MATLAB Project

Maze Solver (Introduction)

Objective: Create a program that can solve a maze. You can represent the maze as a matrix where 1s represent walls and 0s represent paths, and use algorithms like BFS (breadth-first search) to find the shortest path.

Skills Learned: Algorithm design, matrix manipulation, and pathfinding.

Additional Features: Allow the user to input their own maze or visualize the maze-solving process with graphical representations.

The goal of this project is to create a program that solves a maze using the **Breadth-First Search (BFS)** algorithm. The maze is represented as a matrix where:

- **1s** represent walls (impassable areas),
- **0s** represent open paths (where movement is possible).

Key Features:

1. **Input Maze**: The program allows the user to input their own maze in matrix form.
2. **Algorithm**: It uses BFS, which explores all possible paths level by level to find the shortest path from the start point to the end point.
3. **Visualization**: The program can display the maze and the pathfinding process step-by-step, showing how the algorithm navigates the maze.

Learning Outcomes:

- Understand algorithm design (specifically BFS).
- Gain skills in matrix manipulation.
- Learn how to implement pathfinding in a grid-based environment.

Maze Solver using MATLAB: Project Explanation

This project involves building a **Maze Solver** using the **Breadth-First Search (BFS)** algorithm in MATLAB. The solver aims to find the shortest path in a maze represented by a matrix of 1s and 0s, where:

- **1s** represent walls (obstacles),
- **0s** represent open paths (walkable areas).

Here's a detailed explanation of the project:

1. Maze Representation

- **Matrix Format**: The maze is represented as a 2D matrix in MATLAB, where:
    - **1s** represent **walls**, which are impassable areas.
    - **0s** represent **open paths**, where movement is allowed.

2. Breadth-First Search (BFS) Algorithm

- **BFS Overview**: BFS is an algorithm used to explore all possible paths in a graph or grid by visiting nodes level by level. It ensures that the shortest path is found by expanding all nodes of a given distance before moving on to greater distances.
- **How BFS Works in the Maze**:
    - The algorithm starts at the **start point**.
    - It explores all the adjacent **valid paths** (cells with 0s) in a **queue-based manner**.
    - Each cell is marked as visited to avoid revisiting.
    - Once the **end point** is reached, the algorithm traces the path from the end point back to the start using parent pointers or a path array, ensuring the shortest path is found.
- **Steps**:

1. **Initialize**: Create a queue and enqueue the start point. Mark it as visited.
2. **Explore Neighbors**: While there are cells in the queue:
    - Dequeue the current cell.
    - Explore its neighbors (up, down, left, right).
    - If a neighbor is within bounds, a valid path (0), and not visited yet, enqueue it and mark it as visited.
    - If the end point is found, the path is reconstructed.
3. **Backtrack to Find Path**: Once the destination is reached, backtrack from the end point to the start point, reconstructing the shortest path.

3. User Input for Maze

- **Input Format**: The user can input their own maze as a matrix, either manually or by loading a predefined file.
- **Start and End Points**: The user can define where the start and end points are located in the maze.

- The program can also validate if the maze has a valid start and end point, and if a path exists.

4. Path Visualization

- **Step-by-Step Visualization**: The algorithm can be set to visualize its progress by showing the maze and highlighting cells as they are visited.
- **Display**: MATLAB can use **imagesc** or **imshow** to display the maze and color-code the visited nodes, the path, and the walls. This helps users understand how the algorithm explores the maze and finds the solution.

Example:

  - Walls (1s) could be shown in **black**,
  - Open paths (0s) in **white**,
  - The explored path can be displayed in **red**.

5. Additional Features

- **Error Handling**: If the maze has no solution (no path from start to end), the program can notify the user that the maze is unsolvable.
- **Multiple Mazes**: The user can input and solve multiple mazes without restarting the program.
- **GUI (Optional)**: For advanced users, a GUI could be implemented to allow more interactive maze creation and visualization.

