Wroclaw University Of Science And Technology

CANER OLCAY 276715

Artificial Intelligence and Computer Vision

17.11.2024

List 5

```python
from collections import deque

def display_maze(maze):
    for row in maze:
        print(''.join(row))
    print()

def find_positions(maze, start_symbol='S', goal_symbol='G'):
    start, goal = None, None
    for i, row in enumerate(maze):
        for j, cell in enumerate(row):
            if cell == start_symbol:
                start = (i, j)
            elif cell == goal_symbol:
                goal = (i, j)
    if not start or not goal:
        raise ValueError("Maze must contain 'S' (start) and 'G' (goal) symbols.")
    return start, goal

# Depth-First Search
def dfs(maze, start, goal):
    stack = [start]
    visited = set()
    parent = {}

    while stack:
        current = stack.pop()
        if current in visited:
            continue
        visited.add(current)

        if current == goal:
            break

        for d in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            ni, nj = current[0] + d[0], current[1] + d[1]
            if 0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and maze[ni][nj] not in ('#', 'S') and
(ni, nj) not in visited:
                stack.append((ni, nj))
                parent[(ni, nj)] = current

    current = goal
    while current != start:
        if current != goal:  # Ensure the goal cell remains 'G'
            maze[current[0]][current[1]] = '*'
        current = parent.get(current, start)
    maze[start[0]][start[1]] = 'S'
    return maze

# Example maze
maze = [
    list("################"),
    list("#******#########"),
    list("#*####*********#"),
    list("# ############*#"),
    list("#*****#*#***##G#"),
    list("#*###*###*#****#"),
    list("#S###*****######"),
    list("################"),
]

start, goal = find_positions(maze)

print("Original Maze:")
display_maze(maze)

# Solve using DFS
dfs_solution = [row[:] for row in maze]
dfs_solution = dfs(dfs_solution, start, goal)
print("Maze Solved with DFS:")
display_maze(dfs_solution)
```

Task 1: Maze Solving Using Depth-First Search (DFS)

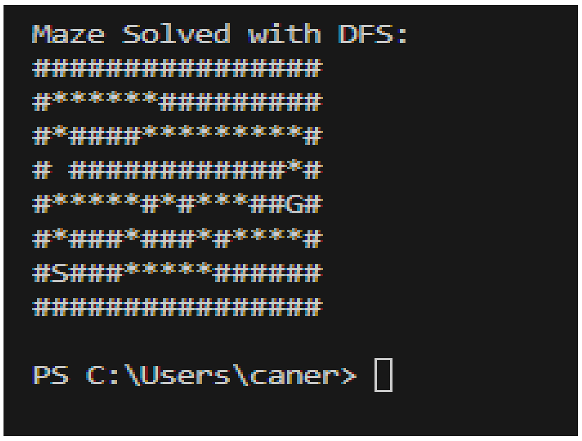
This task focuses on solving a maze using the Depth-First Search (DFS) algorithm. The algorithm identifies a path from the start position (S) to the goal position (G) while ensuring the visual integrity of the maze by preserving the goal marker G. The key details of the implementation are as follows:

1. Maze Representation:
    - The maze is represented as a 2D grid with the following elements:
        - #: Walls that block movement.
        - S: The starting position.
        - G: The goal position.
        - *: The solution path, marking the cells visited by the algorithm.
2. Algorithm Description:
    - The algorithm begins at the S position and explores paths deeply using a stack until the goal position (G) is reached.
    - A parent dictionary is used to record the path, allowing the algorithm to backtrack from the goal to the start to reconstruct the solution.
    - The visited set ensures that cells are not revisited, avoiding infinite loops.
3. Special Consideration:
    - To maintain the clarity of the solution, the G position remains unaltered while the path leading to it is marked with *.
4. Outputs:
    - The original maze is displayed for reference.

    - After solving, the maze is shown with the solution path from S to G clearly marked.
5. Advantages of DFS:

    - DFS explores paths deeply, making it efficient in mazes where solutions are located deeper within the search tree.
    - It is simple to implement and does not require additional memory for exploring paths level by level.

Sample Output: The solved maze with the path marked:



```
Maze Solved with DFS:
################
#******#########
#*####*********#
# ############*#
#*****#*#***##G#
#*###*###*#****#
#S###*****######
################

PS C:\Users\caner> 
```

This solution ensures correctness by retaining the goal position (G) and visually marking the shortest path found by the DFS algorithm.

Task 2:

```
from collections import deque
import heapq

def display_maze(maze):
    for row in maze:
        print(''.join(row))
    print()

def find_positions(maze, start_symbol='S', goal_symbol='G'):
    start, goal = None, None
    for i, row in enumerate(maze):
        for j, cell in enumerate(row):
            if cell == start_symbol:
                start = (i, j)
            elif cell == goal_symbol:
                goal = (i, j)
    if not start or not goal:
        raise ValueError("Maze must contain 'S' (start) and 'G' (goal) symbols.")
    return start, goal

# A* Algorithm

def a_star_with_path_costs(maze, start, goal):
    open_set = []
    heapq.heappush(open_set, (0, start))
    g_score = {start: 0}
    parent = {}

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            break

        for d in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            ni, nj = current[0] + d[0], current[1] + d[1]
            if 0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and maze[ni][nj] not in ('#', 'S'):
                tentative_g_score = g_score[current] + 1
                if (ni, nj) not in g_score or tentative_g_score < g_score[(ni, nj)]:
                    g_score[(ni, nj)] = tentative_g_score
                    f_score = tentative_g_score + abs(ni - goal[0]) + abs(nj - goal[1])
                    heapq.heappush(open_set, (f_score, (ni, nj)))
                    parent[(ni, nj)] = current

    # Reconstruct the path and assign costs
    current = goal
    while current != start:
        if current != goal:  # Ensure the goal cell remains 'G'
            maze[current[0]][current[1]] = str(g_score[current])[-1]  # Show path cost as last
digit
        current = parent.get(current, start)
    maze[start[0]][start[1]] = 'S'
    return maze

# Example maze
```

```
maze = [
    list("###############"),
    list("#S#     #    #"),
    list("# # ### # #  #"),
    list("# #   # # # G#"),
    list("# ### # ###  #"),
    list("#     #      #"),
    list("###############"),
]

start, goal = find_positions(maze)

print("Original Maze:")
display_maze(maze)

# Solve using A*
a_star_solution = [row[:] for row in maze]
a_star_solution = a_star_with_path_costs(a_star_solution, start, goal)
print("Maze Solved with A* Path Costs:")
display_maze(a_star_solution)
```

*Maze Solving Using A Algorithm with Path Costs**

In this task, the A* algorithm is used to solve a maze by finding the optimal path from the start (S) to the goal (G). Additionally, the solution visualizes the path costs for each cell along the way, reflecting the cumulative cost to reach that cell. The key points of the implementation are as follows:

1. Maze Representation:
    - The maze is a 2D grid with:
        - #: Walls that block movement.
        - S: The starting point.
        - G: The goal position.
        - Numerical values: Path costs showing the cumulative steps required to reach each cell along the solution path.
2. *A Algorithm Description**:
    - A* is an informed search algorithm that uses a priority queue to explore paths with the lowest estimated cost first.
    - It combines:
        - g_score: The actual cost from the start to the current cell.
        - h_score: The heuristic estimate (Manhattan distance) from the current cell to the goal.
        - f_score = g_score + h_score: The total estimated cost.

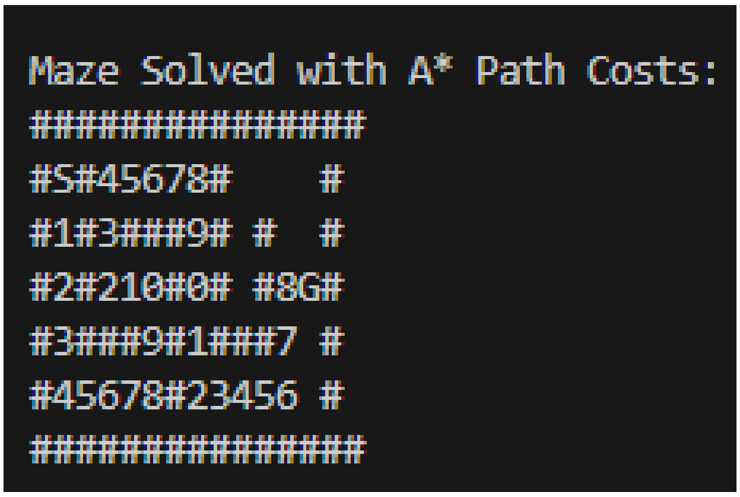
- The algorithm iteratively selects the cell with the lowest f_score from the priority queue and explores its neighbors.

3. Visualization of Path Costs:
    - Each cell along the solution path is assigned a cost, displayed as the last digit of the cumulative cost to reach that cell.
    - The goal cell (G) remains unaltered for clarity.
4. Outputs:
    - The original maze is displayed first for reference.
    - After solving the maze, the solution is shown with path costs, providing a clear visualization of the optimal path and its associated costs.

Key Features of A*:

- Guarantees the shortest path in terms of cost when the heuristic (Manhattan distance) is admissible.
- Explores paths intelligently, reducing the number of visited nodes compared to uninformed algorithms like BFS or DFS.

Sample Output: The solved maze with path costs:



```
Maze Solved with A* Path Costs:
###############
#S#45678#    #
#1#3###9# #  #
#2#210#0# #8G#
#3###9#1###7 #
#45678#23456 #
###############
```