NSYSU-MATH Data Structure – Spring 2024

Homework 3

Design: Maze Solver Design Using Depth-First Search with Stack

Data Preparation

For this assignment, you will be provided with a zip file named `HW3.zip`, which contains template files and public test data. Your primary objective is to implement the `search_from_stack()` function within the `Maze` class, using either Python or C++. Below is an overview of the directory structure and the contents in the zip file:

1. Python Implementation (`Py`/ directory):
   - ✓ `maze_stack.py`: This is where you will implement your `search_from_stack()` function..
   - ✓ `maze.py`: Contains a reference implementation of the search algorithm using recursion, for your review and understanding.
   - ✓ `*.txt`: Includes test data files for mazes and paths.
2. C++ Implementation (`Cpp/` directory):
   - ✓ `maze_stack.cpp:` This is where you will implement your `search_from_stack()` function.
   - ✓ `maze.cpp:` Provides a reference implementation using recursion, which can help guide your own implementation.
   - ✓ `*.txt:` Test data files for mazes and paths.
   - ✓ *.h: Header files used for graphical representations and additional functionality.

Description

In our recent lessons, we've explored how recursive implementations can be transformed into stack-based approaches. This assignment requires you to apply this concept by designing a maze solver that employs depth-first search (DFS), similar to the examples in our textbook. Unlike the examples, however, you are tasked with utilizing a stack-based approach. The assignment is structured into three main parts as follows:

1. Modify the maze search algorithm in `maze.py` (for Python) or `maze.cpp` (for C++) so that the calls to `search_from()` follow a different order. After making these adjustments, run the program to observe any resultant changes in behavior. Explain the significance of call order in the algorithm's behavior and explain whether it affects the algorithm's ability to find a path through the maze.

2. Function Implementation:
   - ✓ Implement the function named `search_from_stack()` in the provided template file. For Python, use `maze_stack.py`. For C++, use `maze_stack.cpp`.
3. Discussion:
   - ✓ Discuss the difference between recursion and stack implementation and their pros and cons.

Specifications

1. Function name: `search_from_stack()`
2. Input: The function takes three parameters: `maze`, which is a custom class we define representing the maze, and `start_row, start_col,` which are the starting coordinates.
3. Output: The function should return a Boolean value to indicate whether a path has been found. It should also return the found path from start to exit as a list or vector of tuples/pairs, representing the row and column coordinates.
4. **The `maze` is stored as a matrix, detailed in our textbook.**
5. **Use the `list` (in Python) or a `vector` (in C++) to store the found path. The path should be a list of tuples that store the row and column coordinates as tuples. In C++, you should use `vector` of pairs. (e.g. [(3,4), (3,5)….])**
6. The depth-first search algorithm should be used for this implementation, following the same order of the original recursion implementation.
7. You are allowed to use only the standard libraries of Python or C++. In addition, use the `Stack`() class from our provided code base.
8. Do not focus on visualization for this assignment; your primary goal is to ensure the correct path and Boolean value are returned. However, if you are interested, feel free to explore the GUI code or display your path.

Usage of the programs

1. Use `python .\maze.py .\maze1.txt` to execute the program and run the maze solver with the specified maze file.
2. Use `python .\maze.py .\maze1.txt .\correct_path1.txt` to compare the solution generated by your program against a correct path file, verifying the implementation's accuracy.
3. Use `python .\maze.py .\maze1.txt -nogui` if you wish to run the program without the graphical user interface, suitable for environments where GUI support is unavailable or unnecessary.
4. Ensure to add `-O2 -lgdi32` flags when compiling your C++ program on Windows to optimize the execution and include necessary libraries for graphics support.

Deliverables

1. Deadline: 2024/4/28 (Sun.), 11:59 PM. Hand in the following two items to the cyber universities. Please see our Facebook group for the late policy and rules.
2. Report:
   - ✓ Modify the maze search program in `maze.py` (or `maze.cpp` for C++) so that the calls to `search_from()` are in a different order. Observe the changes in the program's behavior, explain the reasons for any differences, and discuss the impact of call order on the algorithm's ability to find a path through the maze.
   - ✓ Describe the design of your program and the data structures you utilized. Discuss what you have learned from completing this homework.
   - ✓ Analyze the differences between recursion and stack implementation, including their advantages and disadvantages.
3. Program Source Files:
   - ✓ Submit your source files and report according to instructions stated here. **Ensure that you follow the provided template files**.
   - ✓ Source File Comments: Each file must begin with three lines of comments indicating the Author, Date, and Purpose of the program. Include appropriate comments throughout your code for clarity.

Grading Policy

- Function Correctness: 60% (36% for public test cases and 24% for hidden test cases).
- Report and discussion: 40%.

Reference

1. https://runestone.academy/ns/books/published/pythonds3/Recursion/ExploringaMaze.html
2. https://medium.com/swlh/solving-mazes-with-depth-first-search-e315771317ae
3. https://varsubham.medium.com/maze-path-finding-using-dfs-e9c5fa14106f
4. https://sqlpad.io/tutorial/python-maze-solver/