**Homework 3 Report** **Mehmet Çağrı Aksoy**
EEM 480 12105037560
Assoc. Prof. Emin Germen

## Problem Statement

In the figure given below there is a definitive path from 1 to 32. Path should be found using an algorithm. One point to N, S, E, W, NE, NW, SE, SW should be shown. The program has to ask the path of the file of the question The text file will only contain the numbers and space between the numbers.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 12 | 20 | 14 | 15 | 16 | 17 | 15 | 14 |
| 21 | 13 | 19 | 18 | 17 | 16 | 13 | 20 |
| 2 | 22 | 1 | 4 | 5 | 12 | 18 | 21 |
| 23 | 1 | 3 | 2 | 11 | 6 | 19 | 22 |
| 24 | 24 | 4 | 3 | 25 | 10 | 32 | 23 |
| 25 | 27 | 28 | 5 | 7 | 8 | 9 | 31 |
| 9 | 26 | 7 | 29 | 6 | 28 | 32 | 30 |
| 10 | 11 | 8 | 26 | 30 | 31 | 27 | 29 |

Given table

## Introductions

To solve this problem, Netbeans IDE 8.2 is used, in Ubuntu 18.04 x64. OverLeaf Online Latex editor is used to design and write this report.

## Solution

If we come to the algorithm side, we need a few important points to solve the problem. These are the structure needed to find the solution, the problems that may arise if there is a dead end, and the integration of the backtracking algorithm into our scenario. The algorithm first requests a file path from the user, and the file is scanned from the received file path with the "Scanner" function and the reading process is started. The data read in parallel is printed on the previously defined "Maze" array.

```
maze[x][y]=readCodes.nextByte();
```

The data received from the user and successfully added to the array is stored for later use. These functions also work in a "try-catch" algorithm. If the file path received from the user is not correct, the program prompts the file path again. In the second step, the "SolveMaze ()" class runs. In this step, first, definitions are made through the Position class. Since the program does not know which direction to go in the "Maze", the directions are pre-defined for convenience. As can be seen on the bottom code block, east and south is defined. offset[0] represents East side of maze.

```
 // EAST
offset[0].row = 0;
offset[0].col = 1;
   // SOUTH
offset[1].row = 1;
offset[1].col = 0;
```

**Important point:**

The unique aspect of my algorithm was that it started from "32" instead of starting from "1". Thinking a little tricky here, it is often proved that the beginning of maze problems is difficult and complicated, while the last part is easier and less complex. As a result, the written algorithm selects it as the starting point when it finds the 32nd number. Afterwards, it decreases the number that it is looking for and then comes up to "1". In this block of code, defined below, one of the program searches for all aspects. If one is equal to "temp", the program breaks the loop, and exits the loop.

```
temp = 31;
while (option <= LastOption) {
                              r = here.row + offset[option].row;
                              c = here.col + offset[option].col;
                              if (maze[r][c] == temp) break;
                              option++; // next option
                              }
```

**Exception:** If more than one starting point is defined, and one is reaching the result and the other is a dead end, the program gets rid of this by the "try-catch" algorithm and the "catch" block finds another start points.

The position information is thrown to the stack as you proceed step by step. If the result is reached, the loop is exited and the "PrintSolution ()" class is executed to print the path found. However, as mentioned before, if one of the roads is a dead end, the pop is going to go backwards by "pop". Reminder -¿ Stack: Last In First Out (LIFO): The last object into a stack is the first object to leave the stack, used by a stack Thanks to the "Pop" function, the data in the stack is reduced and represents a step back in this maze.

```
Note: "temp" represents a number that required to find which between 1 to 32.
// was a neighbor found?
                           if (option <= LastOption) {
                              // move to maze[r][c]
                              here = new Position(r,c);
                              path.push(here); //Add "here" Position into the Stack
                              here.row = r;
                              here.col = c;
                              maze[r][c] = 0; // set to 0 to prevent revisit
                              option = 0;
                              temp--;
                              }
                           else {// no neighbor to move to, goes back !
                              if (path.IsEmpty()) return;
                              next = path.pop();
                              if (next.row == here.row)
                                 option = 2 + next.col - here.col;
                              else option = 3 + next.row - here.row;
                              here = next;
                              ++temp;
                             }
```

As have seen above, the mechanism of finding and returning the path of maze algorithm works in this block of code.

Finally, as started to look for the maze solution from the 32Nd point, so the positions are saved in the stack were actually signed upside down. But this situation has provided us a very important benefit. As known, the stacks are in the Last In First Out (LIFO) property structures, So when we want to print our maze's path, stack shows last items always.

Thus, by going in reverse, the 32 to 1 stack with "pop" function deletes the top data and prints the solution from 1 to 32.

## Implementation

As much as creating an algorithm, transferring the algorithm to the code is at least as important as the algorithm itself. To integrate the written algorithm into the code block, java properties such as stack, list, and array is used. Code block consists of 6 different files. One is "interface" and the others are classes whose is named "HW3, Maze, MyLinkedList, MyStack, Position". The most important point in designing the algorithm is complexity. Two complexity types exist, time and space. When applying this algorithm to the Java program, the purpose of keeping time complexities low was considered. All Stack operations have O (1) complexity, but the Array data search operations require O (n) complexity. A while or for loop corresponds to approximately O (n) complexity. Because the arrays used in this program were two-dimensional, O ($n^2$) is required complexity for the search in the array.

## References


1. https://introcs.cs.princeton.edu/java/43stack/
2. https://medium.com/@andreaiacono/backtracking-explained-7450d6ef9e1a
3. http://eem.eskisehir.edu.tr/Ders.aspx?dersId=82