6. Learning Outcomes

- **Algorithm Design**: Users will learn how BFS works and how it guarantees the shortest path in an unweighted grid.
- **Matrix Manipulation**: The project involves a lot of working with 2D arrays (matrices), which is an essential MATLAB skill.
- **Pathfinding**: This project reinforces concepts of pathfinding algorithms, which are widely used in games, robotics, and AI.

7. Conclusion

This project serves as an excellent introduction to algorithm design, matrix manipulation, and pathfinding. By solving a maze using BFS in MATLAB, users gain hands-on experience with key programming and problem-solving techniques in a fun and interactive way.

CODE :-

```
% Maze Solver Using Breadth-First Search (BFS)
% The maze is represented as a binary matrix (1's are walls, 0's are paths)

function mazeSolver()
  % Prompt the user to input the maze or use a sample maze
  prompt = 'Would you like to input your own maze? (y/n): ';
  user_input = input(prompt, 's');

  if user_input == 'y' || user_input == 'Y'
    maze = input('Enter the maze as a matrix (e.g., [0 1 0; 0 1 0; 0 0 0]): ');
  else
    % Default maze for testing
    maze = [
      1 1 1 1 1 1;
      1 0 0 0 1 1;
      1 0 1 0 1 1;
      1 0 1 0 0 1;
      1 0 1 1 0 1;
      1 1 1 1 0 0;
    ];
  end

  % Specify start and end points
  start = [2, 2];  % Example start point (row, column)
  goal = [6, 6];   % Example goal point (row, column)

  % Solve the maze using BFS
  [path, visited] = bfs(maze, start, goal);

  if ~isempty(path)
    fprintf('Path found: \n');
```

