2. State Space Search: Robot Pathfinding in a Maze

Use Case: Program an agent to navigate through a maze to find the shortest path to the exit.

Objective: Implement and compare Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms to solve the maze. Analyze the performance in terms of time and space complexity.

Experiment Outline:

In this experiment, the goal is to navigate an agent (robot) from a starting point to an exit in a maze by implementing and comparing two popular search algorithms: Depth-First Search (DFS) and Breadth-First Search (BFS). The primary objectives are to:

1. Understand how DFS and BFS explore a maze to find paths.
2. Compare the two algorithms in terms of the length of the path found, execution time, and space complexity.

1. Setting Up the Environment

We start by importing the necessary libraries:

- **NumPy** for handling maze data in a matrix form.
- **Matplotlib** for visualizing the maze and paths.
- **Deque** (from the collections module) for implementing the BFS queue efficiently.
- **Time** to measure the execution time of each algorithm.

2. Maze Creation

A maze is represented as a 2D grid (matrix) where:

- 0 represents open cells that the agent can move through.
- 1 represents obstacles or walls that the agent cannot traverse.

The create_maze() function sets up this maze with defined start and goal points.

```
def create_maze():
    maze = np.array([
        [0, 1, 0, 0, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0],
        [0, 0, 0, 1, 0, 0, 0],
        [0, 1, 1, 1, 1, 1, 0],
        [0, 0, 0, 0, 0, 1, 0]
    ])
```

```
    start = (0, 0)   # Top-left corner
    goal = (4, 6)    # Bottom-right corner
    return maze, start, goal
```

3. Visualizing the Maze and Path

The display_maze() function visualizes the maze and any path found. Cells in the path are marked with a unique value for easy identification.

```
def display_maze(maze, path=[]):
    maze_copy = np.copy(maze)
    for position in path:
        maze_copy[position] = 2  # Path marked with '2'

    plt.imshow(maze_copy, cmap="coolwarm")
    plt.xticks([]), plt.yticks([])
    plt.show()
```

4. Implementing Breadth-First Search (BFS)

BFS is a search algorithm that explores neighbors level by level. It uses a queue (FIFO) to ensure that each level is explored before moving deeper. BFS is optimal for finding the shortest path in an unweighted maze because it explores all shortest paths first.

1. **Initialize**: Begin with a queue containing the start point.
2. **Explore**: For each cell, check its four possible neighbors (up, down, left, right).
3. **Goal Check**: If the goal is reached, break out and backtrack to construct the path.
4. **Return Path**: Reconstruct the path by following the parent pointers from the goal to the start.

```
def bfs(maze, start, goal):
    queue = deque([start])
    visited = set()
    visited.add(start)
    parent_map = {}

    while queue:
        node = queue.popleft()

        if node == goal:
            break

        for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            neighbor = (node[0] + direction[0], node[1] + direction[1])

            if (0 <= neighbor[0] < maze.shape[0] and 0 <= neighbor[1] < maze.shape[1] and
                neighbor not in visited and maze[neighbor] == 0):

                visited.add(neighbor)
```

```
            parent_map[neighbor] = node
            queue.append(neighbor)

    path = []
    if goal in parent_map:
        while goal != start:
            path.append(goal)
            goal = parent_map[goal]
        path.append(start)
        path.reverse()

    return path
```

5. Implementing Depth-First Search (DFS)

DFS is a search algorithm that explores each path as far as possible before backtracking. DFS uses a stack (LIFO), which can lead it down a single path until it either reaches the goal or hits an obstacle.

