Exercise: MazeGame

The game initializes with a maze size of 5x5.
The loop continues indefinitely until the player either reaches the exit or hits an obstacle.

During each iteration of the loop:
・The current game state is displayed, showing the player's position, the exit position, and the maze layout.
・The player is prompted to enter a move direction (up, down, left, or right).
・The player's position is updated according to the input direction.
・The game status is checked to determine whether the player wins, hits an obstacle, or continues exploring.
・Depending on the game status, an appropriate message is printed, and the loop either continues or breaks to end the game.

Modify the mazeGame.py to

1. Generate a maze with at least one path to exit. The pathways of the maze must be randomized.
2. Automatically solve the maze.

Submission requirements:

1. Source codes (Part1.py; Part2.py)
2. PDF documents
   Explaining your strategy and demonstrate the results using 10x10 maze.
3. Additional features! (shortest path, multiple paths, … and more)
4. Upload to e-learning before 3/15 14:10
5. Schedule a demo with the course TA (賴佳嬪)

Part 1

Generate a maze with at least one path to exit. The pathways of the maze must be randomized.

Strategy:

Use BFS (Breadth First Search) to check whether the randomized maze has a route can go from the source to the end. If it's not, rerandomize the maze until we get a maze with a route can go from source to the end.

Program:

```
def generate_maze(self):
    for i in range(self.size):
        for j in range(self.size):
            # 生成隨機迷宮，0 表示通道，1 表示障礙物
            self.maze[i, j] = random.choice([0, 1])

    # 確保起點和終點是通道
    self.maze[0, 0] = 0
    self.maze[self.size - 1, self.size - 1] = 0
```

Original function of generating maze

```
def generate_maze(self):
    while True:
        for i in range(self.size):
            for j in range(self.size):
                # 生成隨機迷宮，0 表示通道，1 表示障礙物
                self.maze[i, j] = random.choice([0, 1])

                # 確保起點和終點是通道
                self.maze[0, 0] = 0
                self.maze[self.size - 1, self.size - 1] = 0

        if self.check():
            break
```

Revised version of generating maze function

The difference between the original function and revised version is that the revised one use a check() function to ensure the maze has a route can go from the source to the end.

```
def check(self):
    def bfs():
        nonlocal record, queue

        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        temp = set()
        for x, y in queue:
            for a, b in directions:
                if (x + a, y + b) not in record and 0 <= x + a < self.size and 0 <= y + b < self.size
                    and self.maze[x + a][y + b] != 1:
                    record.add((x + a, y + b))
                    temp.add((x + a, y + b))

        queue = temp

    record = set([(0, 0)])
    queue = set([(0, 0)])
    while queue:
        bfs()
        if (self.size - 1, self.size - 1) in queue:
            return True
    return False
```

Function checking whether a route from source to the end exist.

Use BFS (Breath First Search) to expand the nodes to check whether a path can go from the source to the end. If there's a path are found, return True immediately.

Part 2

Automatically solve the maze.

Strategy:

Use Dijkstra algorithm to find the shortest path from the source to the end.

Program:

```
maze_game = MazeGame(10)

while True:
    maze_game.display_game()
    print("Enter your move (up, down, left, right):")
    move = input().strip().lower()
    maze_game.move_player(move)
    status = maze_game.check_game_status()
    if status == 'Win':
        print("Congratulations! You win!")
        break
    elif status == 'Hit obstacle':
        print("Oops! You hit an obstacle. Game over!")
        break
    else:
        print("Continue exploring...")
```

Original main function to solve the maze

```
maze_game = MazeGame(10)
shortest_path = maze_game.shortest_path()
i = 0

while True:
    maze_game.display_game()
    move = shortest_path[i]
    i += 1
    print(f"Your move: {move}")
    maze_game.move_player(move)
    status = maze_game.check_game_status()
    if status == 'Win':
        print("Congratulations! You win!")
        break
    elif status == 'Hit obstacle':
        print("Oops! You hit an obstacle. Game over!")
        break
    else:
        print("Continue exploring...")
```

Revised version of main function

Before starting to solve the maze, get the shortest path first. After getting the path, use the result to replace human entering to reach the goal of automatically solving.

```
def shortest_path(self):
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    record = {(0, 0): (0, 1, 0)}
    queue = [(0, (0, 0), (-1, -1))]
    while queue:
        v, (x, y), (mx, my) = heapq.heappop(queue)
        record[(x, y)] = (v, mx, my)
        for a, b in directions:
            if 0 <= a + x < self.size and 0 <= b + y < self.size and self.maze[x + a][y + b] == 0:
                if (a + x, b + y) not in record or v < record[a + x, b + y][0]:
                    if (a + x, b + y) == (self.size - 1, self.size - 1):
                        queue = []
                        break
                    heapq.heappush(queue, (v + 1, (a + x, b + y), (x, y)))
                    record[(a + x, b + y)] = (v + 1, x, y)

    coordinates = [(self.size - 1, self.size - 1)]
    while True:
        coordinates.append((x, y))
        x, y = mx, my
        _, mx, my = record[(x, y)]
        if (mx, my) == (-1, -1):
            break
    coordinates.append((0, 0))

    result = []
    for i in range(len(coordinates) - 1, 0, -1):
        x, y = coordinates[i]
        a, b = coordinates[i - 1]
        if x > a:
            result.append("up")
        elif x < a:
            result.append("down")
        elif y > b:
            result.append("left")
        else:
            result.append("right")
    return result
```

Function to find shortest path

Use Dijkstra algorithm to find the shortest path. The original queue contains the source only. After expanding the nodes, we'll approach the shortest path gradually. When we encounter the same node, compare the distance, keep the shortest one.

After Dijkstra is run to the end, get the path by backtracking the nodes' parents. Finally, we'll get the shortest path with nodes. Then, convert the route to directions by each adjacency nodes. That's the result we'll use in the main function.