N-puzzle Problem Description

This task aims to teach you how to implement the A* search algorithm to solve 15-puzzle problem.

Prerequisites

- Python 3.7 (at least)
- numpy
- copy

Some libs may not be installed automatically when Python is installed before. You should manually install them via package manager like `pip` or `conda`.

File Structure

puzzle_state.py

In this file, class `PuzzleState` is defined to represent problem state. In the constructor, `square_size` denotes the size of the chessboard (e.g. `square_size=3` means a 3-by-3 chessboard, namely, the 8-puzzle problem). In this project, we set `square_size=4` to solve a 15-puzzle problem.

- **You don't need to edit codes in `PuzzleState`!**
- Method `generate_state()` is prepared to be called to randomly generate the state of chessboard. Parameter `random` is to be set as `True` if you want a random state generation.
- `run_moves()` function can be used to determine whether the generated moving commands are available to move from the initial state to the target state.
- `print_moves()` function prints the details of each step during the applying of the moving commands.
- **You are required to complete the function `astar_search_for_puzzle_problem()` to implement the A* searching core codes.**

n_puzzle_state_main.py

The main file of the N-puzzle problem solving program. The default N is set to 15, that is, a 15-puzzle problem with `square_size=4`.

- A fixed target state is set previously. The initial state is achieved by a series of random moving from the target state. The default moving step number is 100.
- **You don't need to edit codes in this file!** Once you have completed the A* searching in its corresponding function, run this main file to apply moving commands and print result.

Tips

You are required to complete `astar_search_for_puzzle_problem()` function in `puzzle_state.py`. The input parameters are `init_state` denoting initial state and `dst_state` denoting target state. A `move_list` denoting the list of moving commands is returned. When we apply the commands from `move_list` one-by-one, the state will transit from `init_state` to `dst_state`.

Here provides a simple program pipeline. Pay attention that this part is just for reference. Better implementations may be proposed by yourselves. See more details in the code.

```
def astar_search_for_puzzle_problem(init_state, dst_state):
    """
    Use AStar-search to find the path from init_state to dst_state
    :param init_state:  Initial puzzle state
    :param dst_state:   Destination puzzle state
    :return:  All operations needed to be performed from init_state to dst_state
        moves: list of Move. e.g: move_list = [Move.Up, Move.Left, Move.Right,
Move.Up]
    """
    start_state = init_state.clone()
    end_state = dst_state.clone()

    open_list = []   # You can also use priority queue instead of list
    close_list = []

    move_list = []  # The operations from init_state to dst_state

    # Initial A-star
    open_list.append(start_state)

    while len(open_list) > 0:
        # Get best node from open_list
        curr_idx, curr_state = find_front_node(open_list)

        # Delete best node from open_list
        open_list.pop(curr_idx)

        # Add best node in close_list
        close_list.append(curr_state)

        # Check whether found solution
        if curr_state == dst_state:
            moves = get_path(curr_state)
            return moves

        # Expand node
        childs = expand_state(curr_state)

        for child_state in childs:

            # Explored node
            in_list, match_state = state_in_list(child_state, close_list)
            if in_list:
                continue

            # Assign cost to child state. You can also do this in Expand
operation
            child_state = update_cost(child_state, dst_state)

            # Find a better state in open_list
            in_list, match_state = state_in_list(child_state, open_list)
            if in_list:
                continue
            open_list.append(child_state)
```

Some helper functions are detailed as follows:

- `find_front_node()`: Get best node. In this code, a list `open_list` is adopted to retain the unexplored nodes. You can replace it with a priority queue (or heap) to improve the efficiency of the program.
- `state_in_list()`: Check whether a state is in the state list.
- `get_path()`: Return moving commands list from initial state to current state.
- `expand_state()`: Expand current state to generate children nodes.
- `once_move`: Update the parent node information in the child node but not update the cost value.
- `update_cost()`: Update the cost value of current state (including g and h). **Implement your heuristic method here.**

Please explicitly call the function `update_cost()` in order to make us check your design of the heuristics.

Others

- Mind that the moving direction is of the space tile in the chessboard.
- Mind the types of the variables when you are coding.
- Mind the order of moving commands.
- For debugging, you can lower the input steps in function `generate_moves()` to get an easier test scenario.
- If you want to test your heuristic function in harder scenarios:
  - Larger input steps in `generate_moves()` for deeper searching depth.
  - Randomly generate both the initial and the target state. Or you can manually design hard states.
  - Larger `square_size`. Test your function in 24-puzzle or even 35-puzzle problem.