1. **Initialize**: Start with a stack containing the start point.
2. **Explore**: For each cell, check the four directions, just as in BFS.
3. **Goal Check**: If the goal is found, exit the loop and backtrack to build the path.
4. **Return Path**: Construct the path from goal to start following parent pointers.

```
def dfs(maze, start, goal):
    stack = [start]
    visited = set()
    visited.add(start)
    parent_map = {}

    while stack:
        node = stack.pop()

        if node == goal:
            break

        for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            neighbor = (node[0] + direction[0], node[1] + direction[1])

            if (0 <= neighbor[0] < maze.shape[0] and 0 <= neighbor[1] < maze.shape[1] and
                neighbor not in visited and maze[neighbor] == 0):

                visited.add(neighbor)
                parent_map[neighbor] = node
                stack.append(neighbor)

    path = []
    if goal in parent_map:
        while goal != start:
            path.append(goal)
            goal = parent_map[goal]
```

```
        path.append(start)
        path.reverse()

    return path
```

6. Performance Analysis Function

This function evaluates the performance of each algorithm by measuring:

1. **Execution Time**: The time taken to find the path.
2. **Space Complexity**: Approximated here by the length of the final path (for simplicity).

```
def analyze_performance(algorithm, maze, start, goal):
    start_time = time.time()
    path = algorithm(maze, start, goal)
    end_time = time.time()

    execution_time = end_time - start_time
    space_complexity = len(path)

    return path, execution_time, space_complexity
```

7. Running and Comparing BFS and DFS

We now run both BFS and DFS on the same maze, and we compare their outputs (path, execution time, and space complexity).

```
maze, start, goal = create_maze()

# BFS
bfs_path, bfs_time, bfs_space = analyze_performance(bfs, maze, start, goal)
print("BFS Path:", bfs_path)
print("BFS Time:", bfs_time)
print("BFS Space Complexity:", bfs_space)
display_maze(maze, bfs_path)

# DFS
dfs_path, dfs_time, dfs_space = analyze_performance(dfs, maze, start, goal)
print("DFS Path:", dfs_path)
print("DFS Time:", dfs_time)
print("DFS Space Complexity:", dfs_space)
display_maze(maze, dfs_path)
```

8. Analysis and Conclusion

1. **Path Length**: BFS finds the shortest path as it explores all nodes layer by layer. DFS, however, may not find the shortest path since it explores depth-first and may take longer routes.

2. **Execution Time**: BFS might take longer in larger mazes with many paths, as it explores every level. DFS, being depth-first, may take a more direct but sometimes less optimal path.
3. **Space Complexity**: BFS uses more memory as it stores all nodes at each level. DFS, which backtracks, may use less memory since it only tracks nodes in the current depth path.

This experiment demonstrates the practical implications of choosing between BFS and DFS for different scenarios, especially in environments with various obstacles and paths.

Labset 2: State space search

In [1]:
```
# important libraries
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
import time
```

In [2]:
```
# create maze
def create_maze():
    maze = np.array([
        [0, 1, 0, 0, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0],
        [0, 0, 0, 1, 0, 0, 0],
        [0, 1, 1, 1, 1, 1, 0],
        [0, 0, 0, 0, 0, 1, 0]
    ])
    start = (0, 0)
    goal = (4, 6)
    return maze, start, goal
```

In [3]:
```
# visualization function for maze
def display_maze(maze, path=[]):
    maze_copy = np.copy(maze)
    for position in path:
        maze_copy[position] = 2  # Path marked with '2'

    plt.imshow(maze_copy, cmap="coolwarm")
    plt.xticks([]), plt.yticks([])
    plt.show()
```

In [4]:

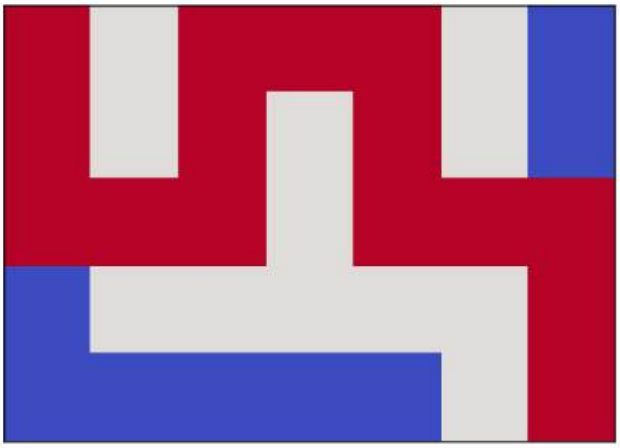
```
# implement BFS
def bfs(maze, start, goal):
    queue = deque([start])
    visited = set()
    visited.add(start)
    parent_map = {}

    while queue:
        node = queue.popleft()

        if node == goal:
            break

        for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            neighbor = (node[0] + direction[0], node[1] + direction[1])

            if (0 <= neighbor[0] < maze.shape[0] and 0 <= neighbor[1] < maz
e.shape[1] and
                neighbor not in visited and maze[neighbor] == 0):

                visited.add(neighbor)
                parent_map[neighbor] = node
                queue.append(neighbor)

    path = []
    if goal in parent_map:
        while goal != start:
            path.append(goal)
            goal = parent_map[goal]
        path.append(start)
        path.reverse()

    return path
```

In [5]:
```python
# implement DFS
def dfs(maze, start, goal):
    stack = [start]
    visited = set()
    visited.add(start)
    parent_map = {}

    while stack:
        node = stack.pop()

        if node == goal:
            break

        for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            neighbor = (node[0] + direction[0], node[1] + direction[1])

            if (0 <= neighbor[0] < maze.shape[0] and 0 <= neighbor[1] < maz
e.shape[1] and
                neighbor not in visited and maze[neighbor] == 0):

                visited.add(neighbor)
                parent_map[neighbor] = node
                stack.append(neighbor)

    path = []
    if goal in parent_map:
        while goal != start:
            path.append(goal)
            goal = parent_map[goal]
        path.append(start)
        path.reverse()

    return path
```

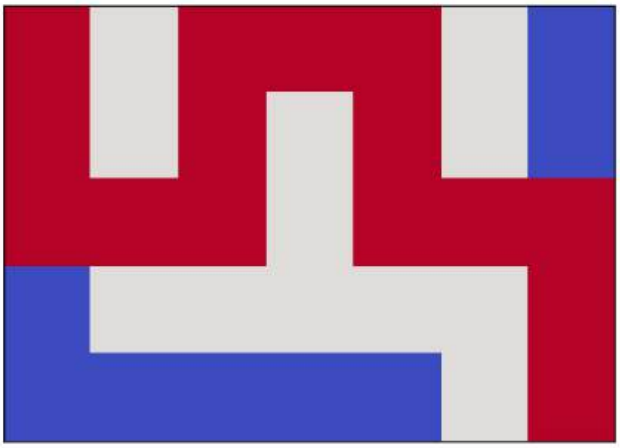
In [6]:
```python
# analyze and compare performance
def analyze_performance(algorithm, maze, start, goal):
    start_time = time.time()
    path = algorithm(maze, start, goal)
    end_time = time.time()

    execution_time = end_time - start_time
    space_complexity = len(path)

    return path, execution_time, space_complexity
```

In [7]:
```
# Run and compare BFS and DFS
maze, start, goal = create_maze()

# BFS
bfs_path, bfs_time, bfs_space = analyze_performance(bfs, maze, start, goal)
print("BFS Path:", bfs_path)
print("BFS Time:", bfs_time)
print("BFS Space Complexity:", bfs_space)
display_maze(maze, bfs_path)

# DFS
dfs_path, dfs_time, dfs_space = analyze_performance(dfs, maze, start, goal)
print("DFS Path:", dfs_path)
print("DFS Time:", dfs_time)
print("DFS Space Complexity:", dfs_space)
display_maze(maze, dfs_path)
```

```
BFS Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (1, 2), (0, 2), (0, 3),
(0, 4), (1, 4), (2, 4), (2, 5), (2, 6), (3, 6), (4, 6)]
BFS Time: 0.0
BFS Space Complexity: 15
```



```
DFS Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (1, 2), (0, 2), (0, 3),
(0, 4), (1, 4), (2, 4), (2, 5), (2, 6), (3, 6), (4, 6)]
DFS Time: 0.0
DFS Space Complexity: 15
```



In [ ]: