Assiut University

Faculty of Computers and Information

Department of Computer Science

Artificial Intelligence (CS361 - Course)

Maze Problem

Team Members:

1. *Alyaa Ezz Shahat (Code and Methods, Results).*
2. *Huda Mawood Kamal (Introduction, Format and Layout).*
3. *Nadia Mostafa Shehata (Introduction, Format and Layout).*
4. *Asmaa Hassan Ali (Design).*

Contents

0.1Introduction

0.1.1definition

Mazes have been fascinating humans for centuries. From the labyrinth of Greek mythology to the garden mazes of the European nobility, these intricate networks of paths and walls are both challenging and fun. But did you know that you can create and solve mazes programmatically using Python? In this blog post, we’ll walk you through the process, step by step.

0.1.2importance

The generalized maze problem is a versatile tool with a wide range of applications. By modeling various real-world scenarios as maze problems, researchers and practitioners can develop, test, and implement solutions that improve efficiency, safety, and performance across multiple domains, from robotics and network routing to game development and urban planning.

0.1.3Solvers and Algorithms

There is more than one algorithm to solve the Maze problem such as :

the Random mouse algorithm, Dead-end filling algorithm, Recursive algorithm, Hand On Wall Rule algorithm, Pledge algorithm, Trémaux's algorithm algorithm, Maze-routing algorithm and Randomized depth-first search algorithm.

Each algorithm has its own strengths and weaknesses, and the choice of algorithm may depend on factors such as the number of queens and the size of the board.

The Random mouse algorithm

This is a trivial method that can be implemented by a very unintelligent robot or perhaps a mouse. It is simply to proceed following the current passage until a junction is reached, and then to make a random decision about the next direction to follow. Although such a method would always eventually find right solution, this algorithm can be extremely slow..

The Dead-end filling

Dead-end filling is an algorithm for solving mazes that fills all dead ends, leaving only the correct ways unfilled. It can be used for solving mazes on paper or with a computer program, but it is not useful to a person inside an unknown maze since this method looks at the entire maze at once. The method is to

1. find all of the dead-ends in the maze, and then
2. "fill in" the path from each dead-end until the first junction is met.

Note that some passages won't become parts of dead end passages until other dead ends are removed first. A video of dead-end filling in action can be seen to the right.

Dead-end filling cannot accidentally "cut off" the start from the finish since each step of the process preserves the topology of the maze. Furthermore, the process won't stop "too soon" since the result cannot contain any dead-ends. Thus if dead-end filling is done on a perfect maze (maze with no loops), then only the solution will remain. If it is done on a partially braid maze (maze with some loops), then every possible solution will remain but nothing more.

.

The Recursive algorithm

If given an omniscient view of the maze, a simple recursive algorithm can tell one how to get to the end. The algorithm will be given a starting X and Y value. If the X and Y values are not on a wall, the method will call itself with all adjacent X and Y values, making sure that it did not already use those X and Y values before. If the X and Y values are those of the end location, it will save all the previous instances of the method as the correct path

The Hand On Wall Rule

The Hand on Wall Rule (left-hand or right-hand rule) guarantees finding an exit in a simply-connected maze (where all walls are connected together or to the outer boundary). By keeping one hand in contact with a wall, you will traverse every corridor next to that connected section of walls at least once.

This algorithm is a depth-first in-order tree traversal. Topologically, it can be seen as walking around a loop or circle formed by the connected maze walls.

If the maze is not simply-connected (e.g. start/endpoints are surrounded by passage loops, or paths cross over/under each other), the wall-following method may not reach the goal.

It's important to start wall-following at the maze entrance. If you start at an arbitrary point inside a non-simply-connected maze, you could get trapped in a separate looping wall with no exits.

In higher-dimensional mazes (e.g. 3D), wall-following can be applied if the passages can be projected onto a 2D plane in a deterministic way, and the current orientation is known to determine left/right.

The Pledge Algorithm is mentioned as an alternative methodology for non-simply-connected mazes.

The Pledge algorithm

The Pledge algorithm, named after John Pledge, solves the problem of navigating disjoint mazes from any starting point inside to an outer exit. Unlike the wall follower method, which can trap the solver in loops, the Pledge algorithm uses a chosen preferential direction and counts turns (positive for clockwise, negative for counter-clockwise) to navigate around obstacles. The solver leaves the obstacle when the sum of turns and the current heading return to zero, effectively avoiding traps. This method ensures finding an outer exit from inside the maze but does not work for finding a goal within the maze from an outside entrance

The Trémaux's algorithm

Trémaux's algorithm, invented by Charles Pierre Trémaux, is an efficient method for navigating mazes. The algorithm involves marking entrances of passages as you move through the maze and follows these rules:

Mark Entrances: Whenever you pass through an entrance, leave a mark.

Choosing an Exit at a Junction:

If only the entrance you just came from is marked, choose an unmarked entrance if available.

If at a dead end, turn around and go back the way you came, unless that entrance is marked twice.

Otherwise, choose an entrance with the fewest marks (preferably zero, else one)

The Maze-routing algorithm

The maze-routing algorithm is a low overhead method to find the way between any two locations of the maze. The algorithm is initially proposed for chip multiproccessors (CMPs) domain and guarantees to work for any grid-based maze. In addition to finding paths between two locations of the grid (maze), the algorithm can detect when there is no path between the source and destination. Also, the algorithm is to be used by an inside traveler with no prior knowledge of the maze with fixed memory complexity regardless of the maze size; requiring 4 variables in total for finding the path and detecting the unreachable locations. Nevertheless, the algorithm is not to find the shortest path.

Maze-routing algorithm uses the notion of Manhattan distance (MD) and relies on the property of grids that the MD increments/decrements *exactly* by 1 when moving from one location to any 4 neighboring locations.

The Randomized depth-first search

the recursive backtracker algorithm for maze generation:

This is a randomized version of the depth-first search algorithm, frequently implemented using a stack.

It starts at a random cell in the grid, and then randomly selects an unvisited neighboring cell. It removes the wall between the two cells and marks the new cell as visited, adding it to the stack.

If a cell has no unvisited neighbors, it is considered a dead-end. The algorithm then backtracks through the stack until it reaches a cell with an unvisited neighbor, and continues the path from there.
This process continues until every cell has been visited, at which point the algorithm backtracks all the way back to the starting cell.
To avoid stack overflow issues, the algorithm can be rearranged into a loop that stores the backtracking information directly in the maze itself.
Mazes generated using this depth-first search approach tend to have a low branching factor and many long corridors, as the algorithm explores as far as possible along each branch before backtracking.

0.2 Methodology

0.2.1 Depth-Dirst Dearch algorithm

DFS is straightforward to implement using recursion or an explicit stack. It explores each branch of the maze deeply before backtracking, making it easy to follow. DFS systematically explores all possible paths from the start point. If there's a path to the end, DFS will find it. If no path exists, DFS will explore all possibilities and indicate that no solution is found.
For relatively small mazes, like an 8x8 grid, DFS performs well. It doesn't require significant memory or computational resources, making it suitable for typical maze problems.

Represent the Maze:

- The maze can be represented as a 2D grid or matrix where each cell is either open (path) or blocked (wall).

Define the Starting and Ending Points:

- Identify the coordinates of the starting point (e.g., top-left corner) and the ending point (e.g., bottom-right corner).

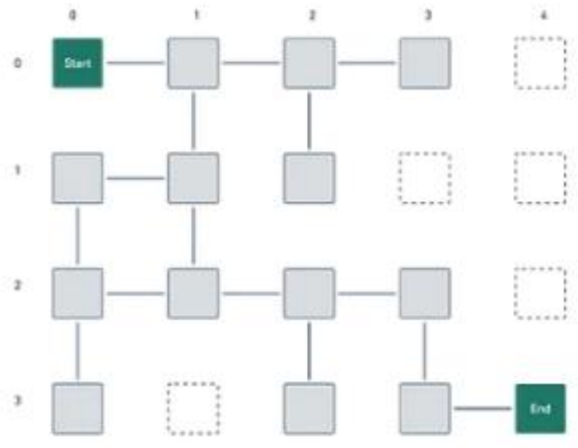


Figure 0

- **Initialize the Data Structures**:

  We use a stack to manage the path traversal (LIFO order).
  Maintain a set to keep track of visited nodes to avoid revisiting and looping.

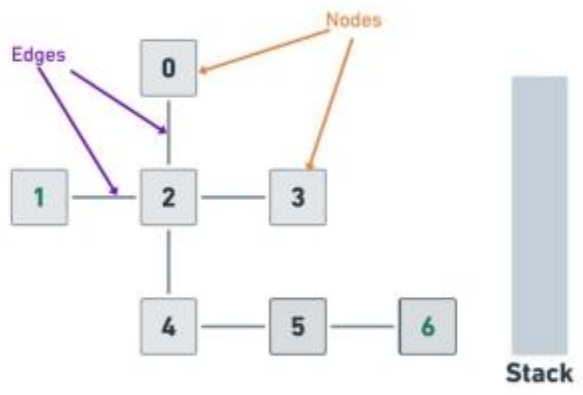


Figure 1-Nodes, Edges, and an empty stack.

0.2.2 Backtracking algorithms steps

- **DFS Traversal**:
  - **In Step 1 as shown in figure 2:**
    - We call DFS on *Node 1* which puts it on the call stack.
    - We mark Node 1 as visited to avoid evaluating it again, which could prevent finding a path.

- We check if Node 1 is the target node (Node 6). Since it's not, we proceed to Node 1’s adjacency list.
- Each node has a list of its connected nodes. We see that Node 2 is connected to Node 1 and proceed to the next step.