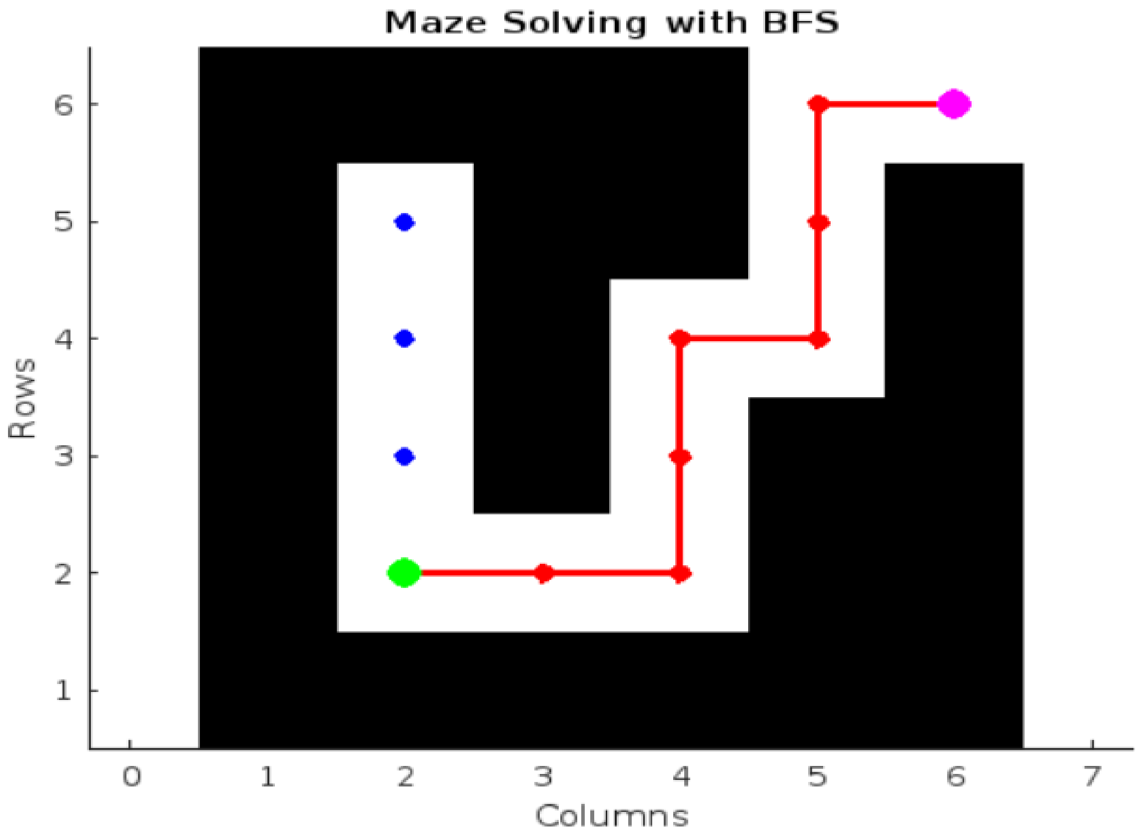
```
        disp(path);
        % Visualize the maze and the path
        visualize_maze(maze, visited, path, start, goal);  % Pass start and goal
    else
        fprintf('No path found.\n');
    end
end
% Maze Solver Using Breadth-First Search (BFS)
% The maze is represented as a binary matrix (1's are walls, 0's are paths)

function maze_solver()

    % Prompt the user to input the maze or use a sample maze
    prompt = 'Would you like to input your own maze? (y/n): ';
    user_input = input(prompt, 's');

    if user_input == 'y' || user_input == 'Y'
        maze = input('Enter the maze as a matrix (e.g., [0 1 0; 0 1 0; 0 0 0]): ');
    else
        % Default maze for testing
        maze = [
            1 1 1 1 1 1;
            1 0 0 0 1 1;
            1 0 1 0 1 1;
            1 0 1 0 0 1;
            1 0 1 1 0 1;
            1 1 1 1 0 0;
        ];
    end

    % Specify start and end points
    start = [2, 2];  % Example start point (row, column)
    goal = [6, 6];   % Example goal point (row, column)
```

```
    % Solve the maze using BFS
    [path, visited] = bfs(maze, start, goal);

    if ~isempty(path)
        fprintf('Path found: \n');
        disp(path);
        % Visualize the maze and the path
        visualize_maze(maze, visited, path, start, goal);  % Pass start and goal
    else
        fprintf('No path found.\n');
    end
end

function [path, visited] = bfs(maze, start, goal)
    % Get the size of the maze
    [rows, cols] = size(maze);

    % Directions for movement (up, down, left, right)
    directions = [0, 1; 0, -1; 1, 0; -1, 0];

    % Initialize the queue for BFS (stores [row, col] coordinates)
    queue = start;

    % Create a visited matrix to mark visited cells
    visited = zeros(rows, cols);
    visited(start(1), start(2)) = 1;

    % Create a parent matrix to trace the path
    parent = zeros(rows, cols, 2);

    % Perform BFS
    while ~isempty(queue)
        % Get current position
        current = queue(1, :);
```

```
        queue(1, :) = [];

        % Check if we've reached the goal
        if all(current == goal)
            path = reconstruct_path(parent, start, goal);
            return;
        end

        % Explore the neighbors (up, down, left, right)
        for i = 1:4
            neighbor = current + directions(i, :);
            r = neighbor(1);
            c = neighbor(2);

            % Check if the neighbor is within bounds and is a valid move
            if r > 0 && r <= rows && c > 0 && c <= cols && maze(r, c) == 0 && visited(r, c) == 0
                visited(r, c) = 1;  % Mark as visited
                queue = [queue; r, c];  % Add to the queue
                parent(r, c, :) = current;  % Set the parent for path reconstruction
            end
        end
    end

    % If no path is found
    path = [];
end

function path = reconstruct_path(parent, start, goal)
    % Reconstruct the path from goal to start using the parent matrix
    path = goal;
    current = goal;

    while ~all(current == start)
        current = squeeze(parent(current(1), current(2), :))';
```

```
        path = [current; path];
    end
end

function visualize_maze(maze, visited, path, start, goal)
    % Visualize the maze and the pathfinding process

    figure;
    hold on;

    % Plot the maze (walls as black and paths as white)
    imagesc(maze);
    colormap([1 1 1; 0 0 0]);  % White for path, Black for walls
    axis equal;

    % Plot the visited cells (in blue)
    [visited_rows, visited_cols] = find(visited == 1);
    plot(visited_cols, visited_rows, 'bo', 'MarkerFaceColor', 'b');

    % Plot the final path (in red)
    if ~isempty(path)
        [path_rows, path_cols] = deal(path(:, 1), path(:, 2));
        plot(path_cols, path_rows, 'ro-', 'MarkerFaceColor', 'r', 'LineWidth', 2);
    end

    % Mark the start and goal positions
    plot(start(2), start(1), 'go', 'MarkerFaceColor', 'g', 'MarkerSize', 10);  % Green for start
    plot(goal(2), goal(1), 'mo', 'MarkerFaceColor', 'm', 'MarkerSize', 10);    % Magenta for goal

    title('Maze Solving with BFS');
    xlabel('Columns');
    ylabel('Rows');
    hold off;
end
```

