Institute of Pervasive Computing

Algorithms and Data Structures 1
Summer term 2024
Jäger, Rihani, Vázquez Pufleau, Anzengruber-Tanase

Assignment 3

Deadline: **Thu. 02.05.2024, 12:00**
Submission via: **Moodle**

Elaboration time

Remember the time you need for the elaboration of this assignment and document it in Moodle.

Recursion

For this assignment, please submit the pdf of the pen-and-paper work and the source code of your `my_maze.py`.

1. Maze Algorithm — 12 points

Develop a recursive algorithm that **recursively** finds and marks the exits of a given maze with a size of at least 3x3. A maze is represented by a string, with rows delimited by \n:

- '#' … represent obstacles/walls, which cannot be traversed
- 'S' … denote the starting position
- 'X' … mark a found exit (on the outer rims of the maze)
- ' ' … empty/blank fields represent possible paths
- '.' … dots mark visited positions/cells

The algorithm:

- Search from a given starting position 'S' and mark the visited positions/cells with a dot ('.').
- Allowed moves are **east, southeast, south, southwest, west, northwest, north and northeast.**
- Mark all found exits with 'X'

In the table below you can see an example for a maze (without exits) for which you should test your algorithm step by step before implementing it. The coordinate of a cell is written as (row, column). The **starting position 'S'** is given by the last two digits of your student ID, e.g., for the student ID k163452**70**, the starting position would be in row 7 and column 0 like shown in the example below (if your student ID ends with 07 use row 0 and column 7 instead). In case your individual starting position is an obstacle '#', you can just overwrite it. **Provide the first eight steps** listing the current **depth**, **position**, if it is an **obstacle or visited**, and **neighbor positions** like shown in the table below. List the neighbor positions based on the given **move order** above, i.e., starting with east and moving clockwise. The next current position is always chosen as the first unvisited position from the last neighbor list ("depth-first-search").

```
   -1012345678910
-1 ############
 0 # #  # ##  #
 1 # # #  # # #
 2 ## #  # #  #
 3 #  # #  # ##
 4 # # ## # # #
 5 #  #  # #  #
 6 ## ## ##  ##
 7 #S##  ## # #
 8 # # #  # # #
 9 #  # #  #  #
10 ############
```

Depth	Curr. Position (row, column)	Obstacle / visited?	Neighbor Positions
0	(7,0)	N	(7,1), (8,1), (8,0), (8,-1), (7,-1), (6,-1), (6,0), (6,1)
1	(7,1)	Y	-
1	(8,1)	Y	-
1	(8,0)	N	(8,1), (9,1), (9,0), (9,-1), (8,-1), (7,-1), (7,0), (7,1)
2	(8,1)	Y	-
2	(9,1)	N	(9,2), (10,2), (10,1), (10,0), (9,0), (8,0), (8,1), (8,2)
3	(9,2)	Y	-
3	(10,2)	Y	-

Your solution based on your student ID:

```
   -1012345678910
-1 ############
 0 # #  # ##  #
 1 # # #  # # #
 2 ## #  # #  #
 3 #  # #  # ##
 4 # # ## # # #
 5 #  #  # #  #
 6 ## ## ##  ##
 7 # ##  ## # #
 8 # # #  # # #
 9 #  # #  #  #
10 ############
```

Depth	Curr. Position (row, column)	Obstacle/ visited?	Neighbor Positions
0		N	

Institute of Pervasive Computing

Algorithms and Data Structures 1
Summer term 2024
Jäger, Rihani, Vázquez Pufleau, Anzengruber-Tanase

Assignment 3

Deadline: **Thu. 02.05.2024, 12:00**
Submission via: **Moodle**

2. Maze Implementation — 12 points

Implement your recursive algorithm from Example 1 using the provided skeleton in the file **my_maze.py** as follows:

- As described in Example 1, start at a given starting position (specified with `start_row`/`start_col` as parameters in `find_exits`) and move **east, southeast, south, southwest, west, northwest, north and northeast** (in this order!) if the cell is not an obstacle.
- Update the `_maze` variable so that empty cells are replaced with a dot as soon as they have been visited. Note that the final `_maze` can still contain empty cells if they were not accessible from 'S'.
- Search the entire maze for exits, which are accessible empty cells on the outer rim of the maze, i.e., with a row number of 0 or `height-1` or a column number of 0 or `width-1`. Create a list of tuples `_exits` with the coordinates (row, column) of all exits and mark all found exits with 'X'. Return true if at least one exit has been found, otherwise false.
- Track the maximum recursion depth during the maze exploration and return it with the property `max_recursion_depth()`.
- **Create your own mazes to test, one in a size of 15x15, another in a size of 20x20 (both containing an exit) – use e.g., a separate main method to execute!**

```
class MyMaze:
    """Maze object, used for demonstrating recursive algorithms."""

    def __init__(self, maze_str: str):
        """Initialize Maze.

        Args:
            maze_str (str): Maze represented by a string,
            where rows are separated by newlines (\n).

        Raises:
            ValueError, if maze_str is empty.
        """
        # We internally treat this as a List[List[str]], as it makes indexing easier.

    def find_exits(self, start_row: int, start_col: int, depth: int = 0) -> bool:
        """Find and save all exits into `self._exits` using recursion, save the
        maximum recursion depth into 'self._max_recursion_depth' and mark the maze.
        An exit is an accessible from S empty cell on the outer rims of the maze.

        Args:
            start_row (int): row to start from. 0 represents the topmost cell.
            start_col (int): column to start from; 0 represents the leftmost cell.
            depth (int): Depth of current iteration.

        Raises:
            ValueError: If the starting position is out of range or not walkable path.
        """

    @property
    def exits(self) -> List[Tuple[int, int]]:
        """List of tuples of (row, col)-coordinates of currently found exits."""

    @property
    def max_recursion_depth(self) -> int:
        """Return the maximum recursion depth after executing find_exits()."""
```

***Note**:*

- *You can make use of the given __str__-method in the skeleton to print your maze at different states for debugging purpose.*
- *Return the coordinates in the correct order as shown in the pen & paper exercise!*