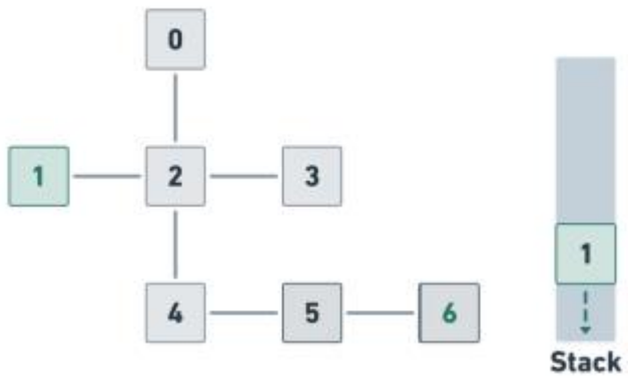


Figure 2

- **In Step 2 as shown in figure 3:**
  - We mark the node as visited.
  - We check if it's the node we're looking for.
  - It’s not so we call DFS on **Node 2** to go even deeper into the graph.

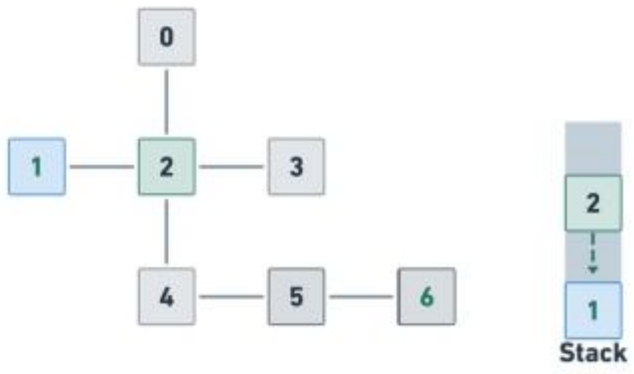


Figure 3

- **In Step 3 as shown in figure 4:**
  - We move onto Node 0, mark it as visited, and check if it's equal to Node 6.
  - We call Depth-First Search (DFS) on Node 0 to explore deeper into the maze.
  - Node 0 only connects to Node 2, which we've already visited.

- Since we can't move further, we perform backtracking, stopping further function calls and retracing steps to explore other paths.

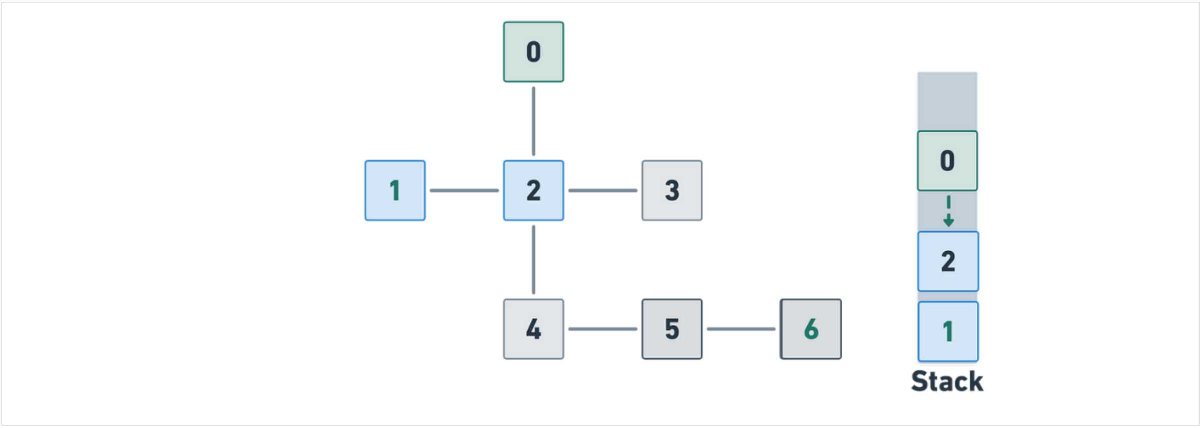


Figure 4

- **In Step 4 as shown in figure 5:**
  - **Node 1** and **Node 2** are still on the stack because they are still part of a potential path.
  - Once we hit Node 0 and find no other paths, we complete the DFS call on Node 0 and backtrack to Node 2 to continue exploring deeper.
  - We look at the next node in 2’s adjacency list and begin to evaluate **Node 3.**

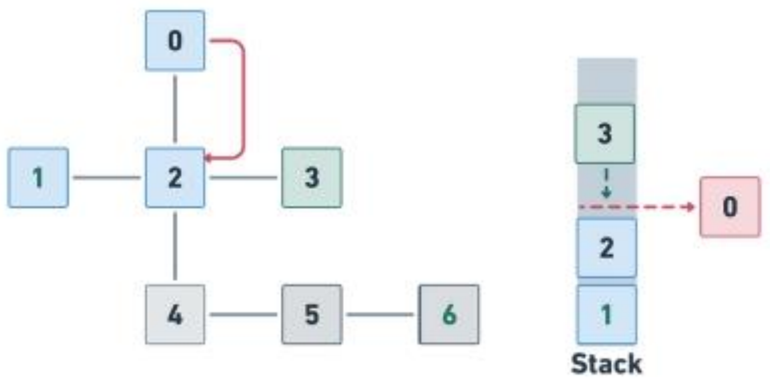


Figure 5

- **In Step 5 as shown in figure 6:**
  - We end the DFS call on Node 3 and thus removing it from the stack.
  - We hop back to 2's adjacency list and begin to evaluate **Node 4.**
  - We notice that **Node 4** is connected to another node we haven't traversed yet and begin the DFS cycle again!

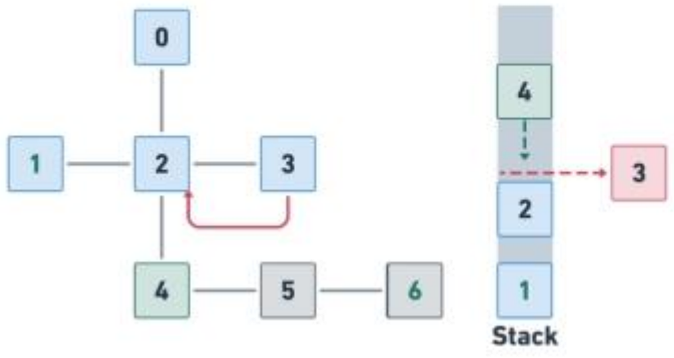
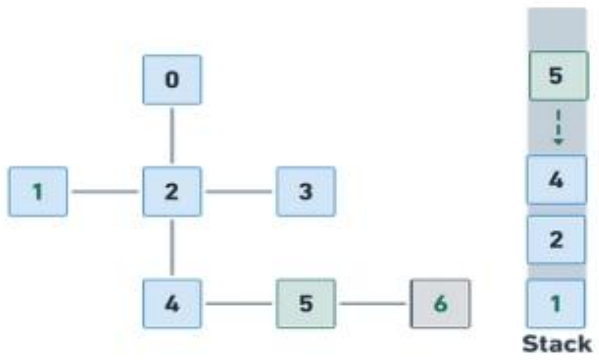


Figure 6

- **In Step 6 and Step 7 as shown in figure 7, 8:**
  - Visiting each node sequentially, we perform a comparison and call DFS on every node encountered.
  - Upon reaching Node 6, our goal check is satisfied, marking the end of our search.
  - Examination of the call stack in figure 9 reveals a path from Node 6 back to Node 1, showcasing the effectiveness of backtracking.
  - With the goal achieved, we remove DFS calls from the stack and print the values, obtaining our desired path.



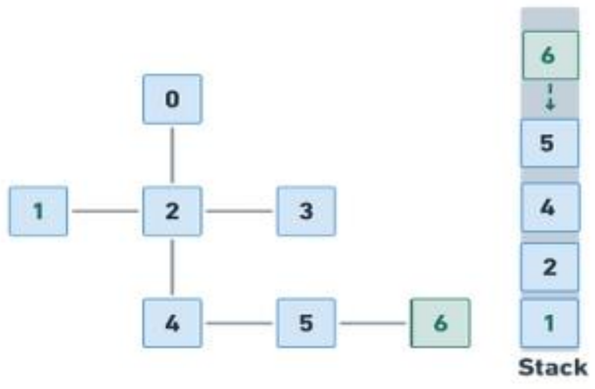


Figure 7 Figure 8

- What we did above with DFS is exactly how we'll solve our maze.

