

**Project Report**

On

**Find all Possible Paths in binary matrix**.



**Design and Analysis of Algorithms**

Masters of Computer Application

[AIML]

Submitted by: - Submitted to:

Muskan Devi Er. Divayanshi

24MCI10074 UID: E17438

24MAM-2A Subject Code: 24CAP-612





1. **Aim:**

This code aims to find all possible paths from the top-left corner to the rightmost column of a randomly generated binary matrix of specified size (rows x columns). In the matrix, cells with 1 represent traversable paths, and 0 represents obstacles.

1. **Problem definition:**
   1. **Input:**

1. Number of rows (rows): The user is prompted to enter the number of rows in the matrix.

2. Number of columns (columns): The user is prompted to enter the number of columns in the matrix.

* 1. **Constraints:**

1. Matrix Size: Performance may degrade with large matrices due to high memory and processing demands.

2. Path Constraints: Paths are limited to cells with 1s, and if no path exists from the start to the rightmost column, it returns zero paths.

3. Boundary Handling: Code has specific logic for boundary cells but may face issues with some edge cases.

4. Duplicate Paths: It doesn't filter out duplicate paths, potentially counting some paths multiple times.

5. Memory Usage: Storing visited paths for each cell can lead to high memory use, especially in larger matrices.

1. **Goal:**

Cells with 1 represent traversable paths.

Cells with 0 represent obstacles.

It uses a breadth-first search (BFS) to explore paths, stores each possible path, and calculates the total number of unique paths found. Additionally, it measures the execution time for performance analysis.

This is useful for pathfinding applications, such as solving mazes or navigating through grids with obstacles.

**3. Programming Languages used:**

The programming language used in the code you provided is **Python**.

1. **Syntax**: The code uses Python's syntax, such as def for defining functions, and indentation to denote code blocks instead of curly braces {}.
2. **Standard Libraries**: The code imports the random and timeit modules, which are both part of Python's standard library.
3. **Data Structures**: It uses Python-specific data structures such as lists ([]), which are commonly used in Python for storing collections of data.
4. **Python's List Comprehensions**: The code uses list comprehensions, a feature that is unique to Python and allows for concise and readable creation of lists.

**4. Flowchart:**

**Start**

**│**

**Input Rows & Columns**

**│**

**Generate Random Matrix**

**│**

**Set mat[0][0] = 1**

**│**

**Print Matrix**

**│**

**Initialize Variables (visited, q, paths)**

**│**

**Start Timer**

**│**

**While Queue is not Empty**

**├── Dequeue pt = [x, y]**

**│**

**├── Check pt Position (top-left, last column, edges, etc.)**

**│ └── Add valid neighbors to Queue**

**│**

**├── If pt is in last column**

**│ └── Increment paths if valid**

**│**

**├── Continue exploring neighboring cells**

**│**

**Backtrack to Find All Paths**

**│**

**If Paths Exist**

**├── Print some paths and costs**

**└── Print total paths and time taken**

**│**

**End**

**Success**

**6. Algorithm:**

 **Start**

* Begin the execution of the program.

 **Input Matrix Dimensions**

* User inputs the number of rows and columns for the matrix.

 **Generate Matrix**

* Randomly generate a matrix with values of 0 and 1.
* Ensure that the top-left cell (0,0) is set to 1.

 **Print Matrix**

* Display the generated matrix for visualization.

 **Initialize Variables**

* Create the visited matrix (for tracking the paths).
* Initialize the queue (starting point for BFS), q = [[0, 0]].
* Initialize path count (paths = 0).
* Start timer for performance measurement.

 **Start BFS Loop**

* **Condition**: While q is not empty, continue exploring.
* **Dequeue**: Pop an element from the queue (current cell: pt = [x, y]).
* **Check Current Cell Position**:
  + If it's the **top-left corner** (0,0), check and add valid adjacent cells to the queue.
  + If it's in the **last column**, check if the cell is valid (1), increment path count.
  + Handle cells in the **leftmost column**, **first row**, **last row**, **inner grid**, and **bottom-left corner**, ensuring the program adds valid neighboring cells to the queue and tracks visited paths.

 **Backtracking to Find Paths**

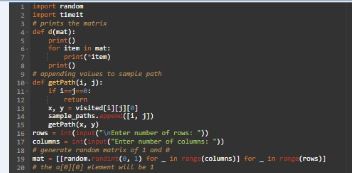
* For each valid path that reaches the last column, backtrack using the visited matrix to reconstruct the full path.
* Store the reconstructed path in a list of all\_paths.

