In the **Moving Maze Problem**, the idea is to navigate a maze where the walls or obstacles may change position over time. Unlike a static maze where the walls remain fixed, a moving maze introduces dynamic elements, requiring algorithms to adapt to changing obstacles.

Let's assume the following:

* The maze is represented by a 2D grid where:
  + 0 represents an open path.
  + 1 represents a wall or obstacle.
* The walls move over time, changing their positions.
* You start at the top-left corner and aim to reach the bottom-right corner.

We'll approach the solution using **BFS**, **DFS**, **UCS**, and **GBFS** while taking into account that walls move dynamically.

**Basic Problem Assumptions:**

* You have an n x m maze.
* The walls may move every step (e.g., each move could shift one row up or down).

We'll first define the helper functions for moving within the maze and updating the walls.

**Helper Functions**

python

Copy code

# Define the moves (up, down, left, right)

DIRECTIONS = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up

# Function to check if a position is valid and within the maze

def is\_valid(x, y, maze):

return 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 0

# Function to simulate moving walls (just an example of dynamic walls)

def move\_walls(maze):

# Example: Shift all walls down by 1 row, looping the bottom row to the top

new\_maze = [[0] \* len(maze[0]) for \_ in range(len(maze))]

for i in range(1, len(maze)):

new\_maze[i] = maze[i-1][:] # Move walls down

new\_maze[0] = maze[-1][:] # Loop last row to the top

return new\_maze

**1. BFS (Breadth-First Search) for Moving Maze**

In **BFS**, we explore level by level and update the maze's walls at each step to simulate the movement of the obstacles.

python

Copy code

from collections import deque

def bfs\_moving\_maze(start, goal, maze):

queue = deque([(start, [])]) # (current position, path)

visited = set()

visited.add(start)

while queue:

(x, y), path = queue.popleft()

if (x, y) == goal:

return path # Return the path to the goal

# Move walls for the next step

maze = move\_walls(maze)

for dx, dy in DIRECTIONS:

new\_x, new\_y = x + dx, y + dy

if is\_valid(new\_x, new\_y, maze) and (new\_x, new\_y) not in visited:

visited.add((new\_x, new\_y))

queue.append(((new\_x, new\_y), path + [(new\_x, new\_y)]))

# Example usage:

start = (0, 0)

goal = (4, 4)

maze = [

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

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

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

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

[0, 0, 0, 0, 0]

]

solution = bfs\_moving\_maze(start, goal, maze)

print("Solution using BFS:", solution)

**Explanation**:

* BFS explores all paths level by level and updates the walls at every step using move\_walls() to simulate the maze's movement.
* It finds the shortest path by expanding all possible moves at the current level before moving to the next.

**2. DFS (Depth-First Search) for Moving Maze**

**DFS** will explore one path to its depth before backtracking. We'll also update the walls dynamically during each step.

python

Copy code

def dfs\_moving\_maze(start, goal, maze):

stack = [(start, [])] # (current position, path)

visited = set()

visited.add(start)

while stack:

(x, y), path = stack.pop()

if (x, y) == goal:

return path # Return the path to the goal

# Move walls for the next step

maze = move\_walls(maze)

for dx, dy in DIRECTIONS:

new\_x, new\_y = x + dx, y + dy

if is\_valid(new\_x, new\_y, maze) and (new\_x, new\_y) not in visited:

visited.add((new\_x, new\_y))

stack.append(((new\_x, new\_y), path + [(new\_x, new\_y)]))

# Example usage:

solution = dfs\_moving\_maze(start, goal, maze)

print("Solution using DFS:", solution)

**Explanation**:

* DFS explores one path completely before moving on to another. It updates the walls at each step.
* This approach may not always find the shortest path since DFS explores deeply before backtracking.

**3. UCS (Uniform Cost Search) for Moving Maze**

In **UCS**, each move has a uniform cost, and we prioritize the paths with the least cost. Walls are updated dynamically at each step.

python

Copy code

def ucs\_moving\_maze(start, goal, maze):

queue = [(0, start, [])] # (cost, current position, path)

visited = set()

visited.add(start)

while queue:

queue.sort(key=lambda x: x[0]) # Sort by cost

cost, (x, y), path = queue.pop(0)

if (x, y) == goal:

return path # Return the path to the goal

# Move walls for the next step

maze = move\_walls(maze)

for dx, dy in DIRECTIONS:

new\_x, new\_y = x + dx, y + dy

if is\_valid(new\_x, new\_y, maze) and (new\_x, new\_y) not in visited:

visited.add((new\_x, new\_y))

queue.append((cost + 1, (new\_x, new\_y), path + [(new\_x, new\_y)]))

# Example usage:

solution = ucs\_moving\_maze(start, goal, maze)

print("Solution using UCS:", solution)

**Explanation**:

* UCS prioritizes paths with the least cost. In this case, each move has a uniform cost, so UCS behaves similarly to BFS.
* The maze's walls are dynamically updated at each step.

**4. GBFS (Greedy Best-First Search) for Moving Maze**

In **GBFS**, we use a heuristic to guide the search. For the moving maze, we can use the **Manhattan distance** between the current position and the goal.

python

Copy code

# Manhattan distance as heuristic

def manhattan\_distance(x1, y1, x2, y2):

return abs(x1 - x2) + abs(y1 - y2)

def gbfs\_moving\_maze(start, goal, maze):

queue = [(manhattan\_distance(\*start, \*goal), start, [])] # (heuristic, current position, path)

visited = set()

visited.add(start)

while queue:

queue.sort(key=lambda x: x[0]) # Sort by heuristic (Manhattan distance)

\_, (x, y), path = queue.pop(0)

if (x, y) == goal:

return path # Return the path to the goal

# Move walls for the next step

maze = move\_walls(maze)

for dx, dy in DIRECTIONS:

new\_x, new\_y = x + dx, y + dy

if is\_valid(new\_x, new\_y, maze) and (new\_x, new\_y) not in visited:

visited.add((new\_x, new\_y))

h\_value = manhattan\_distance(new\_x, new\_y, goal[0], goal[1])

queue.append((h\_value, (new\_x, new\_y), path + [(new\_x, new\_y)]))

# Example usage:

solution = gbfs\_moving\_maze(start, goal, maze)

print("Solution using GBFS:", solution)

**Explanation**:

* GBFS uses the Manhattan distance heuristic to prioritize moves that seem closer to the goal.
* Like other algorithms, the maze walls move dynamically at each step.

**Summary:**

* **BFS**: Explores level by level and is guaranteed to find the shortest path.
* **DFS**: Explores one path deeply before backtracking, but may not always find the shortest path.
* **UCS**: Prioritizes the least-cost path (in this case, all moves have the same cost, so it behaves like BFS).
* **GBFS**: Uses a heuristic (Manhattan distance) to guide the search towards the goal.

In all cases, the maze's walls are dynamically updated at each step, simulating a moving maze, and the algorithms must adapt to these changes while finding a path to the goal.