

# COLLEGE OF ENGINEERING TRIVANDRUM



## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

### CST 306 ALGORITHM ANALYSIS AND DESIGN Maze Generation and Solving Assignment - 2

---

Demel C John  
CSE 24 Batch 2  
Roll no 22  
TVE20CS042

# Maze Generation and Solving

## 1 Task

### Maze Generation and Solving

Design and implement an algorithm to generate a random maze and then solve it. The maze will be represented as a grid of cells, where each cell can either be a wall or a passage. The goal of the algorithm is to create a maze with a single entrance and a single exit, ensuring that there is a path between any two cells in the maze. Additionally, the algorithm should be able to find a solution from the entrance to the exit.

Design and implement an algorithm to generate a random maze using the following requirements:

(a) Maze Generation:

- i. The maze should have a user-defined size (number of rows and columns).
- ii. The maze should have a clear entrance and exit point, located at the top-left and bottom-right corners respectively.
- iii. There should be a path connecting the entrance and exit points, and every cell in the maze should be reachable from the entrance.
- iv. The maze should have a reasonable level of complexity, with dead-ends and twists in the paths.
- v. The algorithm should be efficient, avoiding excessive backtracking or unnecessary computations.

(b) Maze Solving:

- i. Implement an algorithm to solve the generated maze and find a path from the entrance to the exit.
- ii. You can choose a pathfinding algorithm, such as Breadth-First Search (BFS) or Depth-First Search (DFS), to find the solution.
- iii. The solution path should be stored and presented as a sequence of cells or coordinates.

Write a detailed algorithm description that outlines the steps involved in both maze generation and maze solving. Include the data structures, such as arrays or graphs, that will be used to represent the maze and track the progress of the algorithms. Analyze the time and space complexity of your algorithms, and discuss any trade-offs or optimizations that can be made. In addition to the algorithm description, provide a working implementation of the maze generation and solving algorithms in a programming language of your choice.

Test the algorithms by generating and solving mazes of various sizes, and visually represent them using appropriate symbols.

Submit your algorithm description, implementation code, and a report discussing the efficiency, performance, and any challenges encountered during the implementation process.

## 2 Maze Generation

A maze generation algorithm is a method for creating a maze. There are different methods to generate maze. These methods include :Recursive division,Randomized DFS,Cellular automata etc. Here , the method used is similar to randomized DFS. The method tries to find a way to the end point and then backtrack and makes all possible paths. This algorithm starts with a single cell that is marked as visited. A random neighboring cell that is not yet visited is then chosen and marked as visited. This process is repeated until all cells in the grid have been visited.

Stack: A stack is a data structure that stores data in Last In First Out (LIFO) order. This means that the last element added to the stack is the first element to be removed.

Backtracking: Backtracking is a technique for solving problems that can be broken down into a series of steps. Each step has a set of possible outcomes. If the current outcome is not successful, then the algorithm backtracks to the previous step and tries a different outcome.

The randomized DFS method with backtracking works as follows:

A cell is chosen at random and marked as visited.The neighboring cells of the visited cell are checked to see if they are also visited. If they are not, then one of them is chosen at random and marked as visited.If there are no unvisited neighboring cells, then the algorithm backtracks to the previous cell.The process repeats until all cells in the grid have been visited. This algorithm can be used to generate mazes that are both random and solvable. The use of a stack allows the algorithm to backtrack to previous cells, which prevents it from getting stuck in dead ends.

### 2.1 Algorithm

- The first step is to initialize the current position to  $(0, 0)$  and mark it as visited. This is the starting point of the algorithm.
- The next step is to push the current position onto the stack. This will allow us to keep track of where we have been and where we still need to go.
- The third step is to set a flag to indicate whether the end has been reached to False. This will allow us to terminate the algorithm once we have reached the goal.
- The fourth step is a loop that will continue to run as long as the stack is not empty and the end has not been reached.
- Inside the loop, we will first get the list of valid neighbors of the current position. This is done by checking all of the surrounding cells to see if they are not visited and if they are not walls.
- Once we have the list of valid neighbors, we will randomly choose one of them.
- If the neighbor is not visited, we will mark it as visited and push it onto the stack. We will then set the current position to the neighbor.
- If there are no valid neighbors, we will pop the top element from the stack. This will take us back to the previous position.
- The loop will continue to run until either the stack is empty or the end has been reached.
- Once the loop terminates, we will return the maze.