Output :-



Sample-Input :--

1. **4x4 Maze with Obstacles (Path Exists)**

maze =

[

0, 1, 0, 0;

0, 1, 0, 1;

0, 0, 0, 1;

0, 0, 0, 0

];

% Starting position (top-left corner) start = [1, 1];

% Goal position (bottom-right corner) goal = [4, 4];

2. **<u>5x5 Maze with Multiple Paths (Path Exists)</u>**
   <u>maze</u> =

```
 [
   0, 1, 0, 0, 0;
   0, 1, 0, 1, 0;
   0, 1, 0, 0, 0;
   0, 0, 0, 1, 0;
   1, 1, 0, 0, 0
];
```

   start = [1, 1]; % Starting position (top-left)
   goal = [5, 5];   % Goal position (bottom-right)

3. **<u>5x5 Maze with No Path (Blocked)</u>**
   <u>maze =</u>

```
[
   0, 1, 1, 1, 1;
   0, 1, 1, 1, 1;
   0, 1, 1, 1, 1;
   0, 1, 1, 1, 1;
   0, 0, 0, 0, 0
];
```

   start = [1, 1]; % Starting position (top-left)
   goal = [5, 5];   % Goal position (bottom-right)

4. **<u>6x6 Maze with Dead Ends (Path Exists)</u>**
   <u>maze</u> =

```
[
  0, 1, 1, 1, 1, 0;
  0, 1, 0, 0, 0, 0;
  0, 1, 0, 1, 1, 0;
  0, 1, 0, 0, 0, 0;
  1, 1, 1, 1, 0, 1;
  0, 0, 0, 0, 0, 0
];
```

start = [1, 1]; % Starting position (top-left)
goal = [6, 6];   % Goal position (bottom-right)

5. **<u>7x7 Maze with a Complex Path (Path Exists)</u>**
   <u>maze =</u>

```
[
  0, 1, 0, 0, 0, 0, 0;
  0, 1, 0, 1, 1, 1, 0;
  0, 1, 0, 0, 0, 1, 0;
  0, 1, 1, 1, 0, 1, 0;
  0, 1, 0, 0, 0, 0, 0;
  0, 1, 1, 1, 0, 1, 0;
  0, 0, 0, 0, 0, 0, 0
];
```

start = [1, 1]; % Starting position (top-left)
goal = [7, 7];   % Goal position (bottom-right)

6. **<u>8x8 Maze with Large Blocked Sections (Path Exists)</u>**
   <u>maze =</u>

```
[
  0, 1, 0, 0, 0, 0, 1, 0;
  0, 1, 0, 1, 1, 0, 1, 0;
  0, 1, 0, 1, 0, 0, 1, 0;
  0, 0, 0, 1, 0, 1, 1, 0;
  1, 0, 0, 1, 0, 1, 0, 0;
  1, 0, 0, 0, 0, 1, 0, 0;
  1, 0, 1, 1, 1, 1, 0, 0;
```

```
    0, 0, 0, 0, 0, 0, 0, 0
];
```

start = [1, 1]; % Starting position (top-left)
goal = [8, 8];   % Goal position (bottom-right)