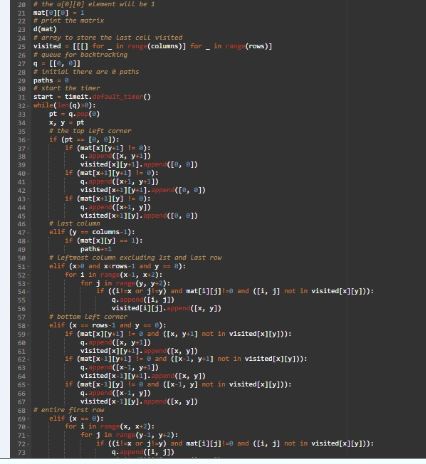
 **Output Results**

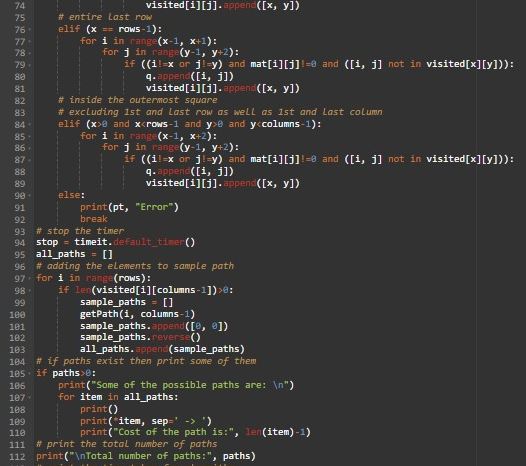
* **If paths exist**:
  + Print some of the found paths with the cost (length of the path).
* Print the **total number of paths** found.
* Display the **time taken** to run the algorithm.

 **End**

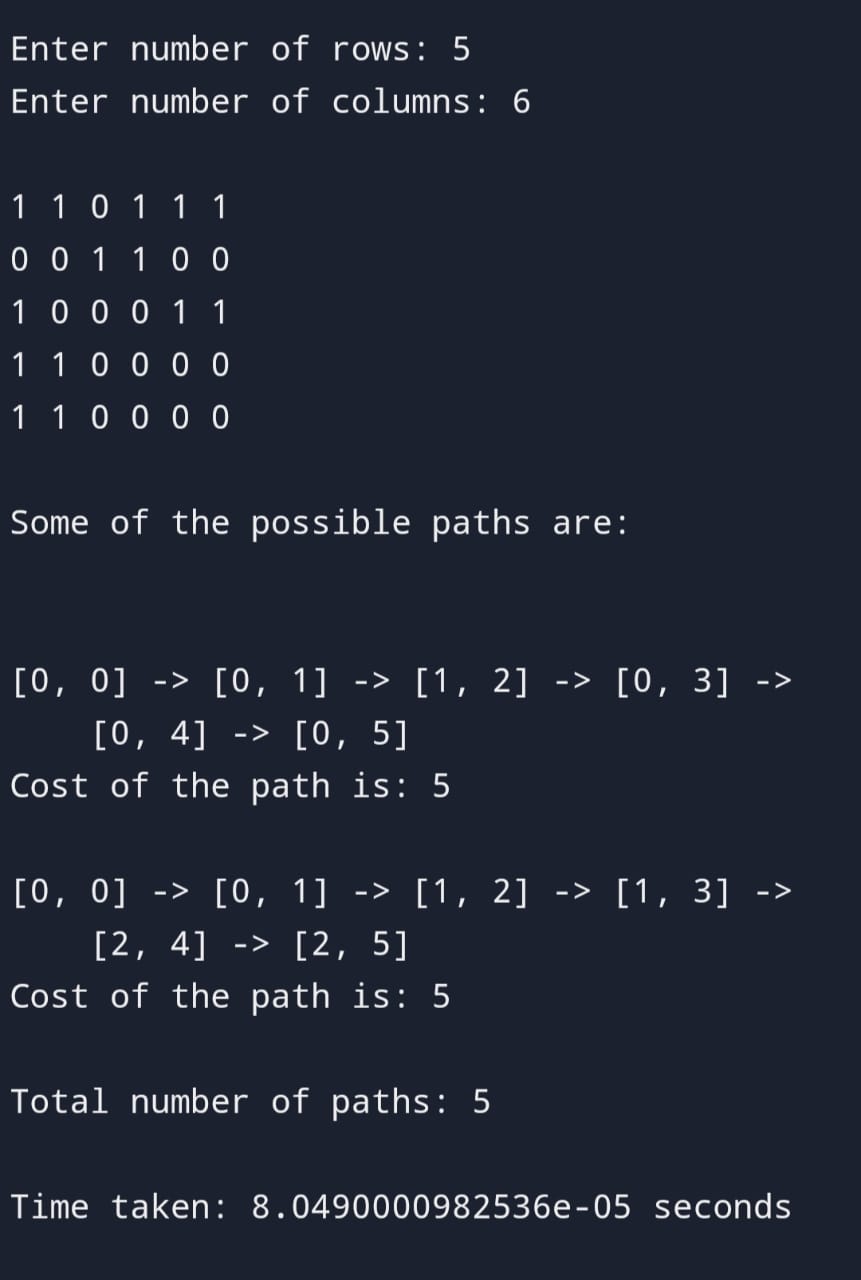
1. **Implementation:**







1. **Output:**



1. **Conclusions:**

This code successfully demonstrates a **pathfinding algorithm** using **breadth-first search (BFS)** to explore and find all possible valid paths in a randomly generated binary matrix. The matrix consists of passable cells (1s) and blocked cells (0s), and the goal is to find paths from the top-left corner (0,0) to any cell in the last column, avoiding blocked cells.