## 2.2 Analysis

The algorithm works accurately and traverse through all area in the matrix and provide path. Despite this accuracy, there is a chance of loop formation but the number of loops are limited and is very small amount. The time complexity of the above code is  $O(n*m)$ , where  $n$  is the number of rows in the maze and  $m$  is the number of columns. This is because the algorithm will need to check all of the cells in the maze to find the goal. The space complexity is computed considering there is  $n*m$  matrix and size of stack which cannot exceed  $n*m$ . So  $O(n*m)$  is space complexity.

## 3 Maze solving

Maze solving is the process of finding a path from the start to the end of a maze. There are many different algorithms that can be used to solve mazes, each with its own advantages and disadvantages. Some common algorithms include:

Random mouse: This algorithm simply picks a random direction at each junction and continues until it reaches the end. This algorithm is very simple to implement, but it can be very inefficient.

Wall follower: This algorithm follows the walls of the maze until it reaches the end. This algorithm is more efficient than the random mouse algorithm, but it can get stuck in loops if the maze is not well-designed.

Dijkstra's algorithm: This algorithm finds the shortest path from the start to the end of the maze. This algorithm is the most efficient, but it can be more difficult to implement. The best algorithm to use for maze solving depends on the specific maze and the resources available. For simple mazes, a simple algorithm like the random mouse algorithm may be sufficient. For more complex mazes, a more efficient algorithm like Dijkstra's algorithm may be necessary.

In addition to algorithms, there are also a number of other techniques that can be used to solve mazes. These techniques include: Brute force, Human intelligence etc. Maze solving is a challenging problem that has been studied by computer scientists for many years. There is no single best algorithm for maze solving, and the best approach often depends on the specific maze and the resources available.

In this problem, the method using a kind of above said solutions. The path is found by checking and going through all possible ways to get end point. This algorithm works by recursively exploring the maze from the start position. At each step, the algorithm checks if the current position is the end position. If it is, the algorithm returns the path to the end position. Otherwise, the algorithm checks the neighbours of the current position. If any of the neighbours are unvisited, the algorithm recursively explores that neighbour. This process continues until the end position is found or all possible paths have been explored.

### 3.1 Algorithm

- Create a copy of the maze. This will be used to keep track of the explored areas.
- Initialize the current position to (0, 0).
- Mark the current position as explored.
- Push the current position onto a stack.
- Set the inside flag to 1.
- Repeat the following steps until the current position is at the (n-1, m-1) coordinate:

- If the current position is at a wall, then: \* Pop the top position off the stack. \* Set the inside flag to 0.
- Otherwise, get the neighbors of the current position.
- If there are any neighbors, then: \* Select the first neighbor that is not a wall. \* Set the current position to the selected neighbor. \* Mark the current position as explored. \* Push the current position onto the stack. \* Set the inside flag to 1.
- Print the stack.

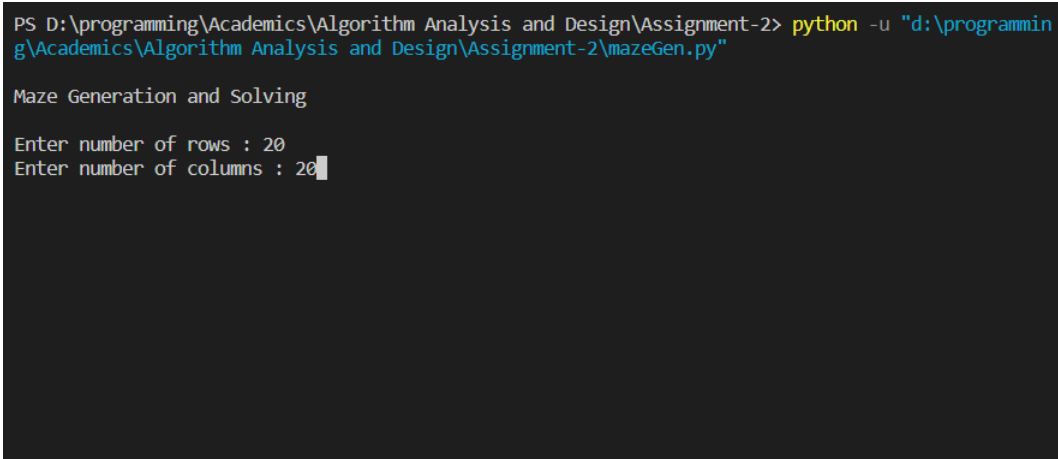
### 3.2 Analysis

The algorithm finds the path to the end point accurately, but the problem is, since going through all possible path, it takes time. The time complexity of the code is  $O(n^2)$ , where  $n$  is the number of cells in the maze. This is because the algorithm needs to explore all of the cells in the maze in order to find the exit. The space complexity of the code is  $O(n)$ , where  $n$  is the number of cells in the maze. This is because the algorithm needs to store the maze, the stack, and the inside flag.

