all possible attempts before it tested them. In this case, the Breadth First Search took more time because of size of the project; it displays all of the comparisons that the algorithm conducts. Both of these search algorithms did produce different results, and they both are correct solutions to the puzzle. This disproves the theory that there are 300,000 incorrect ways and only 1 correct solution for this puzzle.

Figure. 1

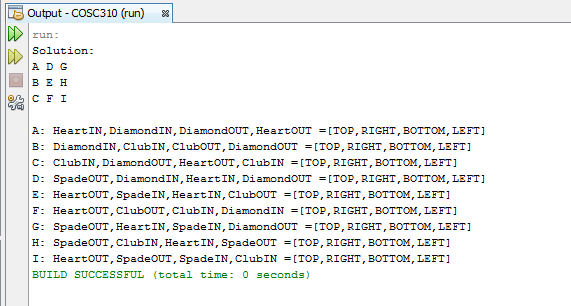
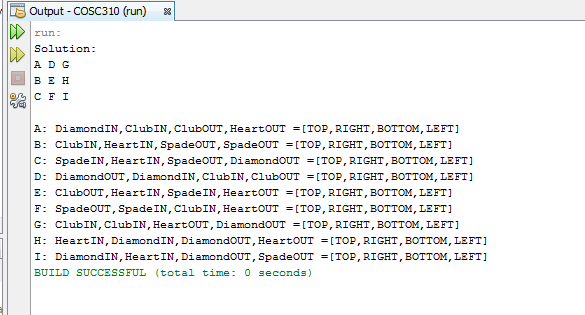


Figure. 2



1. **CONCLUSION**

Both search algorithms will return solutions where all pieces fit together, but they did not get checked to see if they had 6 tabs. The final method before returning the solution is to check how many tabs and slots are along the edges of the puzzle. In order to do this, I had to check the appropriate outer pieces’ sides in the solution. Reading each side for the middle top, left, right, and bottom gave me four of the necessary sides. The pieces located in the corners had two sides to check, giving a total of twelve sides. My Boolean method increments the counter for each tab that is counted, and if there is six tabs then it returns true. In theory if there are six tabs there should be six open slots as well, satisfying that condition.

The puzzle project perfectly demonstrates the difference between depth First Search and Breadth First Search algorithms. Both algorithms use the stack or queue to solve the solution to the puzzle. Surprisingly they produce two different solutions which represents that different searches can produce different solutions. I would like to use Breadth first and depth First Search in future cases, so I can determine what algorithm is more efficient in larger databases compared to smaller ones.

1. **REFERENCES**

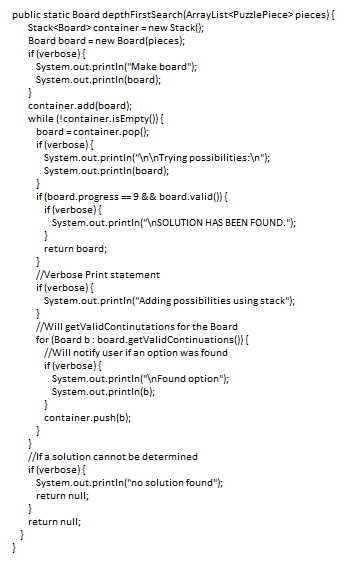
[1] Kun, Jeremy (2013, January 22). Depth- and Breadth-First Search [online]. Available: [http://jeremykun.com/2013/01/22/depth-and-Breadth-first-search/](http://jeremykun.com/2013/01/22/depth-and-breadth-first-search/)

[2] (1996, February 15) Breadth First Search and depth First Search [online]. Available: <http://www.ics.uci.edu/~eppstein/161/960215.html>

[3] Beamer, Scott. Patterson, David. Asanovic, Kryste. (2002) Direction-Optimizing Breadth- First Search [online]. Available:

<http://conferences.computer.org/sc/2012/papers/1000a019.pdf>

1. **APPENDICES**

Figure 3. Figure 4.

