DS2030 Data Structures and Algorithms for Data Science
Lab 3 (Take Home) Due on September 21, 11.59pm

Instructions

- You are to use Python as the programming language. Use may use Visual Studio Code (or any other editor you are comfortable with) as the IDE.
- You have to work individually for this lab.
- You are not allowed to share code with your classmates nor allowed to use code from the internet. You are encouraged engage in high level discussions with your classmates; however ensure to include their names in the report/code documentation. If you refer to any source on the Internet, include the corresponding citation in the report/code documentation. If we find that you have copied code from your classmate or from the Internet, you will get a straight fail grade in the course.
- The submission must be a zip file with the following naming convention - rollnumber.zip. The Python files should be contained in a folder named after the question number.
- Include appropriate comments to document the code. Include a `read me` file containing the instructions on for executing the code. The code should run on institute linux machines.
- Upload your submission to moodle by the due date and time. Do not email the submission to the instructor or the TA.

This lab will improve your understanding of stacks and queues.

1 8-puzzle problem (10 points)

The eight puzzle problem, also known as the sliding tile puzzle, is a classic problem in the field of artificial intelligence and computer science. It is played on a 3×3 grid with eight numbered tiles and one empty space. The goal of the puzzle is to rearrange the tiles from an initial configuration to a desired target configuration by sliding the tiles into the empty space.
The tiles are numbered from 1 to 8, and they are initially placed in a random order within the grid. The empty space is represented by a blank tile. The puzzle can be visualized as follows:

1		3
4	2	5
7	8	6

The player can move the tiles into the adjacent empty space by swapping their positions. Only one tile can be moved at a time, and the movement is restricted to horizontal and vertical directions. Diagonal movements are not allowed.
The objective of the puzzle is to reach a target configuration, which is typically represented as follows:

1	2	3
4	5	6
7	8	

The target configuration has the tiles arranged in ascending order, starting from the top-left corner. The empty space is located in the bottom-right corner in the target configuration.

Solving the eight puzzle problem involves finding a sequence of moves that transform the initial configuration into the target configuration. The challenge lies in finding an optimal solution with the minimum number of moves. Various search algorithms, such as breadth-first search, depth-first search, A* search, and heuristic-based algorithms, can be applied to find a solution to the problem. In this lab we will implement bread-first and depth first search to find the sequence of moves for transforming the initial tile layout to the target layout.
We begin by first defining the states and actions (moves) to write the program. We will assume that the input puzzle state (start state) is represented as a 2D list. The empty tile is represented by the number 0. For example `start_state = [[1, 0, 3], [4, 2, 5], [7, 8, 6]]` describes the initial state illustrated before. The set of actions or moves at any state is defined as `actions = [(0, -1), (0, 1), (-1, 0), (1, 0)] # Possible movements: left, right, up, down`. We have provided helper functions to take care of all puzzle related modules. You will have to implement the depth first and breadth first search functions. You may use the stack and queue classes defined in the book while implementing these functions.
Generating the entire tree and storing it apriori is a memory intensive process. Hence, we will expand the tree on need basis. The following function generated the next state given the current state and the action to be performed

```python
# Function to generate the next state given a current state and an action
# - Copy the current state value into a new variable
# - Determine the location of the empty tile (this is the tile that will move)
# - Dbtain the information pertaining to the action
# - Perform the action - obtaining the new location that should contain the empty tile
# - If the action results in a valid state, then copy the tile in the new location
# (according to the current state) to the old location of the empty tile and
# assign 0 to the new location
def generate_next_state(current_state, action):
    next_state = copy.deepcopy(current_state)
    empty_row, empty_col = find_empty_tile(next_state)
    move_row, move_col = action
    next_row, next_col = empty_row + move_row, empty_col + move_col

    # Check if the move is valid (within bounds of the puzzle)
    if 0 <= next_row < len(current_state) and 0 <= next_col < len(current_state[0]):
        next_state[empty_row][empty_col] = next_state[next_row][next_col]
        next_state[next_row][next_col] = 0
        return next_state
    else:
        return None  # Invalid move, return None

# Function to find the empty tile in the puzzle
def find_empty_tile(state):
    for row in range(len(state)):
        for col in range(len(state[row])):
            if state[row][col] == 0:
                return row, col
```

Code Fragment 1: Generating the next state

The next important function is to check if the current state is the goal state. The following helper function does it for you.

```python
# Function to check if the generated state is goal
def is_goal_state(state, goal_state):
    return state == goal_state
```

Code Fragment 2: Checking for goal state

The last helper function is for printing the puzzle in a visually appealing format.

```python
# Function to print the puzzle state in a visually appealing format
def print_puzzle(state):
    for row in state:
        print(' '.join(map(str, row)))
    print()
```

Code Fragment 3: Printing the puzzle

Your objective is to complete the definition of the following two functions for depth first search and breadth first search.

```python
# Depth-First Search algorithm
# If the goal state is achievable, the function should return the sequence of states
    from the start state to the goal state
# Use a list (or another data structure) to store the states that have already been
    visited. This will require additional memory, but will save us computational time.
def depth_first_search(start_state, goal_state):
```

Code Fragment 4: Depth First Search

```python
# Breadth-First Search algorithm
# If the goal state is achievable, the function should return the sequence of states
    from the start state to the goal state
# Use a list (or another data structure) to store the states that have already been
    visited. This will require additional memory, but will save us computational time.
def breadth_first_search(start_state, goal_state):
```

Code Fragment 5: Breadth First Search