7. **<u>Maze with Complex Obstacles and Multiple Paths</u>**

<u>maze =</u>

```
[
    0, 1, 1, 1, 0, 0, 0, 0;
    0, 1, 0, 0, 0, 1, 1, 0;
    0, 1, 0, 1, 1, 1, 0, 0;
    0, 0, 0, 0, 1, 0, 0, 0;
    1, 1, 1, 0, 1, 0, 0, 0;
    0, 0, 0, 0, 1, 0, 1, 0;
    0, 0, 0, 1, 1, 0, 0, 0;
    0, 0, 0, 0, 0, 0, 0, 0
];
```

start = [1, 1]; % Starting position (top-left)
goal = [8, 8];   % Goal position (b
ottom-right)

Explanation of the Code:

This MATLAB code implements a Maze Solver using the Breadth-First Search (BFS) algorithm. It allows the user to either input their own maze or use a predefined sample, and the program solves the maze to find the shortest path from the start to the goal point.

Key Functions and Components:

1. **mazeSolver Function:**
   - Prompts the user to input a maze or uses a default maze.
   - Specifies start and goal points for solving the maze.
   - Calls the BFS function to find the shortest path.
   - If a path is found, it visualizes the maze and the path; otherwise, it notifies the user that no path was found.

2. **bfs Function:**
   - Breadth-First Search (BFS) algorithm used to explore the maze.
   - Queue: It stores the coordinates of the current cell being explored.
   - Visited Matrix: Keeps track of which cells have been visited to avoid revisiting.
   - Parent Matrix: Stores the parent of each cell to reconstruct the path from the goal to the start.
   - Neighbor Exploration: The algorithm explores all valid neighbors (up, down, left, right) and continues until the goal is reached or all possible paths are explored.
   - If the goal is reached, it calls the reconstruct_path function to backtrack and find the complete path.

3. **reconstruct_path Function:**
   - Reconstructs the path from the goal to the start using the parent matrix.
   - This function traces the path by following the parent of each cell, starting from the goal and working back to the start.

4. **visualize_maze Function:**
   - Displays the maze and the BFS pathfinding process.
   - Walls are shown in black, open paths in white.
   - Visited cells are marked with blue circles, and the final path is shown in red.
   - Marks the start (green) and goal (magenta) positions on the maze.
   - Uses MATLAB's imagesc to plot the maze and plot to visualize the path and visited cells.

Algorithm Flow:

1. **User Input:**
   - The user can input their own maze or use the default one provided in the code.
   - The start and goal points are set in the maze (for example, start at [2,2] and goal at [6,6]).
2. **BFS Execution:**
   - The BFS function starts from the start point, exploring all valid neighbors (up, down, left, right).
   - As it explores, it marks visited cells and adds them to the queue.
   - If the goal is reached, it traces the path from the goal to the start using the parent matrix.
3. **Path Visualization:**
   - Once the path is found, the visualize_maze function is called to display the maze, showing the visited cells and the final path.

Key Concepts:

- Breadth-First Search (BFS): Explores the maze level by level, guaranteeing that the shortest path is found in an unweighted grid.
- Queue: BFS uses a queue to explore nodes in the order they were discovered.
- Parent Matrix: Used to reconstruct the path once the goal is reached.
- Visualization: The maze and its solution are visualized using plots to help users understand the algorithm's progress.

Example Maze:

For the default maze:

Copy code

```
1 1 1 1 1 1
1 0 0 0 1 1
1 0 1 0 1 1
1 0 1 0 0 1
1 0 1 1 0 1
1 1 1 1 0 0
```

- The start point is at [2, 2].
- The goal point is at [6, 6].
- The BFS algorithm explores the maze, finds the shortest path, and visualizes the process with plots.