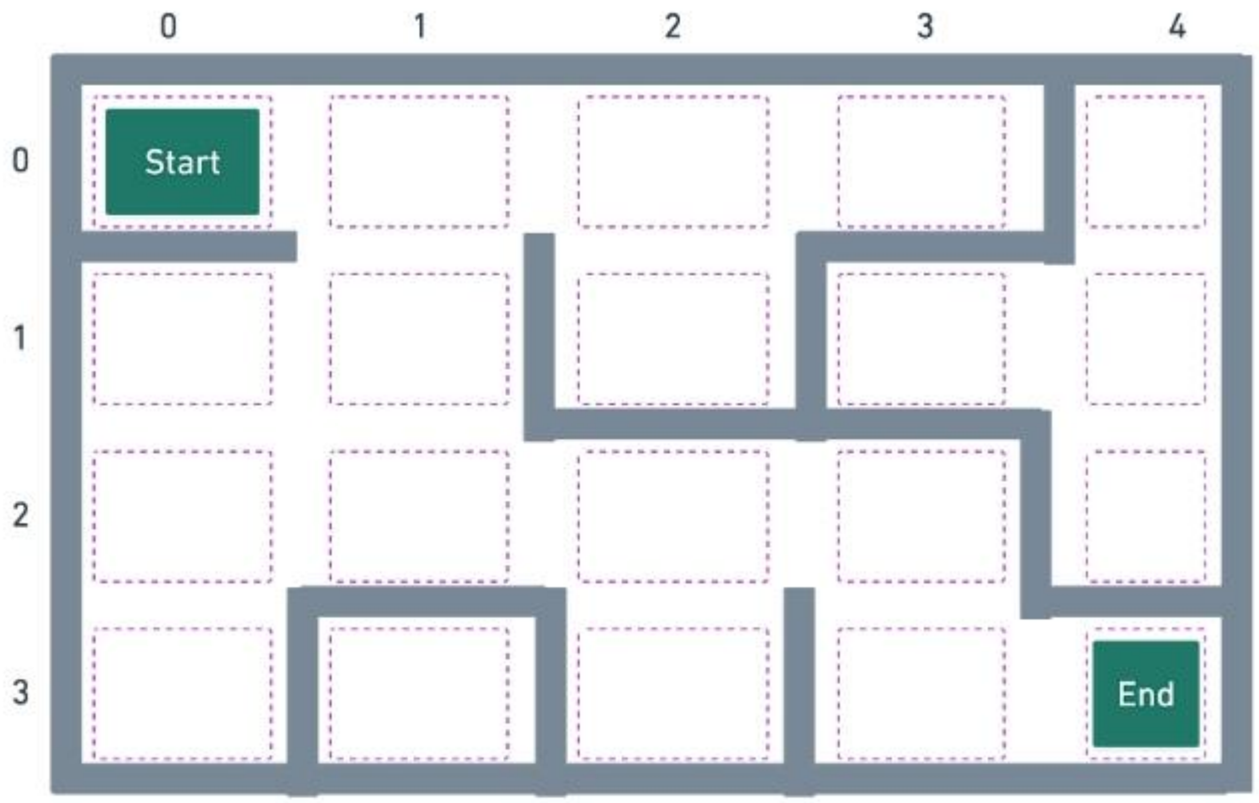


Figure 9

0.2.3algorithm pseudocode

showin in figure 10

0.2.4 complexity analysi

DFS Maze Solving Complexity

Time Complexity

Factors:

NN: Number of cells.

EE: Number of edges (passages).

Traversal: Each cell is visited once.

Edge Exploration: Each edge is explored once.

Result: O(N+E)O(N+E)

Space Complexity:

Factors:

Recursion stack depth up to NN.

Array/set of size NN for visited cells.

Result: O(N)O(N)

0.3 Experimental simulations

0.3.1 Implementing Deapth first search Algorithm to Solve the maze Problem

1. *Initialization*

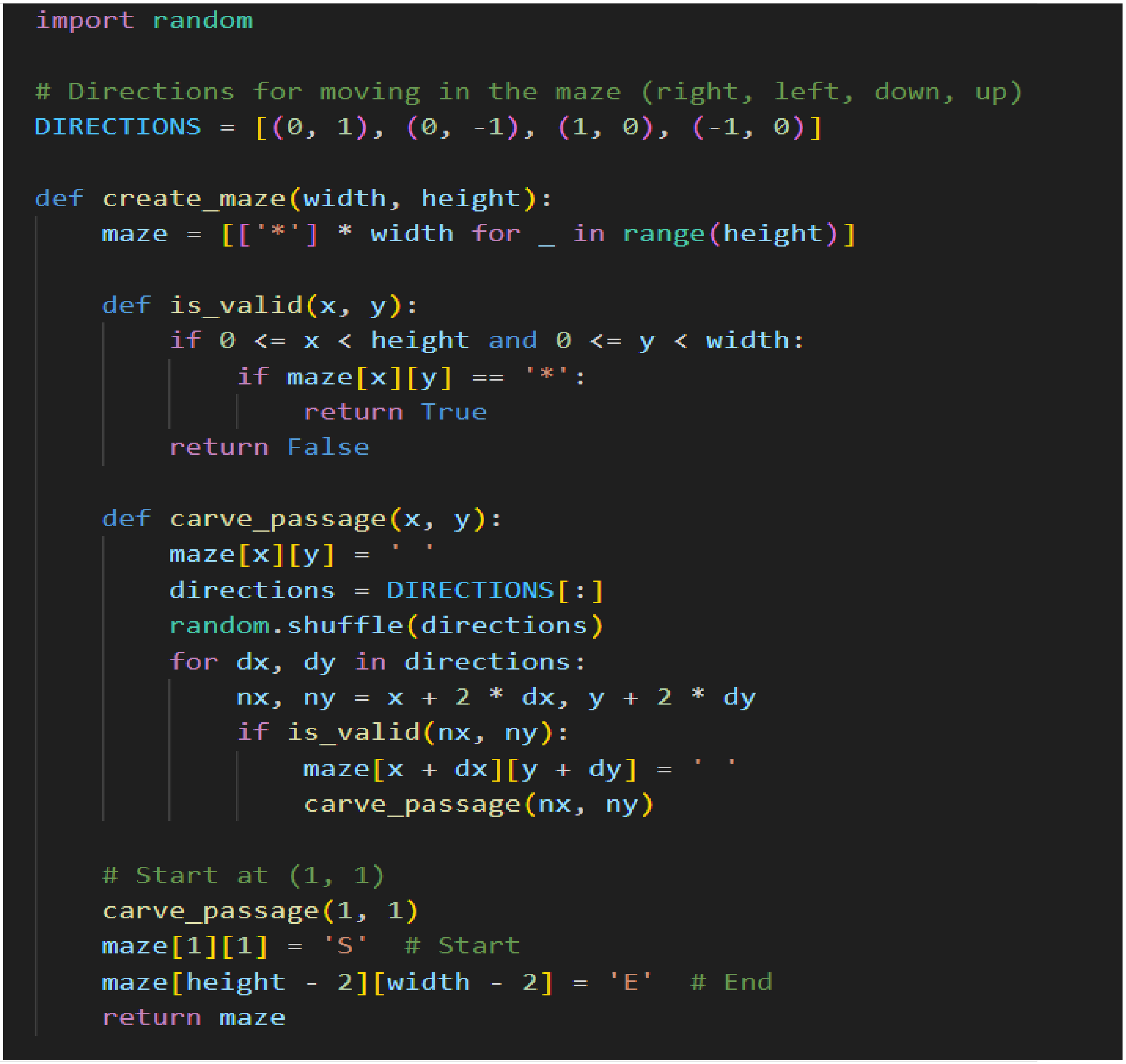
- **Maze Input**: Use the maze generated previously.
- **Starting Point**: Identify the start cell (**'S'**), typically **(1, 1)**.
- **End Point**: Identify the end cell (**'E'**), typically **(height - 2, width - 2)**.

3. *DFS for Path Finding*

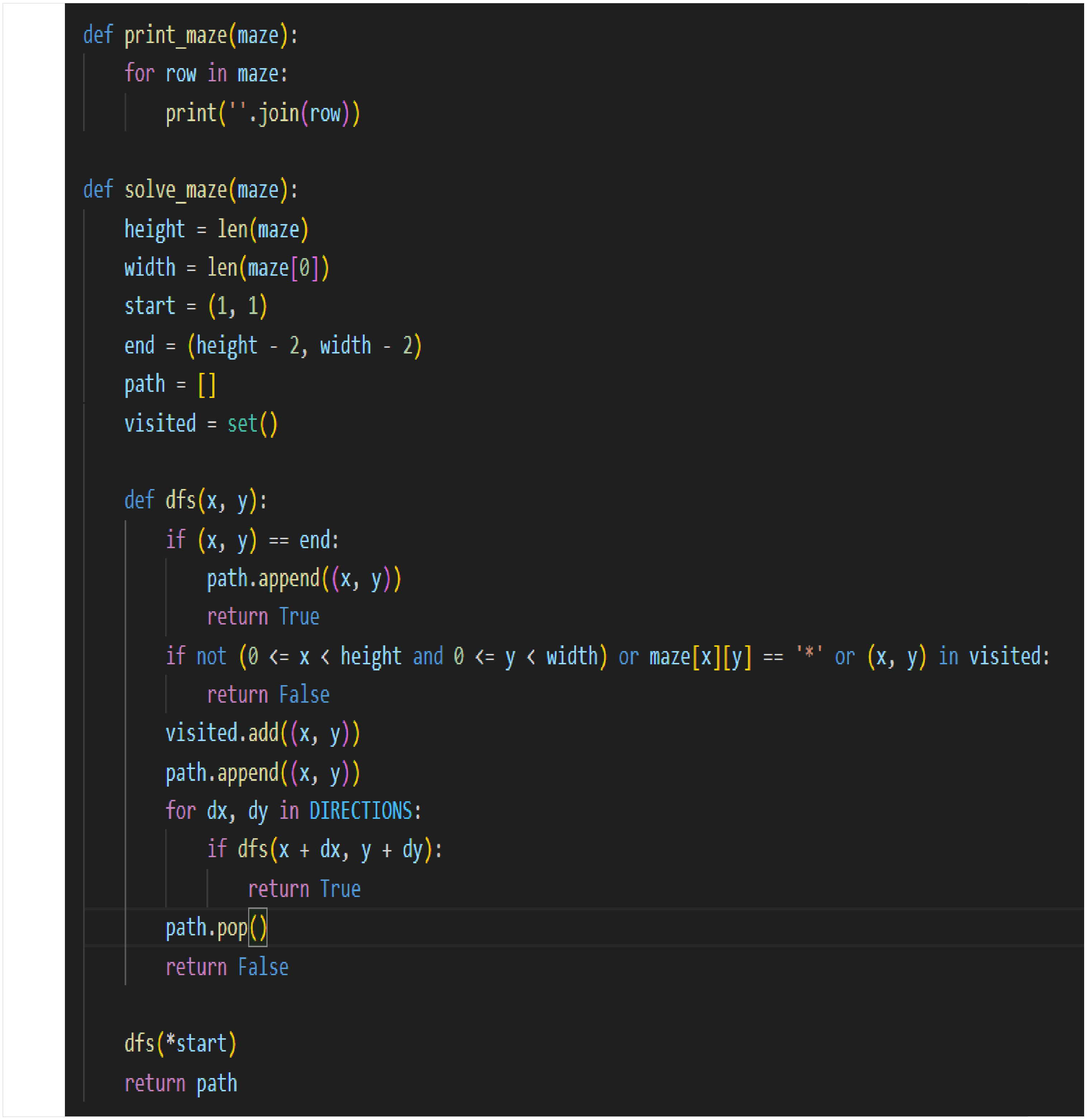
- **Visited Set**: Keep track of visited cells to avoid cycles.
- **Path Tracking**: Maintain a list to store the current path.

4. *Recursive DFS*

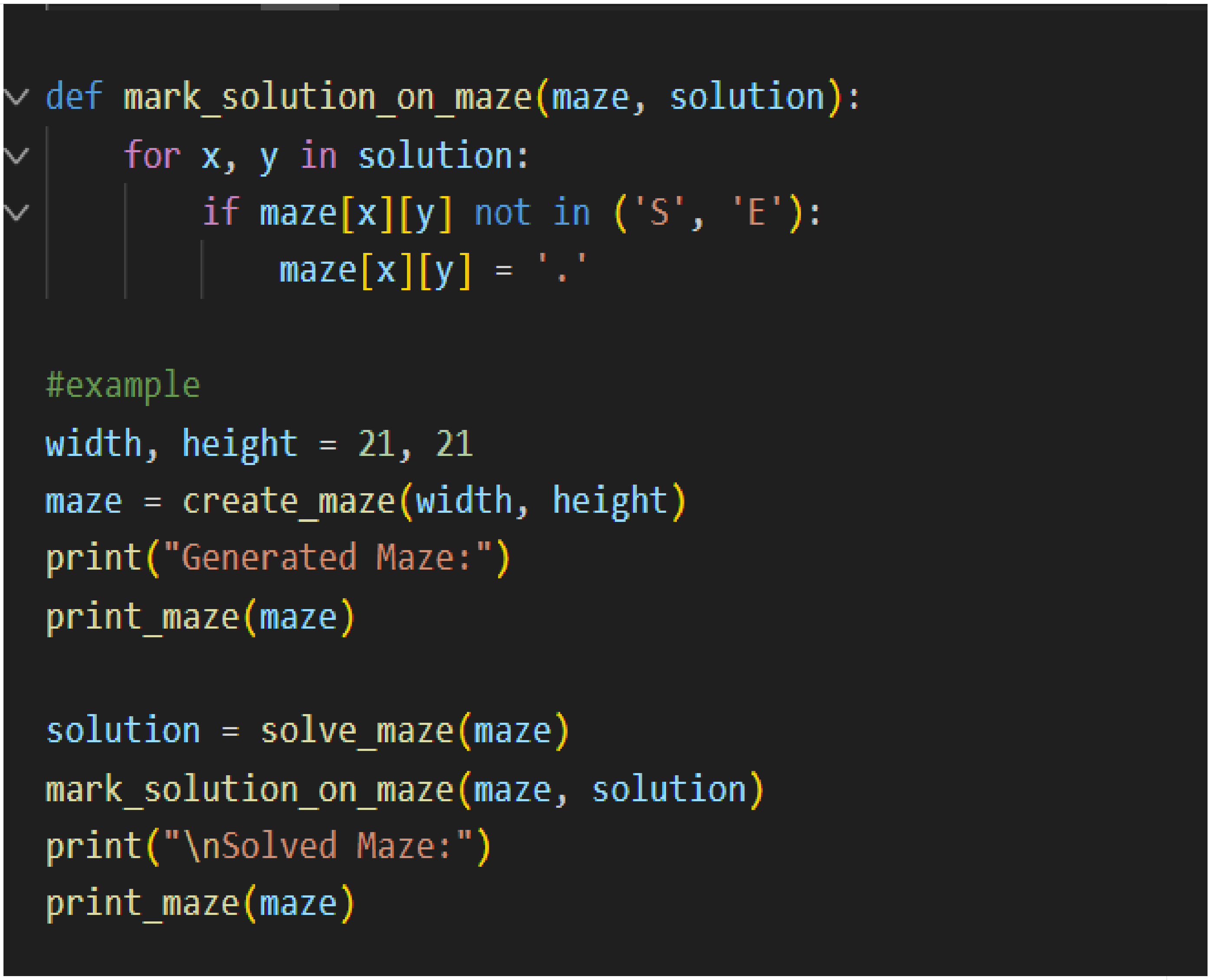
- **Base Case**: If the current cell is the end cell, add it to the path and return **True**.
- **Recursive Case**:
  - Mark the current cell as visited.
  - For each direction, calculate the coordinates of the next cell.
  - If the next cell is within bounds, a passage (`' '`), and not visited, recursively apply DFS from the next cell.
  - If a valid path is found, add the current cell to the path and return **`True`**.
  - If no valid path is found from the current cell, backtrack by removing it from the path and return **`False`**.



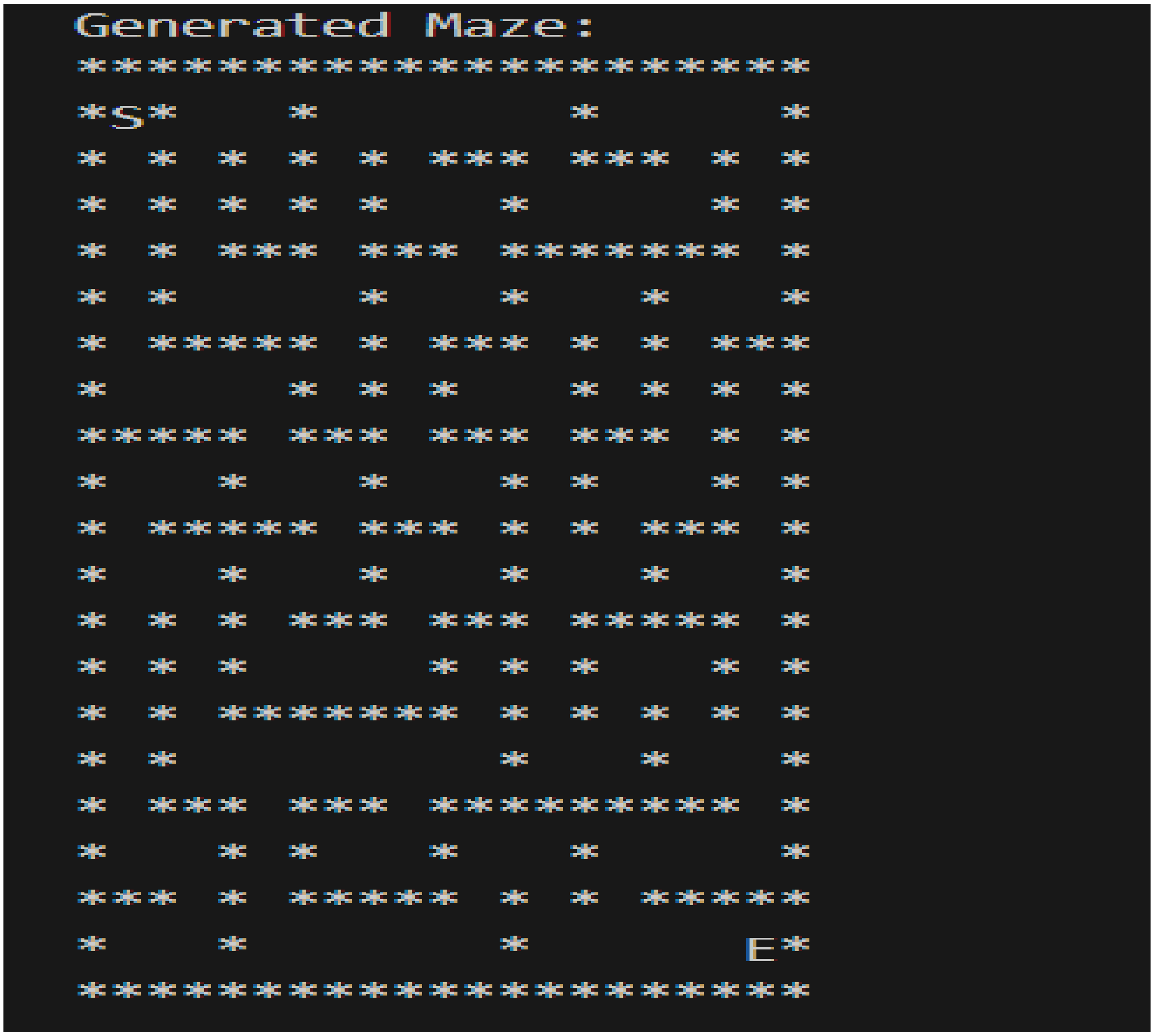
```python
import random

# Directions for moving in the maze (right, left, down, up)
DIRECTIONS = [(0, 1), (0, -1), (1, 0), (-1, 0)]

def create_maze(width, height):
    maze = [['*'] * width for _ in range(height)]

    def is_valid(x, y):
        if 0 <= x < height and 0 <= y < width:
            if maze[x][y] == '*':
                return True
        return False

    def carve_passage(x, y):
        maze[x][y] = ' '
        directions = DIRECTIONS[:]
        random.shuffle(directions)
        for dx, dy in directions:
            nx, ny = x + 2 * dx, y + 2 * dy
            if is_valid(nx, ny):
                maze[x + dx][y + dy] = ' '
                carve_passage(nx, ny)

    # Start at (1, 1)
    carve_passage(1, 1)
    maze[1][1] = 'S'  # Start
    maze[height - 2][width - 2] = 'E'  # End
    return maze
```



