I. DFS

```
# Maze dimensions and obstacles
maze_size = 6
obstacles = [(0,1),(1,1),(3,2),(3,3),(3,4),(3,5),(0,4),(4,1),(4,2),(4,3)]
start = (0,0)
goal = (0,5)

# checks whether a given position of (x,y) is valid to move or not
def is_valid(x,y):
 return 0 <= x < maze_size and 0 <= y < maze_size and (x,y) not in obstacles

#Dfs function (Depth-first search)
def dfs (current, visited, path):
 x, y = current
 if current == goal:
  path.append(current)
  return True
 visited.add(current)
 moves = [(x-1,y), (x+1, y), (x, y-1), (x, y+1)]
 for move in moves:
  if is_valid(*move) and move not in visited:
   if dfs(move, visited, path):
    path.append(current)
    return True
 return False

#Call DFS function to find the path
visited = set()
path = []
if dfs(start, visited, path):
```

```
  path.reverse()
  print("Path found:")
  for position in path:
    print(position)
else:
  print("No path found!")
```

II. BFS

```
from collections import deque



class GridProblem:
    def __init__(self, initial_state, goal_state, grid):
        # Initializes a grid problem instance with initial and goal states, and the grid layout
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.grid = grid

    def is_goal(self, state):
        # Checks if the given state is the goal state
        return state == self.goal_state

    def is_valid_cell(self, row, col):
        # Checks if the given cell coordinates are within the grid boundaries and not
blocked
        return 0 <= row < len(self.grid) and 0 <= col < len(self.grid[0]) and
self.grid[col][row] == 0

    def expand(self, node):
        # Expands the given node by generating child nodes for valid adjacent cells
        row, col = node.state
        children = []
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            new_row, new_col = row + dr, col + dc
            if self.is_valid_cell(new_row, new_col):
                child_state = (new_row, new_col)
                child_node = Node(child_state, parent=node)
```

```
            children.append(child_node)
        return children


class Node:
    def __init__(self, state, parent=None, action=None):
        # Initializes a node with a state, parent node (optional), and action (optional)
        self.state = state
        self.parent = parent
        self.action = action


def breadth_first_search(problem):
    # Performs breadth-first search algorithm to find a solution for the given problem
    node = Node(problem.initial_state)
    if problem.is_goal(node.state):
        return node

    frontier = deque([node])
    reached = {problem.initial_state}

    while frontier:
        node = frontier.popleft()

        for child in problem.expand(node):
            state = child.state

            if problem.is_goal(state):
                return child
            if state not in reached:
```

```
            reached.add(state)
            frontier.append(child)
    return None


def reconstruct_path(node):
    # Reconstructs the path from the goal node back to the initial node
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))


def print_complete_path(path):
    # Prints the complete path from start to goal
    if path:
        for step, point in enumerate(path):
            print("Step {}: {}".format(step, point))
    else:
        print("No solution found")


# Example usage and grid definition
"""
    1 : Denotes the obstacles
    0 : Empty space or a non-obstacle cell in the grid
"""
grid = [
    [0, 1, 0, 0, 1, 0, 0],
```

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

# Define initial and goal states
initial_state = (0, 0)
goal_state = (6, 0)

# Define the problem instance
problem = GridProblem(initial_state, goal_state, grid)

# Perform breadth-first search to find a solution
solution_node = breadth_first_search(problem)

# Print solution if found
print('!! Reached the Goal!!' if solution_node else None)
if solution_node:
    print("Solution found!")
    solution_path = reconstruct_path(solution_node)
    print("Complete Path:")
    print_complete_path(solution_path)
else:
    print("No solution found")
```