

|  |
| --- |
| **RAJALAKSHMI INSTITUTE OF TECHNOLOGY** |
| (An Autonomous Institution, Affiliated to Anna University, Chennai) |

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | **2117240070315** |
| **NAME** | **Subina Lakshmi M** |
| **PROJECT TITLE** | **Maze Solver using Depth-First Search Algorithm** |
| **DATE OF SUBMISSION** |  |
| **FACULTY IN-CHARGE** | **Mrs. S. Divya** |

**Signature of Faculty In-charge**

**INTRODUCTION**

Artificial Intelligence (AI) is one of the most transformative fields in computer science. It focuses on developing systems that can simulate human-like intelligence and problem-solving abilities. AI applications range from natural language processing and robotics to autonomous driving and pathfinding problems. Pathfinding, in particular, is a crucial subfield that involves determining the best or possible routes from a starting point to a destination in a given environment.

Maze solving is a classical AI problem that tests an algorithm's ability to explore, make decisions, and find a path within constraints. The maze can be visualized as a grid or graph, where certain paths are blocked by walls, and others are open for traversal. Various algorithms have been developed to handle such problems, including Depth-First Search (DFS), Breadth-First Search (BFS), Dijkstra’s Algorithm, and the A\* algorithm.

This project, titled “Maze Solver using Depth-First Search Algorithm,” implements a Python-based approach to solving a maze using DFS. The project demonstrates how the algorithm explores all possible paths systematically and uses backtracking to reach the goal node. The goal is not only to find a path but also to understand how DFS behaves in different maze configurations.

**PROBLEM STATEMENT**

The challenge addressed in this project is to create a program that can automatically find a valid path from the start point to the destination point in a maze represented as a two-dimensional matrix. Each cell is in the maze matrix can either be a wall or a path. The algorithm must navigate from the starting position to the goal position using only valid moves — typically up, down, left, and right — without revisiting any previously explored cell.

This problem is significant because maze navigation serves as a simplified version of real-world navigation problems encountered in robotics, gaming, and autonomous systems. By solving this problem efficiently using DFS, we can better understand the fundamentals of recursive exploration and backtracking, which form the basis for more complex AI pathfinding algorithms.

**GOAL**

The main objective of this project is to implement a functional maze solver using the Depth-First Search algorithm in Python. The specific goals include:

1. Representing a maze as a 2D matrix and defining start and goal positions.

2. Implementing the DFS algorithm to explore all possible paths within the maze.

3. Ensuring that the algorithm backtracks when it encounters dead ends.

4. Displaying the complete path found from the start to the destination.

5. Analyzing the performance of DFS in different maze configurations.

The project aims to demonstrate how DFS explores depth-first paths before considering alternate routes, highlighting its recursive nature and ability to solve complex traversal problems

**THEORETICAL BACKGROUND**

The Depth-First Search (DFS) algorithm is a well-known technique for traversing graphs and trees. It follows a simple principle: explore as far as possible along each branch before backtracking. DFS can be implemented either using a recursive approach or an explicit stack data structure.

In maze solving, DFS begins at the starting point and recursively explores adjacent unvisited cells until it either reaches the goal or encounters a dead end. When a dead end is reached, the algorithm backtracks to the previous cell and continues exploring other available paths. This makes DFS a perfect example of a backtracking algorithm.

Mathematically, a maze can be represented as a graph where each cell represents a node and valid moves represent edges. DFS explores these nodes systematically. While DFS is not guaranteed to find the shortest path, it is often faster in small mazes and is easier to implement compared to algorithms like A\* or Dijkstra’s. These alternatives, although more efficient for shortest-path problems, involve additional overhead for maintaining distance or heuristic values.

Justification for choosing DFS: - It is simple to implement recursively.

- It efficiently explores complex mazes without requiring additional data structures.

- It provides an excellent introduction to recursive and backtracking algorithms.

- It can be easily visualized and extended to larger problems.