## 4 Result

The method used here generates and solves the maze even though it contains certain time issue and looping problem. The techniques used here generates accurate solution and provides a visual representation.

## 5 Test



```

PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-2> python -u "d:\programming\Academics\Algorithm Analysis and Design\Assignment-2\mazeGen.py"

Maze Generation and Solving

Enter number of rows : 20
Enter number of columns : 20
  
```

Figure 5.1: Output

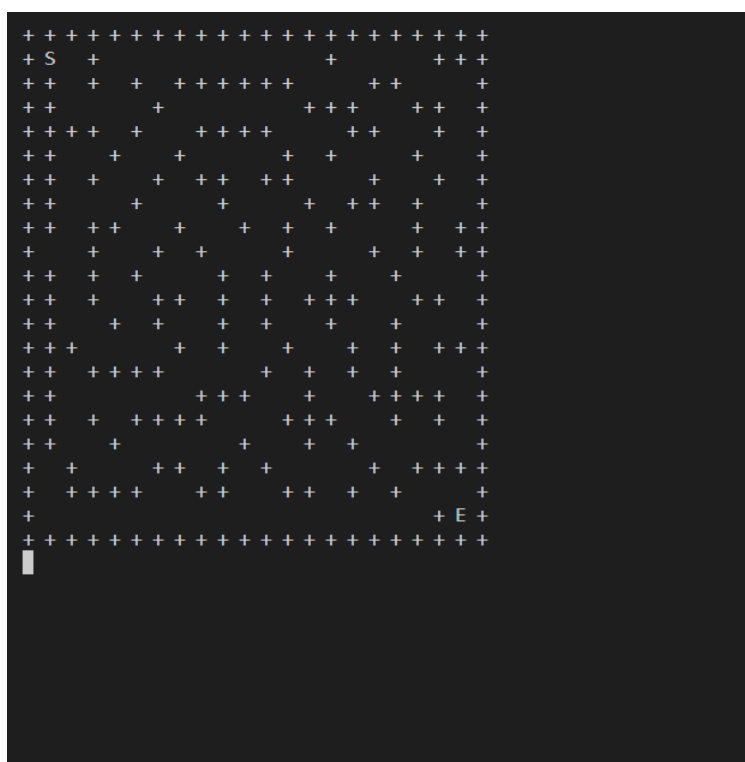


Figure 5.2: Output

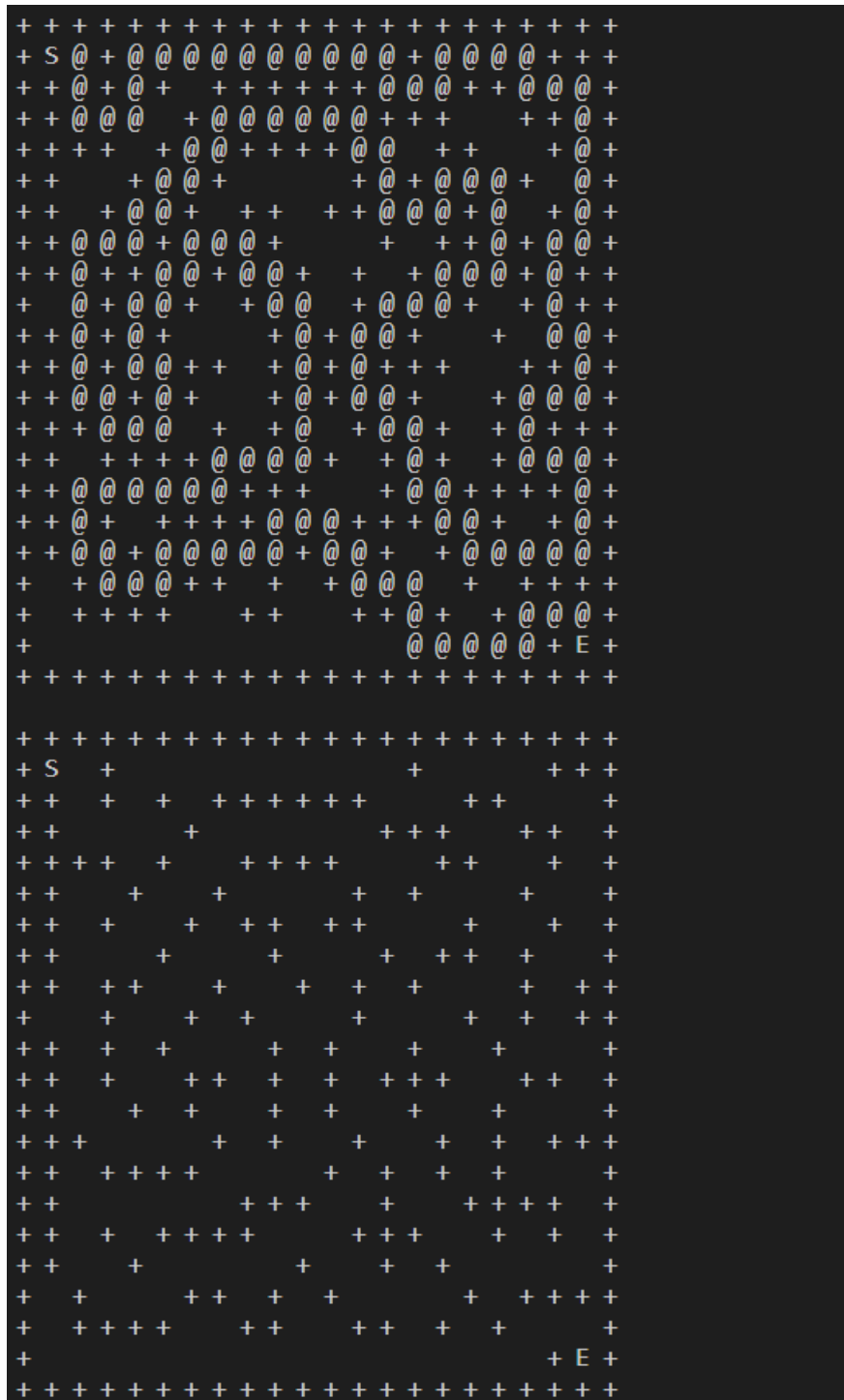


Figure 5.3: Output

```
[
  (0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (1, 3), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7),
  (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (1, 12), (1, 13), (1, 14), (0, 14), (0, 15), (0, 16),
  (0, 17), (1, 17), (1, 18), (1, 19), (2, 19), (3, 19), (4, 19), (5, 19), (6, 19), (6, 18), (7,
  18), (8, 18), (9, 18), (9, 19), (10, 19), (11, 19), (11, 18), (11, 17), (12, 17), (13, 17), (13,
