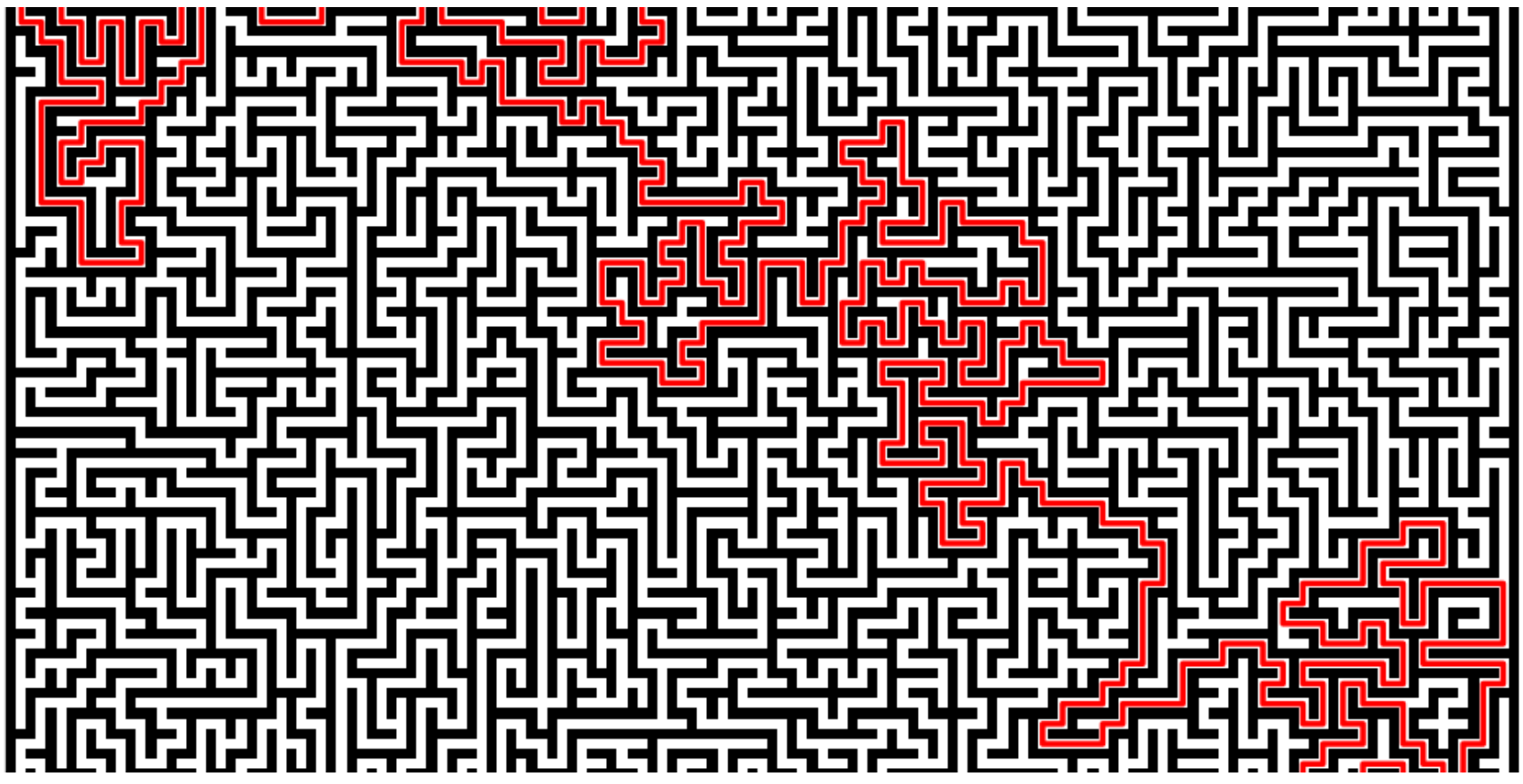

DSA project

1. Detailed Project Description

Our project is a Maze Solver Visualizer that demonstrates how different search algorithms solve mazes using various data structures. The maze is represented as a grid & is loaded from a .txt file consisting of characters that define the layout (# for walls, spaces for walkable paths, etc.).

Visualization:

```
#####A####################################
#  #   #    #   #   #    ##
# ### ### ### ### ##### ### ### ##### # #
#  #   #   #   ##  ##   #  #
# # ##### # ### ##### # # # # ##### # # #
##   ##  ##  #  #### #   ###
# ###### #  # # ##### # # # ##### # #
#   ## #  ##   # #  #   ##
##### ### # ##### ### ######### ### # # #
#  #   #    #   #  #  #  #  #
# # ############# # ### # # # ##### ### #
##    #    ##  ### #   #   #
# ######### ##### # # ### # ##### ### ###
#    # #   ## #  #  #   ## #
##### # # ### ### ### ### ### # # ### # #
##  #   ##   #  #   #  ####
# # ######### # ##### ##### ####### # # #
##      #   #   #     #  ##
#######################################B##
```

The core idea is to apply different search algorithms (such as Depth-First Search (DFS) and Breadth-First Search (BFS)) to solve the maze while visualizing the step-by-step execution of each algorithm in real-time. The user chooses the algorithm through a simple user interface and then sees how it explores the maze until it reaches the goal.

To add creativity and complexity, we include special tiles in the maze with unique behaviors (e.g., teleportation, step penalties, etc.), which influence the algorithm’s behavior dynamically.

The project demonstrates the practical application of structures like stacks, queues, and 2D arrays, and optionally others like priority queues or hash sets.

Some Examples for these Algorithms 👇🏽

Algorithm	Path Quality	Speed & Memory	Analogy
DFS	Any path (not shortest)	Fast, low memory	Deep branch explorer (go deep first)
BFS	Guaranteed shortest	Moderate memory	Ripple expanding wave (layer by layer)
A*	Fastest shortest path	Efficient (heuristic)	GPS with smart guessing (best guess ahead)
Dijkstra	Shortest (weighted)	Slower than A*	Toll cost road finder (weighted roads)
Left & Right-Hand Rule	May not be shortest	Very low memory	Human wall-hugging navigator
Dead-End Fill	Optimal in perfect mazes	Moderate	Erasing all dead ends till solution remains

2. Data Structures Used

2D Array	To store the maze structure after reading it from the .txt file. Each cell holds the value of the tile (wall, space, start, end, special tile).
Stack	Used for implementing DFS (Depth-First Search). The algorithm uses LIFO behavior to explore paths deeply before backtracking.
Queue	Used for implementing BFS (Breadth-First Search). The algorithm uses FIFO behavior to explore all neighbors at the same depth before moving deeper.
Set / Hash Table	To keep track of visited positions efficiently and avoid reprocessing the same cell multiple times.
Priority Queue *(optional)*	Could be used if we implement more advanced algorithms like A* or Dijkstra's algorithm.
Graph Representation	The maze is treated as a graph where each walkable tile is a node connected to its adjacent walkable tiles.

3. Use Case or Scenario

Scenario: Visual Learning Tool for Algorithms 📖

In an educational setting, a student wants to learn how different search algorithms work and how they perform under different conditions. Instead of reading dry textbook examples, they use our application to load a maze and visually see how algorithms like DFS and BFS behave step-by-step.

4. Main Operations Implemented

Search (DFS, BFS, etc..)	Core logic of the project. Finds a path from start to goal. Each algorithm uses a different structure and traversal logic.
Traversal	Moves through the maze grid, checking valid neighbors and visiting unvisited cells.
Insert / Remove	Applies to stack or queue operations during search—pushing/popping nodes during traversal.
Update	Updates the visualization state of the maze (coloring visited tiles, marking the path).
Step Counting	Measures how many steps each algorithm takes to reach the goal.
Handle Special Tiles	Modifies algorithm behavior dynamically (e.g., resets stack on teleportation, adds penalties).

5. User Interaction

Our project includes a simple graphical user interface (GUI) that allows users to interact with the program:

- **Load Maze Button**: Lets the user load a .txt file containing the maze.
- **Algorithm Selection Dropdown**: Allows the user to choose between BFS, DFS, or other supported algorithms.
- **Start Button**: Begins the visualization process.
- **Real-Time Visualization**: The algorithm runs with a small delay (e.g., 200ms per step) to show how it explores the maze.
- **Result Display**: Once the maze is solved, the path taken is highlighted and the number of steps is shown.

6. Challenges or Uncertainties

- **Teleportation Logic**: When the algorithm hits a teleportation tile, resetting the stack or queue while maintaining correctness.
- **Advanced Algorithm Integration**: If we choose to add A* or Dijkstra's algorithm, managing heuristics and data structures like priority queues could add complexity.
- **Performance**: Keeping the real-time delay and UI smooth while processing large mazes or running multiple algorithms may require optimization.
- **Maze File Validation**: Ensuring that maze files are formatted correctly (have a start and end, no unreachable zones, etc...) needs error handling.
- **Balancing Creativity and Clarity**: Adding too many features (like special tiles) might distract from the educational focus if not designed carefully.

Team Members:

1. **Kareem Mohammed Ahmed Taha**
2. **Bassel Ahmed Mohammed Abdelmonem**
3. **Alaa Essam AbdelAzeem Ibrahim**
4. **Ganna Ahmed Gomaa Ahmed**
5. **Amat-ElRahman Sayed Mohammed**

Also Inspired By:

https://youtu.be/4L7BDRmH4cM?si=6D0Clg3Ho87htJbO