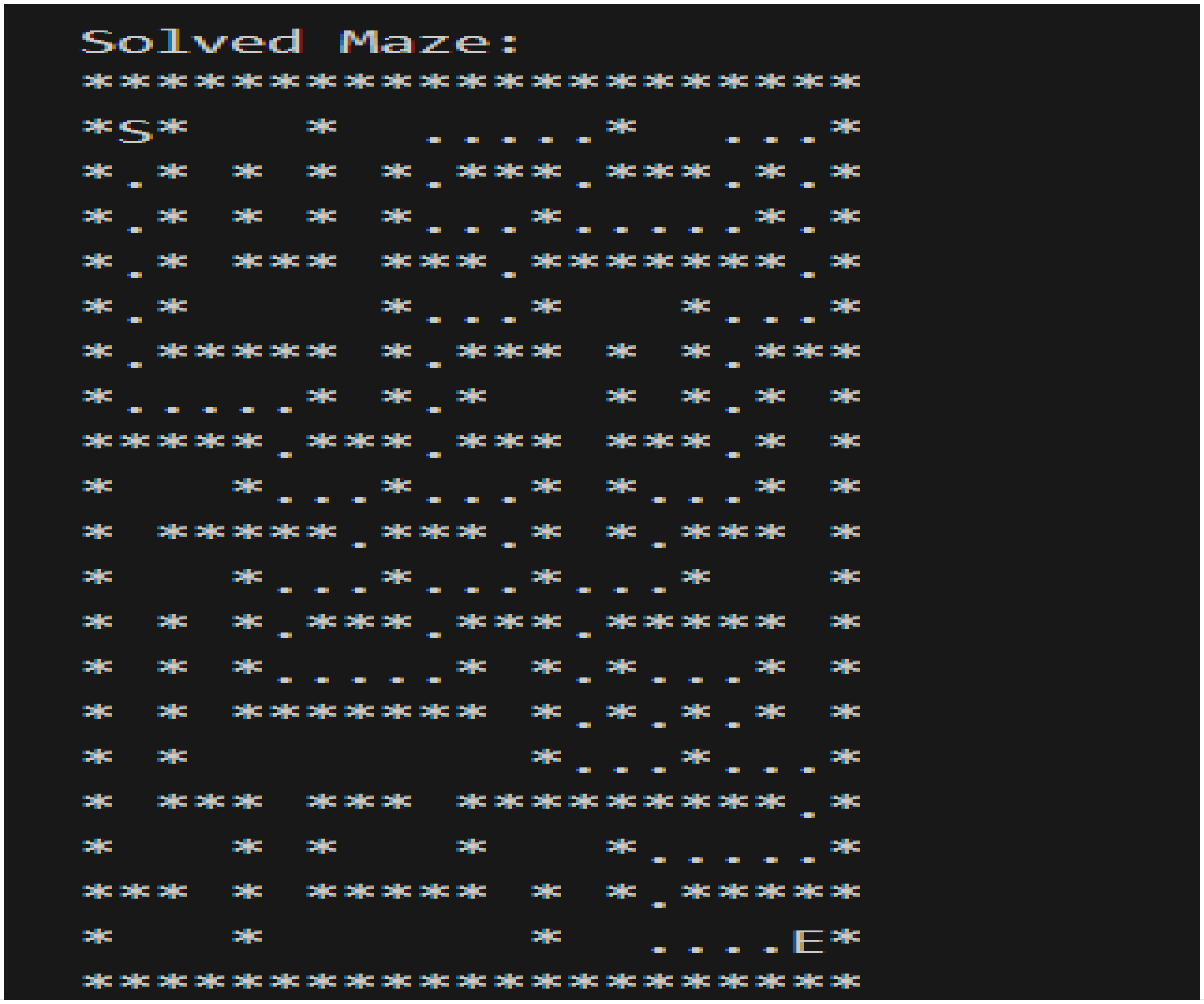
```python
def print_maze(maze):
    for row in maze:
        print(''.join(row))

def solve_maze(maze):
    height = len(maze)
    width = len(maze[0])
    start = (1, 1)
    end = (height - 2, width - 2)
    path = []
    visited = set()

    def dfs(x, y):
        if (x, y) == end:
            path.append((x, y))
            return True
        if not (0 <= x < height and 0 <= y < width) or maze[x][y] == '*' or (x, y) in visited:
            return False
        visited.add((x, y))
        path.append((x, y))
        for dx, dy in DIRECTIONS:
            if dfs(x + dx, y + dy):
                return True
        path.pop()
        return False

    dfs(*start)
    return path
```



```python
def mark_solution_on_maze(maze, solution):
    for x, y in solution:
        if maze[x][y] not in ('S', 'E'):
            maze[x][y] = '.'

#example
width, height = 21, 21
maze = create_maze(width, height)
print("Generated Maze:")
print_maze(maze)

solution = solve_maze(maze)
mark_solution_on_maze(maze, solution)
print("\nSolved Maze:")
print_maze(maze)
```



```
Generated Maze:
*********************
*S*   *       *     *
* * * * * *** *** * *
* * * * *   *     * *
* * *** *** ******* *
* *     *   *   *   *
* ***** * *** * * ***
*     * * *   * * * *
***** *** *** *** * *
*   *   *   * *   * *
* ***** *** * * *** *
*   *   *   *   *   *
* * * *** *** ***** *
* * *     * * *   * *
* * ******* * * * * *
* *         *   *   *
* *** *** ********* *
*   * *   *   *     *
*** * ***** * * *****
*   *       *      E*
*********************
```

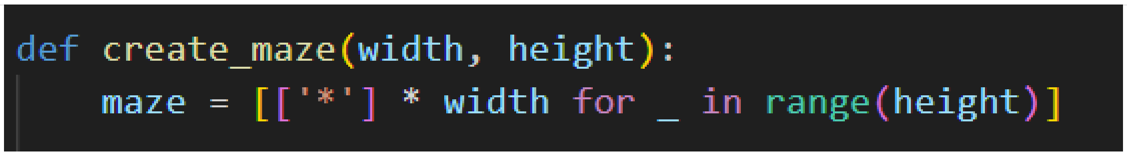


```
Solved Maze:
*********************
*S*   *  .....*  ...*
*.* * * *.***.***.*.*
*.* * * *...*.....*.*
*.* *** ***.*******.*
*.*     *...*   *...*
*.***** *.*** * *.***
*.....* *.*   * *.* *
*****.***.*** ***.* *
*   *...*...* *...* *
* *****.***.* *.*** *
*   *...*...*...*   *
* * *.***.***.***** *
* * *.....* *.*...* *
* * ******* *.*.*.* *
* *         *...*...*
* *** *** *********.*
*   * *   *   *.....*
*** * ***** * *.*****
*   *       *  ....E*
*********************
```

1. create_maze(width, height)

Generate a random maze using Depth-First Search (DFS).

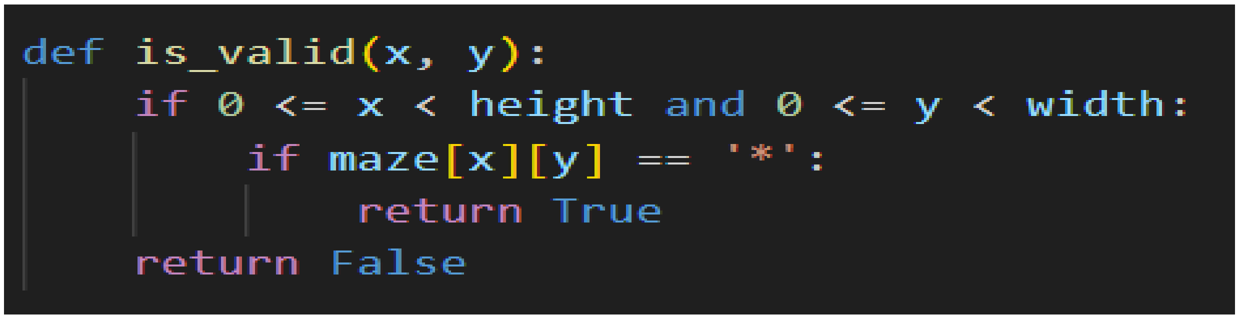
- **Input**: Width and height of the maze.
- **Output**: A 2D list representing the maze.



```python
def create_maze(width, height):
    maze = [['*'] * width for _ in range(height)]
```

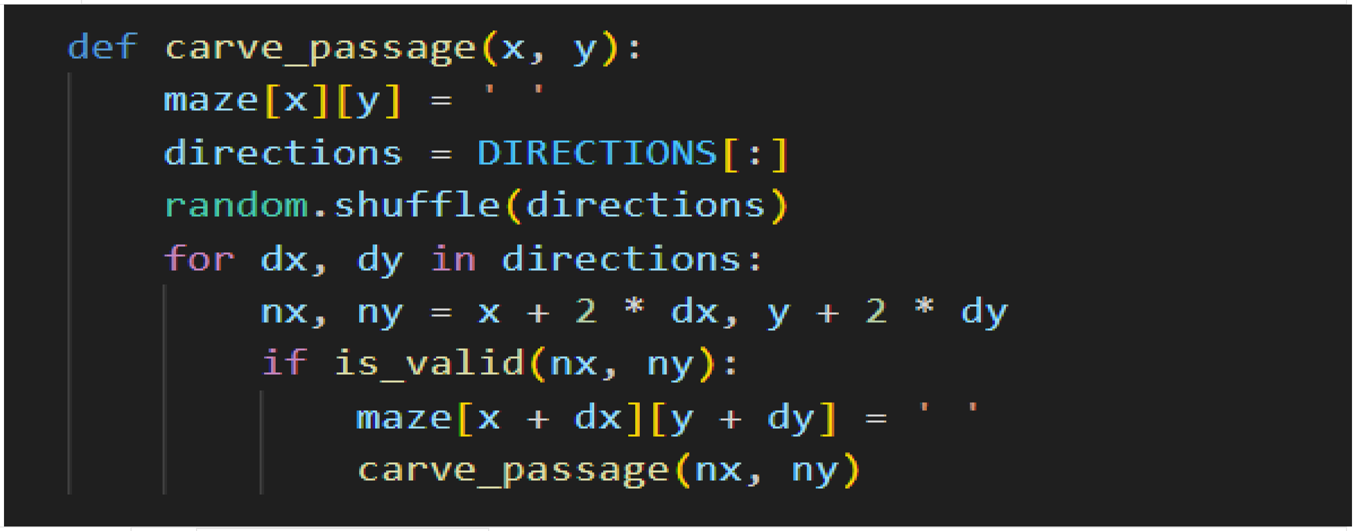
Time complexity for Maze Solving: O(height * width)

- **Initialization:** Creates a grid (**maze**) filled with walls (**'*'**).



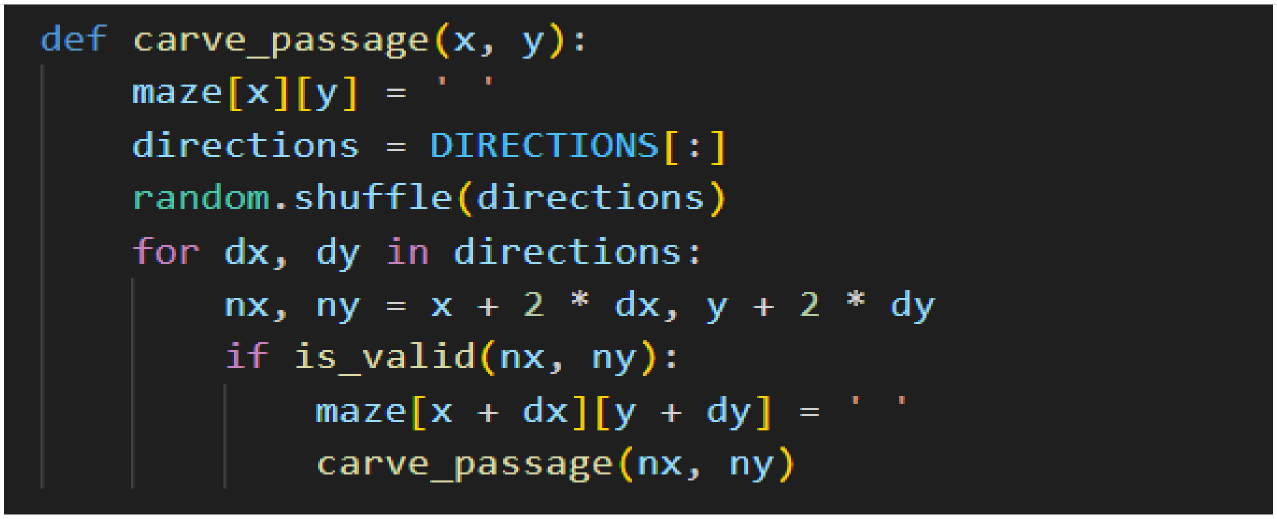
```python
def is_valid(x, y):
    if 0 <= x < height and 0 <= y < width:
        if maze[x][y] == '*':
            return True
    return False
```

  - **is valid(x, y)**: Checks if the cell **(x, y)** is within bounds and is a wall (**'*'**)

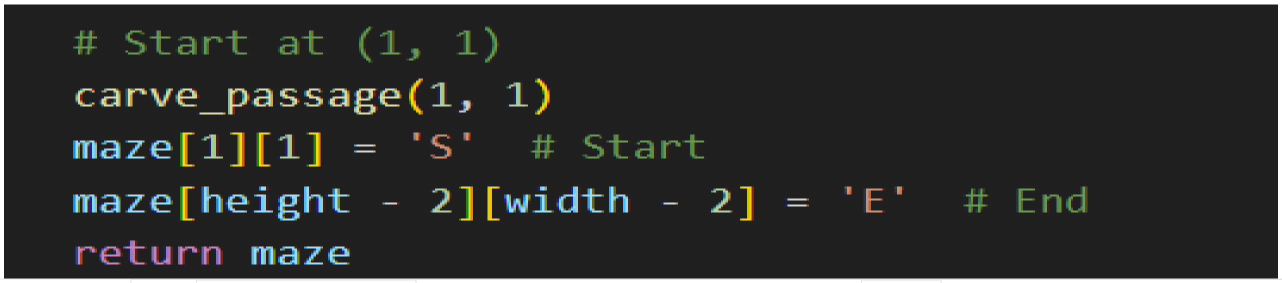


```python
def carve_passage(x, y):
    maze[x][y] = ' '
    directions = DIRECTIONS[:]
    random.shuffle(directions)
    for dx, dy in directions:
        nx, ny = x + 2 * dx, y + 2 * dy
        if is_valid(nx, ny):
            maze[x + dx][y + dy] = ' '
            carve_passage(nx, ny)
```

  - **carve passage(x, y)**: Recursively carves out paths in the maze:
    - Marks the current cell as a passage (**' '**).
    - Randomly shuffles directions to ensure randomness.
    - For each direction, calculates the next cell two steps away and checks if it can carve a path there.
    - If valid, carves the intermediate cell and recursively carves from the new cell.



```python
def carve_passage(x, y):
    maze[x][y] = ' '
    directions = DIRECTIONS[:]
    random.shuffle(directions)
    for dx, dy in directions:
        nx, ny = x + 2 * dx, y + 2 * dy
        if is_valid(nx, ny):
            maze[x + dx][y + dy] = ' '
            carve_passage(nx, ny)
```



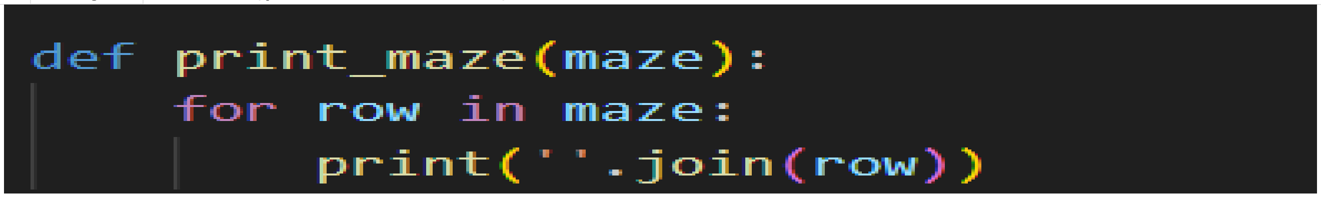
```python
# Start at (1, 1)
carve_passage(1, 1)
maze[1][1] = 'S'  # Start
maze[height - 2][width - 2] = 'E'  # End
return maze
```

- **Starting Carve**: Begins carving from the cell **(1, 1)**.
- **Start and End Points**: Marks the start (**'S'**) and end (**'E'**) points.
- **Return**: The generated maze.

2. print_maze(maze)

· **Input**: A 2D list representing the maze.

· **Output**: None (prints the maze).



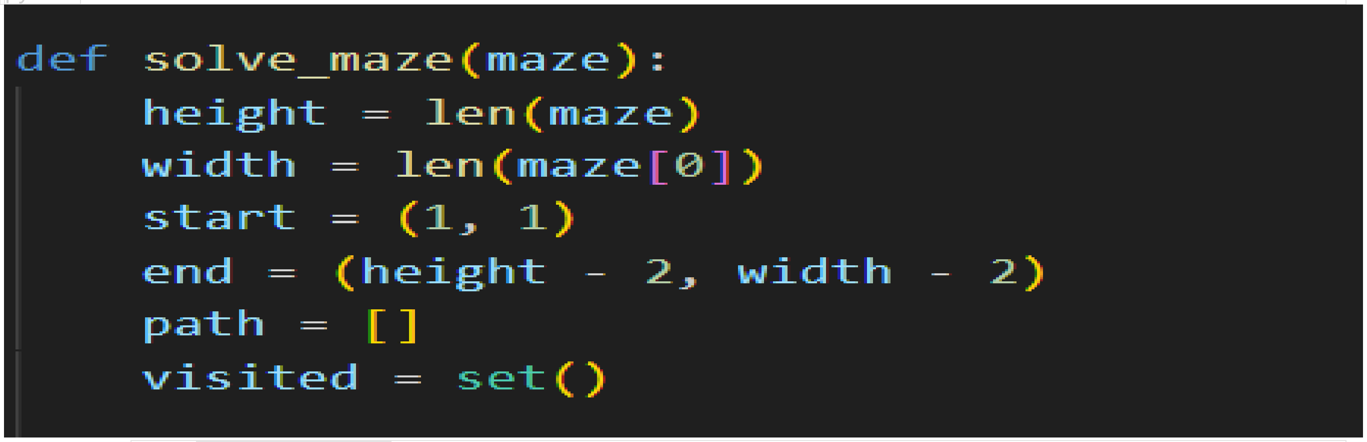
```python
def print_maze(maze):
    for row in maze:
        print(''.join(row))
```

3. solve_maze(maze)

Purpose: Solve the maze using Depth-First Search (DFS).

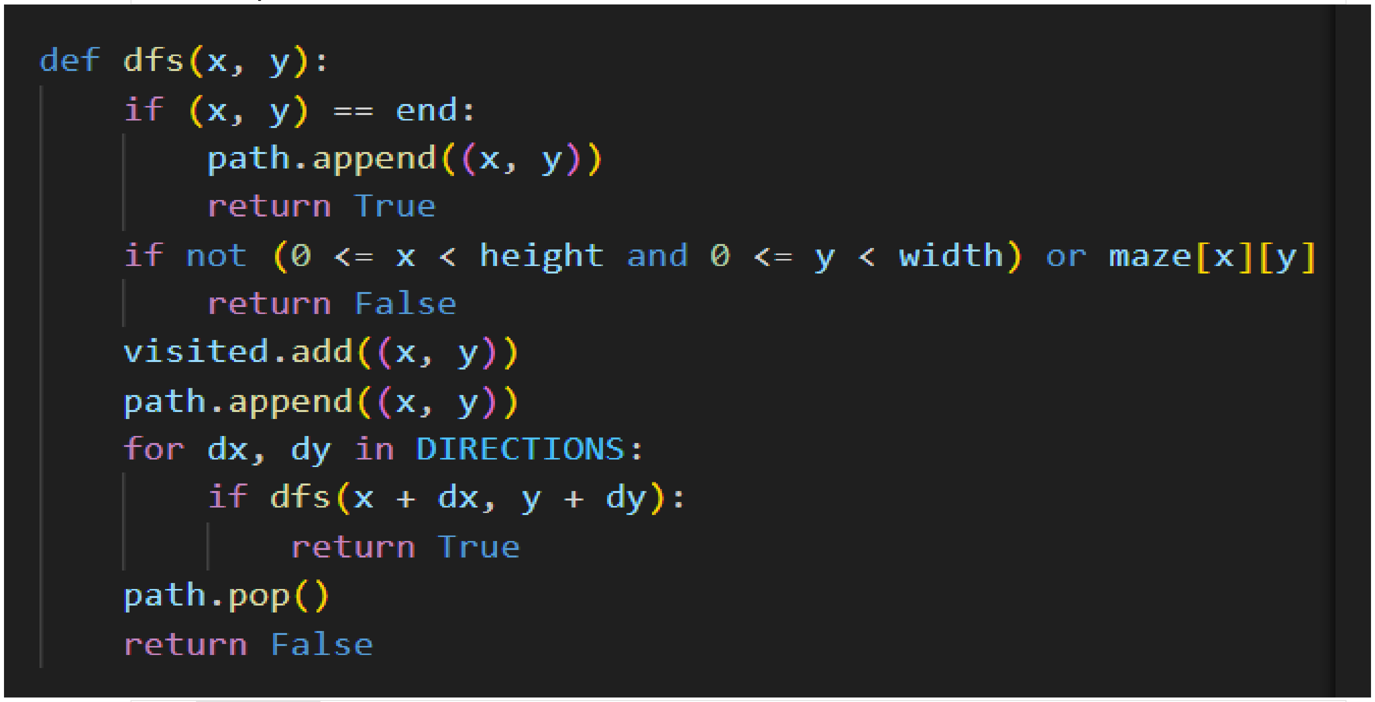
- **Input**: A 2D list representing the maze.
- **Output**: A list of coordinates representing the path from start to end.

python



```python
def solve_maze(maze):
    height = len(maze)
    width = len(maze[0])
    start = (1, 1)
    end = (height - 2, width - 2)
    path = []
    visited = set()
```

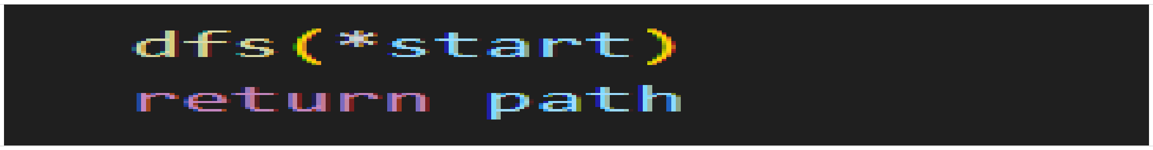
- **Initialization**: Sets up maze dimensions, start and end points, and initializes the path and visited cells.



```python
def dfs(x, y):
    if (x, y) == end:
        path.append((x, y))
        return True
    if not (0 <= x < height and 0 <= y < width) or maze[x][y]
        return False
    visited.add((x, y))
    path.append((x, y))
    for dx, dy in DIRECTIONS:
        if dfs(x + dx, y + dy):
            return True
    path.pop()
    return False
```

- **dfs(x, y)**: Recursively searches for a path to the end:
    - If the current cell is the end, appends it to the path and returns **True**.
    - Checks if the current cell is out of bounds, a wall, or already visited.
    - Marks the cell as visited and adds it to the path.
    - Recursively tries all directions; if a valid path is found, returns **True**.

If no valid path is found, backtracks by removing the cell from the path and returns **False**.



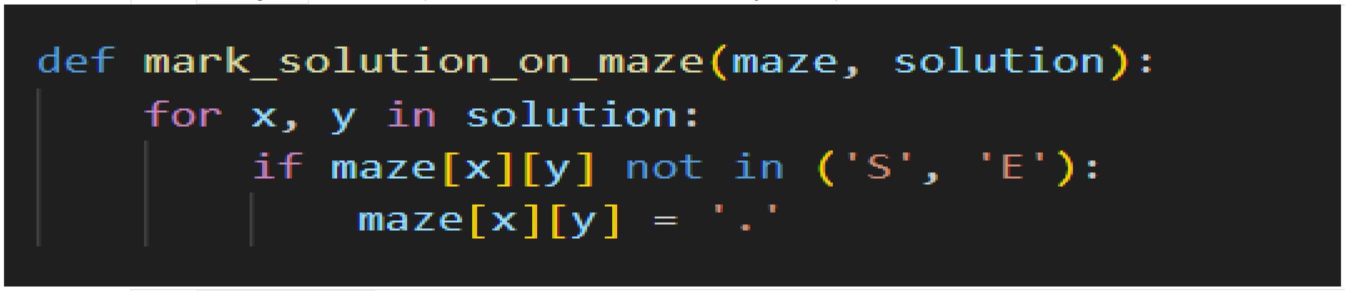
```
    dfs(*start)
    return path
```

- **Start DFS**: Begins the DFS from the start cell.
- **Return**: The found path.

4. mark_solution_on_maze(maze, solution)

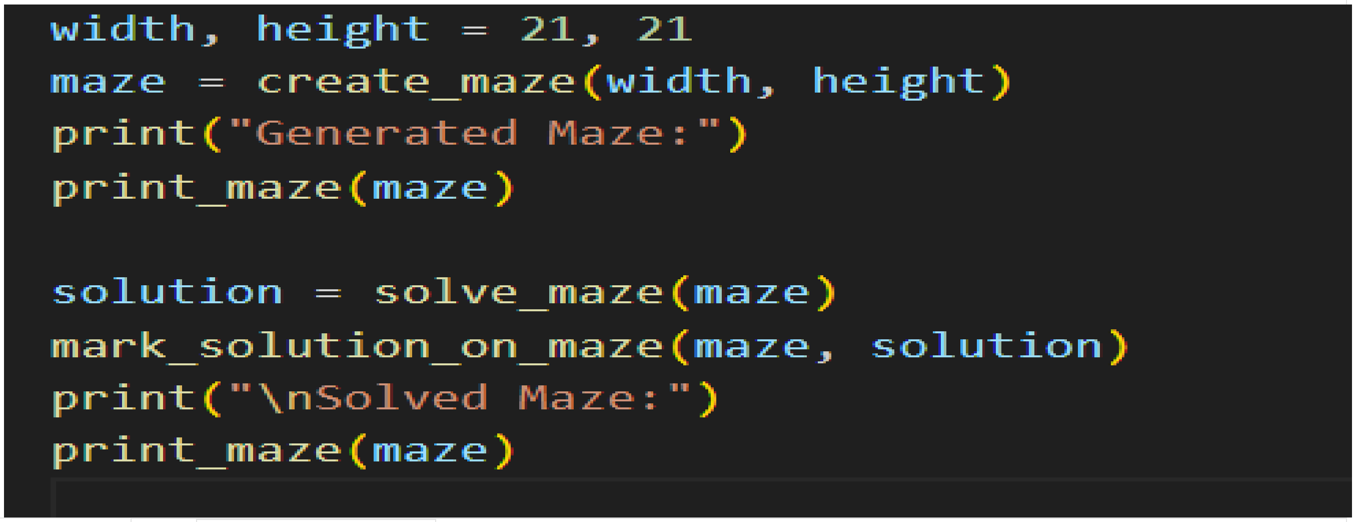
Mark the solution path on the maze.