**ALGORITHM EXPLANATION WITH EXAMPLE**

The DFS algorithm follows the following steps for maze solving:

1. Represent the maze as a 2D grid of 0s (open paths) and 1s (walls).

2. Select a starting point (for example, the top-left corner).

3. Mark the current cell as visited to prevent revisiting.

4. Move to an adjacent cell (up, down, left, or right) that is open and unvisited.

5. If the destination is reached, record the path.

6. If no valid move is possible, backtrack to the previous cell.

7. Continue the process until all paths have been explored or the goal is reached.

Example Maze:

[

[0, 1, 0, 0],

[0, 0, 0, 1],

[1, 0, 1, 0],

[0, 0, 0, 0]

]

Start at (0,0) and try to reach (3,3). DFS will explore deeply in one direction, marking each visited cell, until it either reaches the destination or has to backtrack due to walls.

**IMPLEMENTATION AND CODE**

import tkinter as tk

from tkinter import messagebox

import random, time

ROWS, COLS = 10, 10

CELL\_SIZE = 40

maze = []

start = (0, 0)

end = (ROWS - 1, COLS - 1)

speed = 0.1

root = tk.Tk()

root.title("AI Maze Solver using Depth-First Search (DFS)")

root.config(bg="#1e1e1e")

title\_label = tk.Label(root, text="Maze Solver using Depth-First Search (DFS)",

font=("Arial", 18, "bold"), bg="#1e1e1e", fg="white")

title\_label.pack(pady=10)

canvas = tk.Canvas(root, width=COLS \* CELL\_SIZE, height=ROWS \* CELL\_SIZE, bg="white", highlightthickness=0)

canvas.pack(pady=10)

def generate\_maze():

global maze

maze = [[random.choice([0, 1, 0, 0]) for \_ in range(COLS)] for \_ in range(ROWS)]

maze[start[0]][start[1]] = 0

maze[end[0]][end[1]] = 0

draw\_maze()

def draw\_maze():

canvas.delete("all")

for i in range(ROWS):

for j in range(COLS):

color = "white" if maze[i][j] == 0 else "#2c2c2c"

canvas.create\_rectangle(j \* CELL\_SIZE, i \* CELL\_SIZE,

(j + 1) \* CELL\_SIZE, (i + 1) \* CELL\_SIZE,

fill=color, outline="#3e3e3e")

draw\_cell(start[0], start[1], "#00bfff")

draw\_cell(end[0], end[1], "#32cd32")

def draw\_cell(i, j, color):

canvas.create\_rectangle(j \* CELL\_SIZE, i \* CELL\_SIZE,

(j + 1) \* CELL\_SIZE, (i + 1) \* CELL\_SIZE,

fill=color, outline="#3e3e3e")

root.update()

time.sleep(speed)

path = []

def is\_valid(x, y):

return 0 <= x < ROWS and 0 <= y < COLS and maze[x][y] == 0

def dfs(x, y):

if (x, y) == end:

path.append((x, y))

draw\_cell(x, y, "#32cd32")

return True

if not is\_valid(x, y):

return False

maze[x][y] = 2

draw\_cell(x, y, "#f4d03f")

path.append((x, y))

for dx, dy in [(1, 0), (0, 1), (-1, 0), (0, -1)]:

if dfs(x + dx, y + dy):

return True

draw\_cell(x, y, "#e74c3c")

path.pop()

return False

def solve\_maze():

global path

path.clear()

draw\_maze()

for i in range(ROWS):

for j in range(COLS):

if maze[i][j] == 2:

maze[i][j] = 0

result = dfs(start[0], start[1])

if result:

messagebox.showinfo("Maze Solver", " Path Found Successfully!")

else:

messagebox.showwarning("Maze Solver", " No Path Found!")

draw\_maze()

def change\_speed(val):

global speed

speed = float(val)

speed\_label = tk.Label(root, text="Visualization Speed", font=("Arial", 11), bg="#1e1e1e", fg="white")

speed\_label.pack(pady=(5, 0))

speed\_slider = tk.Scale(root, from\_=0.01, to=0.5, resolution=0.01, orient="horizontal",

length=200, command=change\_speed, bg="#1e1e1e", fg="white", troughcolor="#2e2e2e")

speed\_slider.set(speed)

speed\_slider.pack(pady=5)

button\_frame = tk.Frame(root, bg="#1e1e1e")

button\_frame.pack(pady=10)

tk.Button(button\_frame, text="Generate Maze", command=generate\_maze,

font=("Arial", 12), bg="#0078d7", fg="white", width=14).grid(row=0, column=0, padx=10)