  18), (13, 19), (14, 19), (15, 19), (16, 19), (16, 18), (16, 17), (16, 16), (16, 15), (15, 15),
  (15, 14), (14, 14), (14, 13), (13, 13), (12, 13), (12, 12), (11, 12), (11, 11), (10, 11), (9, 11),
  (9, 12), (8, 12), (8, 13), (8, 14), (7, 14), (7, 15), (7, 16), (6, 16), (5, 16), (4, 16), (4,
  15), (4, 14), (5, 14), (5, 13), (5, 12), (4, 12), (3, 12), (3, 11), (2, 11), (2, 10), (2, 9), (
  2, 8), (2, 7), (2, 6), (3, 6), (3, 5), (4, 5), (4, 4), (5, 4), (5, 3), (6, 3), (6, 2), (6, 1), (
  7, 1), (8, 1), (9, 1), (10, 1), (11, 1), (11, 2), (12, 2), (12, 3), (12, 4), (11, 4), (10, 4), (
  10, 3), (9, 3), (8, 3), (8, 4), (7, 4), (7, 5), (6, 5), (6, 6), (6, 7), (7, 7), (7, 8), (8, 8),
  (8, 9), (9, 9), (10, 9), (11, 9), (12, 9), (13, 9), (13, 8), (13, 7), (13, 6), (14, 6), (14, 5),
  (14, 4), (14, 3), (14, 2), (14, 1), (15, 1), (16, 1), (16, 2), (17, 2), (17, 3), (17, 4), (16,
  4), (16, 5), (16, 6), (16, 7), (16, 8), (15, 8), (15, 9), (15, 10), (16, 10), (16, 11), (17, 11),
  (17, 12), (17, 13), (18, 13), (19, 13), (19, 14), (19, 15), (19, 16), (19, 17), (18, 17), (18,
  18), (18, 19), (19, 19)]
```

Figure 5.4: Output