- **Input**: A 2D list representing the maze and a list of coordinates representing the solution path.
- **Output**: None (modifies the maze in place)



```
def mark_solution_on_maze(maze, solution):
    for x, y in solution:
        if maze[x][y] not in ('S', 'E'):
            maze[x][y] = '.'
```

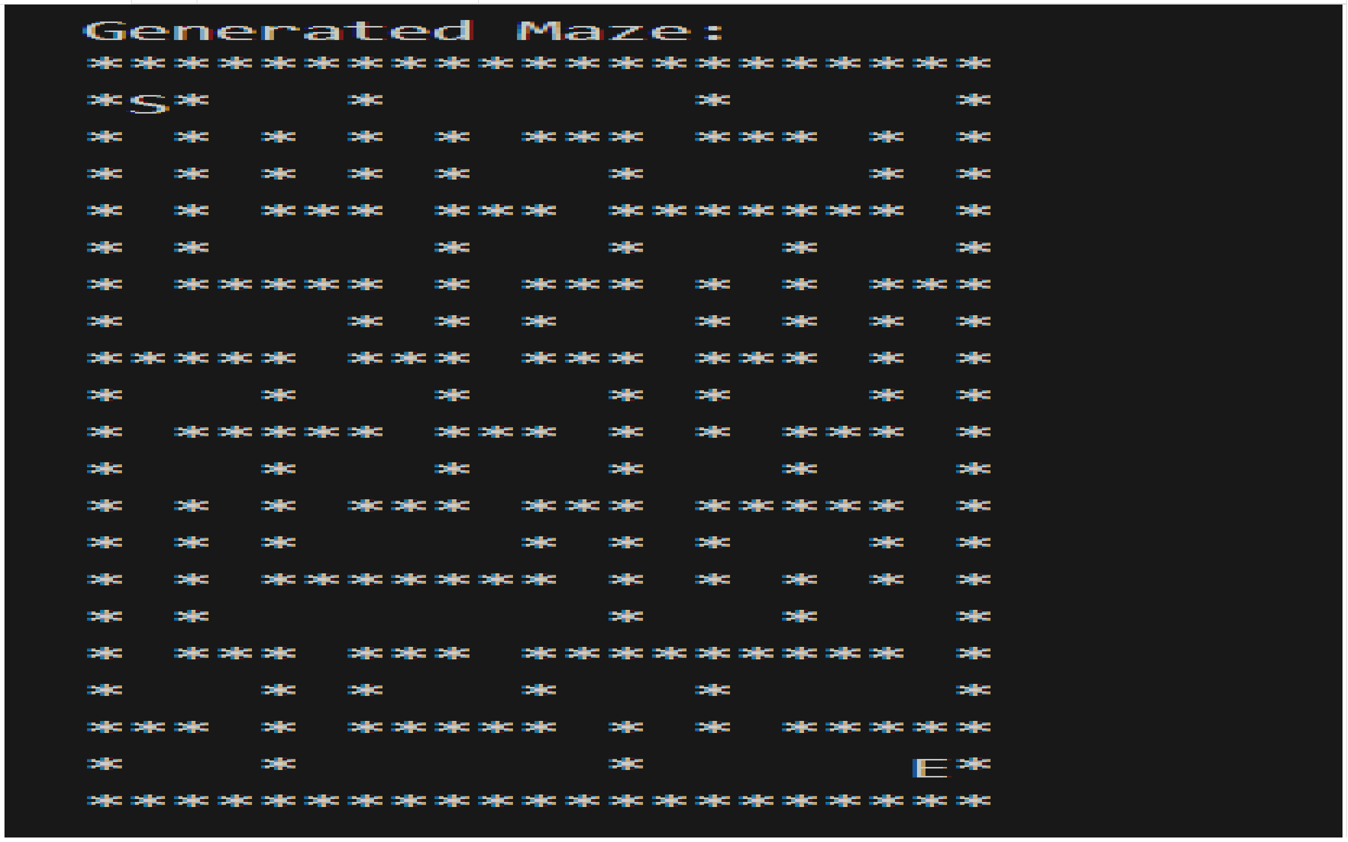
- **Marks Path**: For each coordinate in the solution, if it's not the start or end, marks it as part of the path (**'.'**).



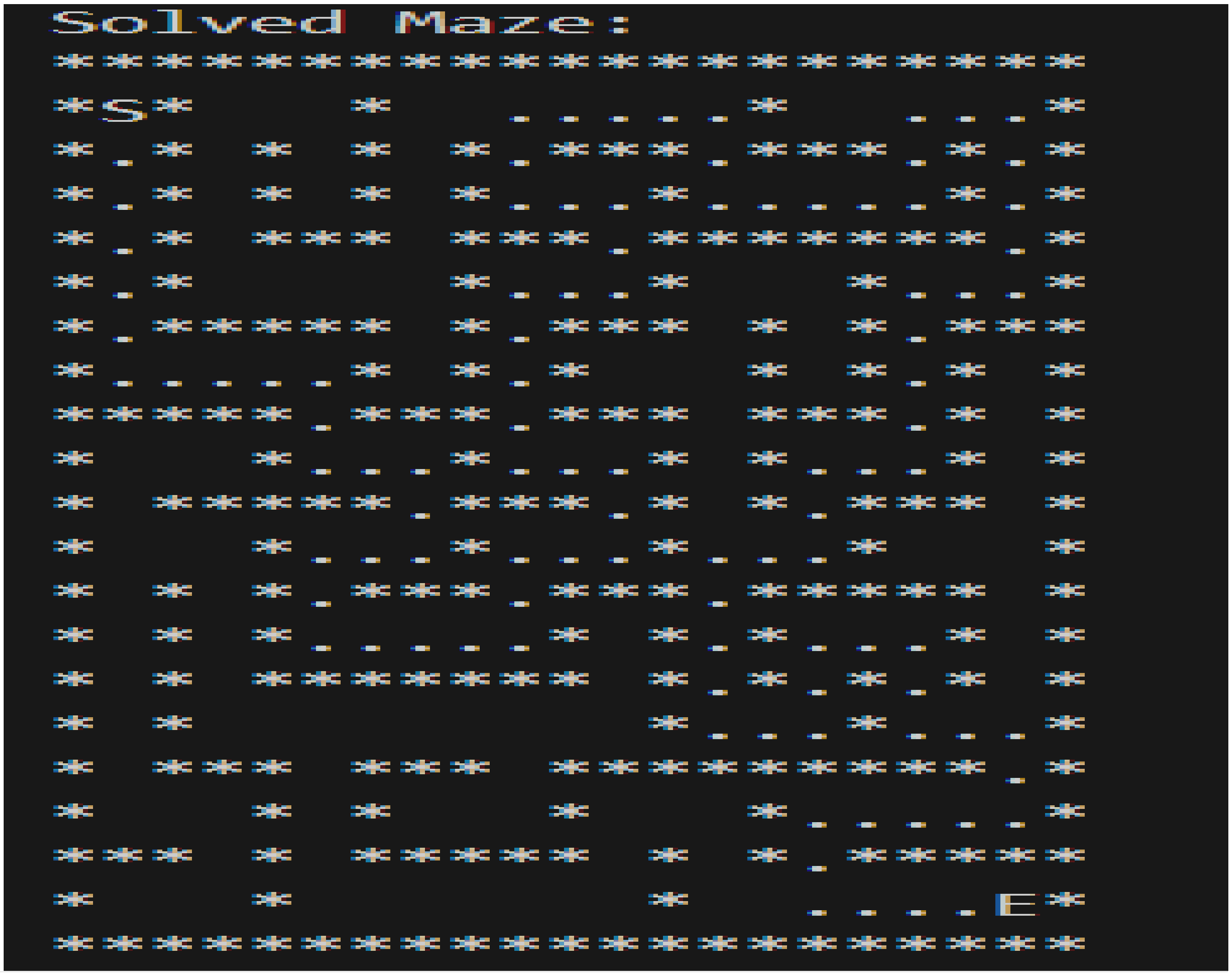
```
width, height = 21, 21
maze = create_maze(width, height)
print("Generated Maze:")
print_maze(maze)

solution = solve_maze(maze)
mark_solution_on_maze(maze, solution)
print("\nSolved Maze:")
print_maze(maze)
```

- **Generate Maze**: Creates a maze of size 21x21.
- **Print Maze**: Prints the generated maze.

- **Solve Maze**: Finds a path from the start to the end.
- **Mark Solution**: Marks the solution path on the maze.
- **Print Solved Maze**: Prints the maze with the solution path marked.



```
Generated Maze:
*********************
*S*   *       *     *
* * * * * *** *** * *
* * * * *   *     * *
* * *** *** ******* *
* *     *   *   *   *
* ***** * *** * * ***
*     * * *   * * * *
***** *** *** *** * *
*   *   *   * *   * *
* ***** *** * * *** *
*   *   *   *   *   *
* * * *** *** ***** *
* * *     * * *   * *
* * ******* * * * * *
* *         *   *   *
* *** *** ********* *
*   * *   *   *     *
*** * ***** * * *****
*   *       *      E*
*********************
```



```
Solved Maze:
* * * * * * * * * * * * * * * * * * * * *
* S *       *     . . . . . *     . . . *
* . *   *   *   * . * * * . * * * . * . *
* . *   *   *   * . . . * . . . . . * . *
* . *   * * *   * * * . * * * * * * * . *
* . *           * . . . *       * . . . *
* . * * * * *   * . * * *   *   * . * * *
* . . . . . *   * . *       *   * . *   *
* * * * * . * * * . * * *   * * * . *   *
*       * . . . * . . . *   * . . . *   *
*   * * * * * . * * * . *   * . * * *   *
*       * . . . * . . . * . . . *       *
*   *   * . * * * . * * * . * * * * *   *
*   *   * . . . . . *   * . * . . . *   *
*   *   * * * * * * *   * . * . * . *   *
*   *                   * . . . * . . . *
*   * * *   * * *   * * * * * * * * * . *
*       *   *       *       * . . . . . *
* * *   *   * * * * *   *   * . * * * * *
*       *               *     . . . . E *
* * * * * * * * * * * * * * * * * * * * *
```

References

[1]
"Maze-solving algorithm," *Wikipedia*, May 15, 2021. https://en.wikipedia.org/wiki/Maze-solving_algorithm

[2]
Wikipedia Contributors, "Maze generation algorithm," *Wikipedia*, Jan. 16, 2019. https://en.wikipedia.org/wiki/Maze_generation_algorithm

[3]
A. Blades, "Solving mazes with Depth-First Search," *The Startup*, Jun. 09, 2020. https://medium.com/swlh/solving-mazes-with-depth-first-search-e315771317ae

[4]
Wikipedia Contributors, "Maze," *Wikipedia*, Jan. 08, 2020. https://en.wikipedia.org/wiki/Maze