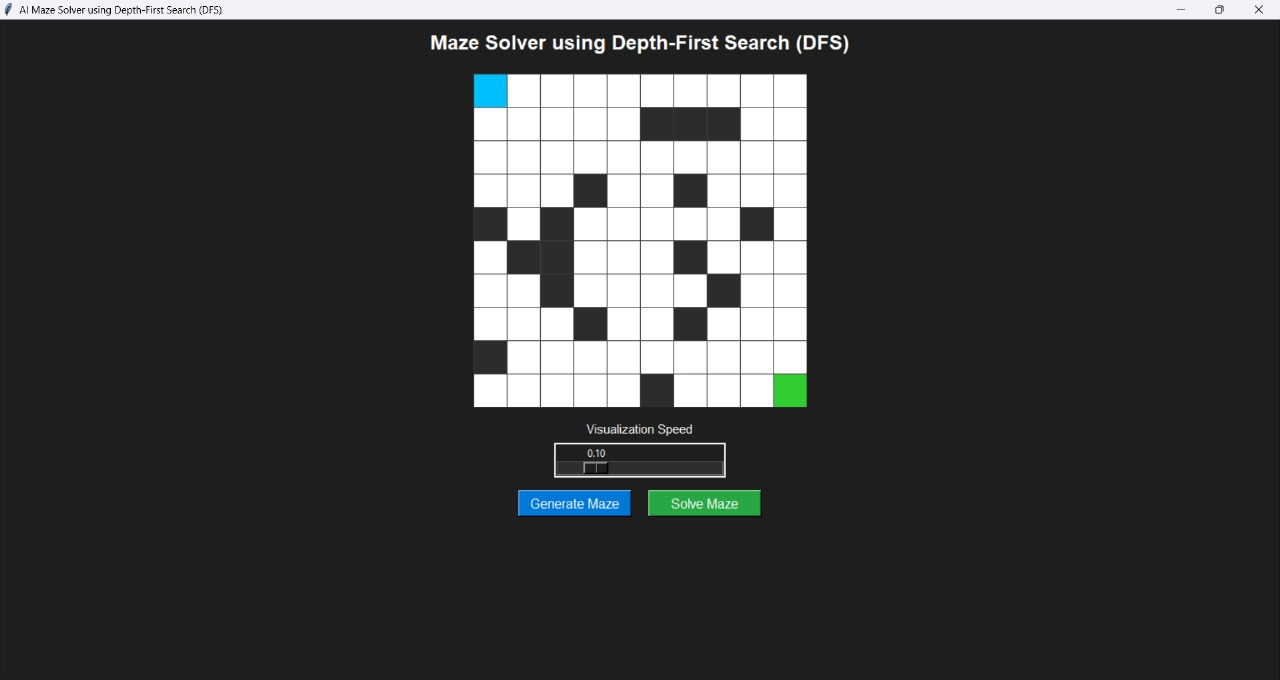
tk.Button(button\_frame, text="Solve Maze", command=solve\_maze,