* **Pathfinding**: The BFS algorithm efficiently explores all potential paths by systematically visiting neighboring cells. It handles various edge cases, such as moving along the borders of the matrix (first and last rows/columns), while ensuring that paths do not revisit already visited cells.
* **Backtracking**: After finding the valid paths, the code reconstructs them using the visited matrix, effectively mapping the route taken to reach the last column for each path.
* **Performance**: The execution time is measured and printed, providing insight into the efficiency of the algorithm.
* **Flexibility**: By using random matrix generation, the program allows for testing with various matrix configurations and sizes, making it versatile for different use cases in pathfinding problems.

**10. Future frameworks:**

1. **Modularization**:

* Break down the code into separate functions to improve readability and maintainability. The pathfinding logic can be separated into functions that handle different matrix elements (e.g., handling first row, last column, etc.).

1. **Pathfinding Algorithm**:

* Consider using Breadth-First Search (BFS) or Depth-First Search (DFS) for better control over pathfinding, especially for finding the shortest path or handling specific constraints.

1. **Matrix Representation**:

* Instead of randomly generating a binary matrix, consider allowing user input or allowing the matrix to be loaded from a file. Additionally, add functionality to visualize the matrix and paths.

1. **Error Handling**:

* Improve error handling for edge cases (e.g., matrices with no valid paths, edge conditions where rows/columns are 1).

1. **Optimization**:

* Minimize redundant checks in the algorithm to reduce time complexity, such as checking if a position is already visited and handling the queue more efficiently.

1. **Path Output**:

* Instead of printing all paths, allow the option to output a specific number of paths or a path length threshold. Also, visualize the paths within the matrix.

1. **Path Cost Calculation**:

* Introduce a clearer cost calculation methodology based on actual weight of the paths or steps involved (if considering weighted matrices in the future)

.

**11. Learning outcomes:**

1. **Matrix Representation and Manipulation**:
   * How to create and work with a random binary matrix in Python.
   * How to ensure specific cells (e.g., start point) are set to a particular value (e.g., 1).
2. **Pathfinding Algorithms**:
   * Understanding basic pathfinding techniques using **Breadth-First Search (BFS)** to explore and find paths in a grid.
   * How to backtrack from the destination cell to the start and reconstruct the full path using a visited list to store previous nodes.
3. **Graph Traversal**:
   * Implementing grid traversal by considering all four possible directions (up, down, left, right) and ensuring only valid moves are taken (within bounds and non-zero cells).
4. **Queue Usage in BFS**:
   * How to use a **queue (FIFO structure)** to explore nodes in BFS, ensuring that all neighbors are processed level by level.
5. **Efficiency Considerations**:
   * How to track visited cells to avoid redundant work and prevent infinite loops.
   * Basic understanding of how BFS guarantees finding the shortest path in an unweighted grid.
6. **Modular Programming**:
   * How to break down a larger problem (pathfinding) into smaller, reusable functions (print\_matrix, get\_path, find\_paths) for cleaner and more manageable code.
7. **Performance Measurement**:
   * Using the timeit module to measure and report the time taken by the algorithm to run, which helps assess the efficiency of your code.
8. **Handling Edge Cases**:
   * How to handle grid boundaries, ensuring the algorithm doesn't access out-of-bound indices.
   * Special cases like no valid paths and paths being available from start to destination.

By working through this code, you would gain practical experience in implementing pathfinding algorithms, matrix manipulation, and optimizing code for better performance and readability.

**Time complexity:**

* **Matrix generation**: O(rows \* columns)
* **BFS traversal**: O(rows \* columns)
* **Path reconstruction**: O(paths \* columns)
* **Overall Time Complexity:** O(rows \* columns + paths \* columns)

**Space Complexity:**

* **Matrix storage**: O(rows \* columns)
* **Visited matrix**: O(rows \* columns)
* **Queue for BFS**: In the worst case, the queue will store all the cells, so it requires O(rows \* columns) space.
* **Paths storage**: Space required to store the paths is O(paths \* columns).
* **Overall space complexity :**O(rows \* columns)

**Github link:** [**https://github.com/MUSKANTIWARI003/DAA.Project-/blob/main/Daa.project**](https://github.com/MUSKANTIWARI003/DAA.Project-/blob/main/Daa.project)

**Github screenshot:**