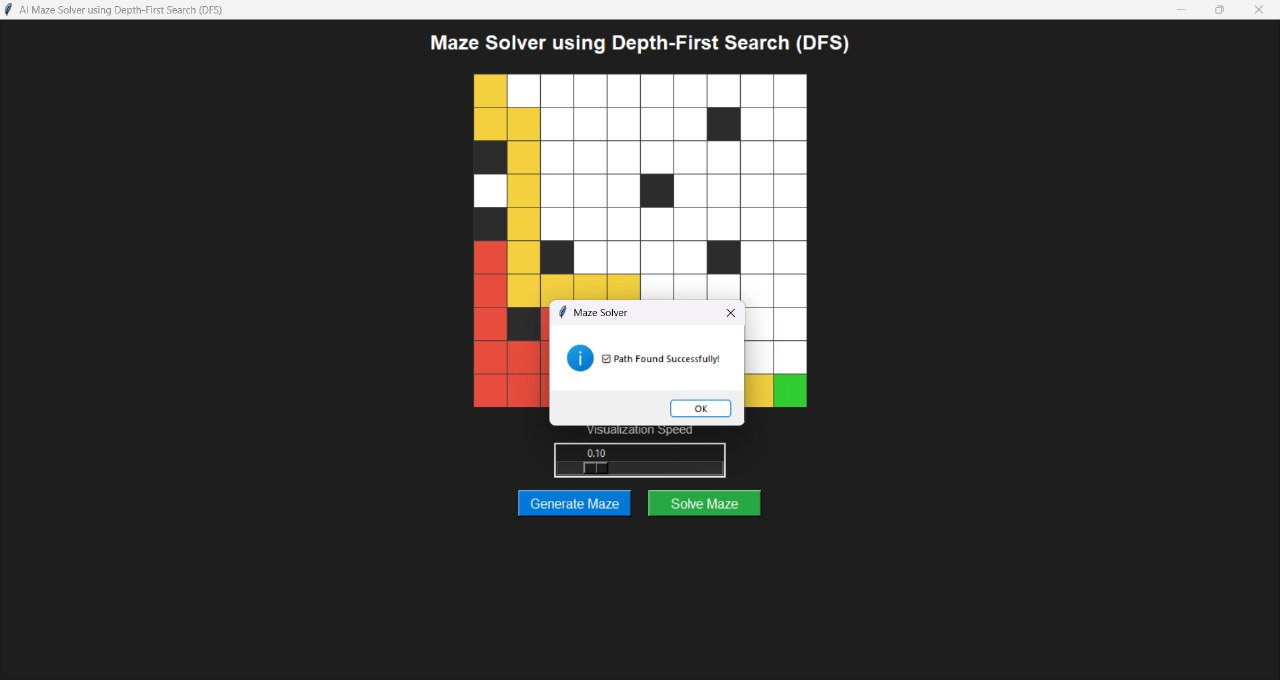
font=("Arial", 12), bg="#28a745", fg="white", width=14).grid(row=0, column=1, padx=10)

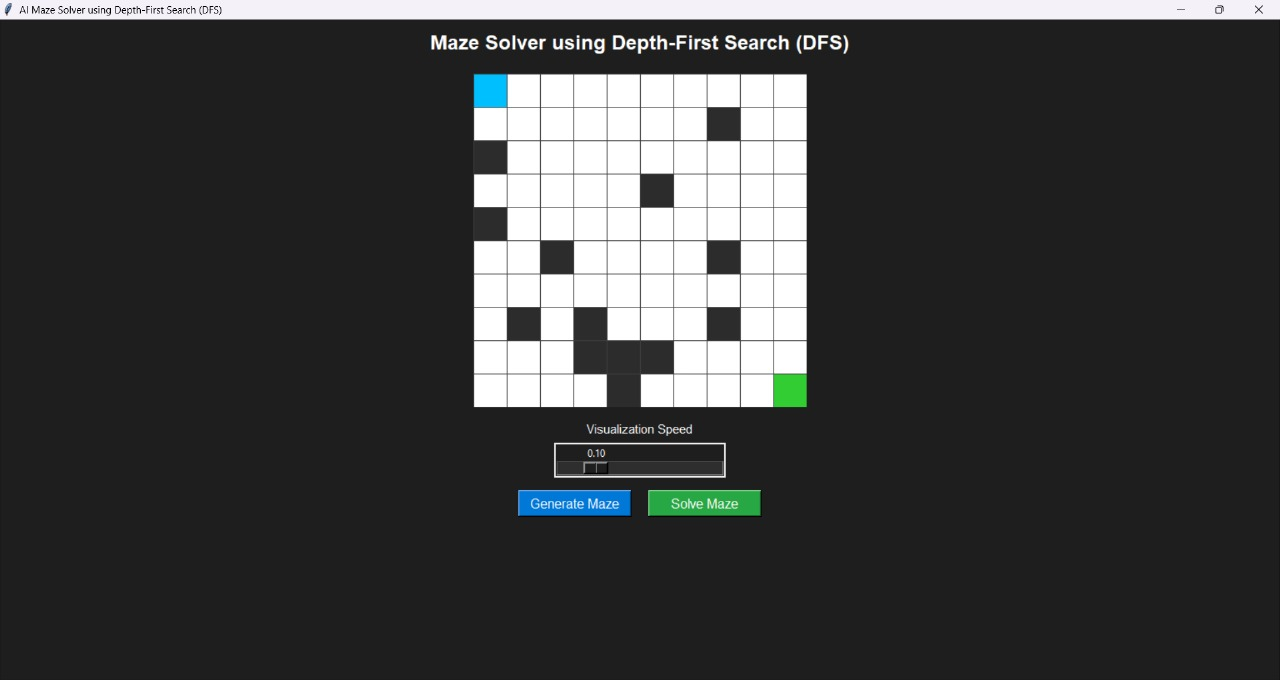
generate\_maze()

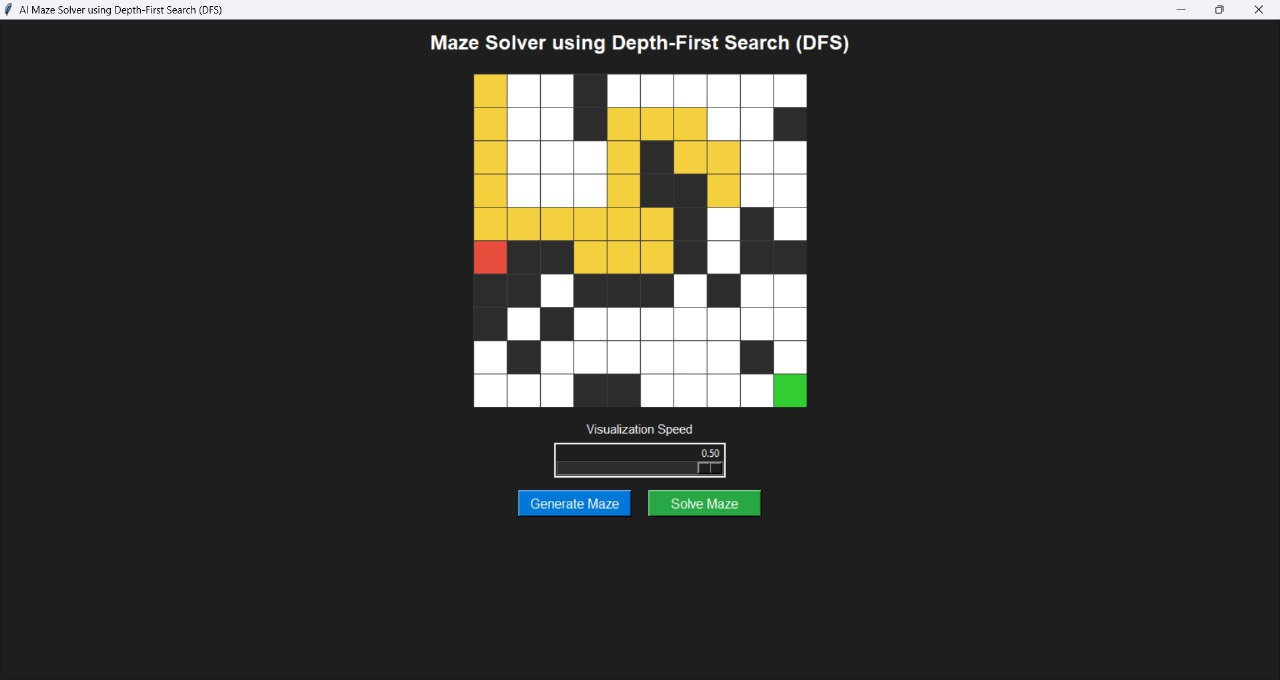
root.mainloop()

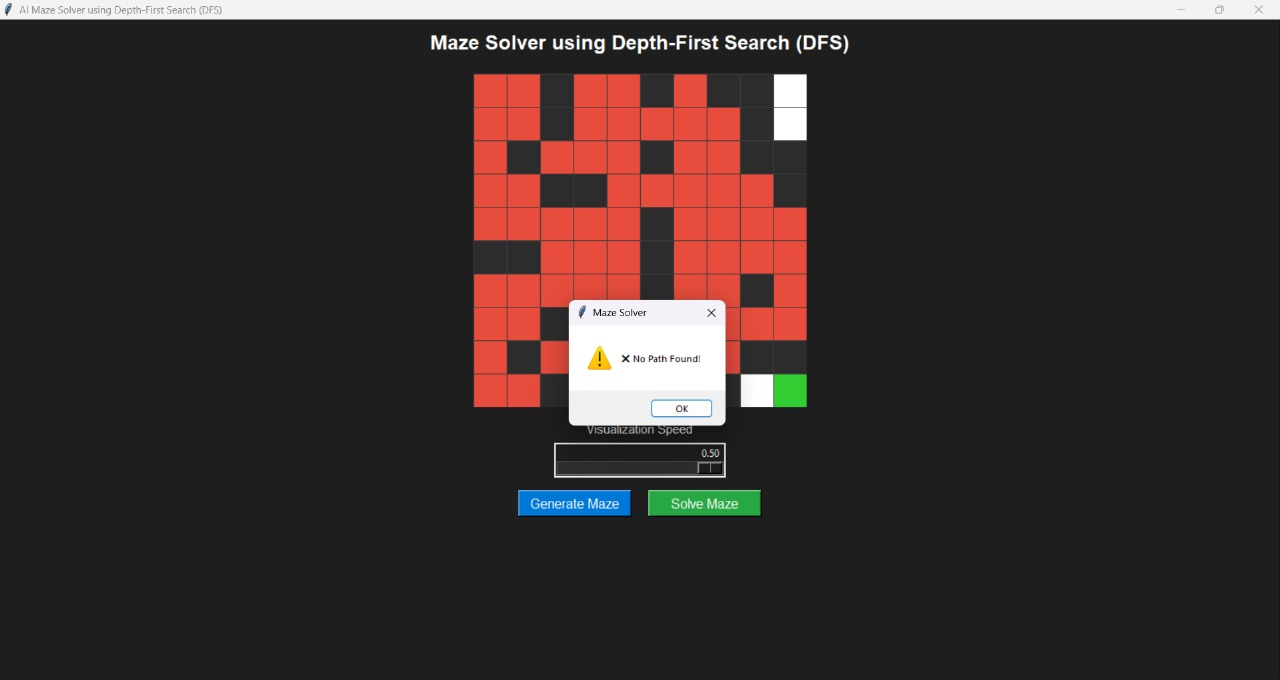
**OUTPUT**











**RESULTS AND FUTURE ENHANCEMENT**

The Maze Solver using Depth-First Search successfully demonstrates the application of a recursive pathfinding algorithm. The approach works well for small and medium-sized mazes, efficiently finding a valid path from start to goal.However, DFS is not always optimal because it may explore unnecessary paths before finding the solution. Despite this limitation, it provides a strong foundation for understanding traversal and search techniques used in Artificial Intelligence.

Future enhancements include:

- Integrating graphical visualization using libraries such as Pygame or Tkinter.

- Implementing additional algorithms like BFS, A\*, or Dijkstra’s for comparison.

- Optimizing recursive calls to handle large mazes without stack overflow.

- Allowing user input for maze generation and difficulty levels.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | **https://github.com/subina0987/Maze-Solver-DFS** |

**Literature survey**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.no** | **Author / Source** | **Title / Publication** | **Description / Contribution** |
| 1 | Stuart Russell & Peter Norvig | Artificial Intelligence: A Modern Approach | Provides foundational knowledge of AI search algorithms, including Minimax and Constraint Satisfaction, which are essential for intelligent game |
| 2 | Thomas H. Cormen et al | Introduction to Algorithms | Explains algorithm design principles, recursion, and optimization strategies that support efficient implementation of game-solving logic. |
| 3 | GeeksforGeeks | Depth-First Search Algorithm in Python | Demonstrates the use of recursive search algorithms in pathfinding and decision-making, forming the basis of Minimax exploration in game trees. |
| 4 | Python Official Documentation Python | Python.org | Offers official guidance on Python programming, libraries (Tkinter), and syntax essential for building the project’s GUI and logic. |
| 5 | Towards Data Science | Exploring Pathfinding Algorithms using DFS and BFS | Discusses AI-based pathfinding techniques that inspired the search approach used in the Minimax decision process. |

**REFERENCES**

1. Stuart Russell & Peter Norvig, "Artificial Intelligence: A Modern Approach"

2. Thomas H. Cormen et al., "Introduction to Algorithms"

3. GeeksforGeeks: "Depth-First Search Algorithm in Python"

4. Python Official Documentation: https://docs.python.org/

5. Towards Data Science: "Exploring Pathfinding Algorithms using DFS and BFS"