15 Puzzle Game

The 15 Puzzle game is a `4x4` tile based game where player moves a single tile, the empty tile, represented as `0`, until it reaches a complete state that looks like:

```
+----+----+----+----+
|  1 |  2 |  3 |  4 |
|----+----+----+----|
|  5 |  6 |  7 |  8 |
|----+----+----+----|
|  9 | 10 | 11 | 12 |
|----+----+----+----|
| 13 | 14 | 15 |  0 |
+----+----+----+----+
```

Consider the following starting game state:

```
+----+----+----+----+
|  1 |  2 |  3 |  4 |
|----+----+----+----|
|  5 |  6 |  7 |  8 |
|----+----+----+----|
|  9 |  0 | 11 | 12 |
|----+----+----+----|
| 13 | 10 | 14 | 15 |
+----+----+----+----+
```

The empty tile, tile `0`, has to move `down right right` in order to arrive at the complete state.

On every move, the empty tile is swapped with an adjacent tile. Tiles can only be swapped horizontally and vertically.

Note that the larger versions of the 15 Puzzle game can be hard to solve. In fact, finding an optimal solution is NP-Complete. However, we can still easily find approximate solutions to the puzzle.

Approximate solutions are actual solutions to the puzzle, but require more moves than optimal solutions (which are the shortest amount of moves to solve the puzzle).

The 15 Puzzle Library

This library implements the 15 Puzzle game in `Python`. You can use this library for:

- Build a 15 Puzzle solver
- GUI for this library
- Improve the library itself
- Implement missing functions

If you want to build a puzzle solver, you can use:

- BFS algorithm
- A* search algorithm
- Databases

A simple approach to building a puzzle solver would be to use the A* algorithm, with Manhattan distance.

Other possible heuristics:

- Misplaced tiles
- Inversions in row or column

Tips:

- Avoid exploring board states you have already seen
- Use data structures to efficiently retrieve states to explore
- The algorithm may run for too long, so have a limit on runtime
- Some algorithms use a lot of memory (not good)

For 15 Puzzle (N=4), difficulty up to 4 is OK (and good for testing). Higher difficult can take longer, but is a good way to test the efficiency of your algorithm.

How the code works

The library implements `class nsquare`. It contains a lot of useful functions, but it lacks a puzzle solver (which you may implement). Here are examples on how to use the `nsquare` library:

```
>> import nsquare
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
 3  6  8  1
 0  2  7  4
 5 10 11 15
 9 13 12 14
```

The size and difficulty of the puzzle are defined in `nsquare(size, difficulty)`. If `size=4`, the this is equivalent to 15 Puzzle. `size=3` is equivalent to 8 Puzzle.

`nsquare` starts off with the solved puzzle board. Then it performs lots of swaps, according to difficulty level. This generates the board that the user needs to solve. Because of this process, there is always a solution to the puzzle.

The row and column indexing are as follows:

```
  col    0    1    2    3
      +----+----+----+----+
row 0 |  1 |  2 |  3 |  4 |
      |----+----+----+----|
row 1 |  5 |  6 |  7 |  8 |
```

```
      |----+----+----+----|
row 2 |  9 |  0 | 11 | 12 |
      |----+----+----+----|
row 3 | 13 | 10 | 14 | 15 |
      +----+----+----+----+
```

nsquare(4, 1)

Function `nsquare(4, 1)` takes as parameter the puzzle size of 4 (15 tiles) and difficulty level 1. It is advised to use puzzle size 4. Higher size may make it harder to solve the puzzle. Higher difficulty (up to 10) can take longer to solve.

When you call `nsquare(4, 1)`, it automatically generates a new board for the player to solve. To not generate a puzzle game and instead generate solved state, ie `[[1,2,3,4],...,[13,14,15,0]]`, use `nsquare(4, 1, default=True)`.

Note that `nsquare` also stores the key to the puzzle in `board_key`. It is advised to not use the key. The key stores the sequence of moves to perform from the *original* generated board, to the get to the solved board.

Also note that `nsquare` keeps track of the *original* board (the one initially generated), and also the *current* board.

The *current* board stores the board after moves were performed. For example, you can perform moves on the board using `move_up()`, and that updates the `nsquare` board (the *original* stays the same).

ns.print_board()

Function `ns.print_board()` actually returns the board as a string, and does not print. That is why you need to print it using `print()`.

The parameter `original=True` allows to print the board that was generated initially. Use `original=False` to print the current board.

```
>> import nsquare
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
 3  6  8  1
 0  2  7  4
 5 10 11 15
 9 13 12 14
>> ns.move_up()
>> ns.move_up()
>> print(ns.print_board())
 1  2  3  4
 0  6  7  8
 5  9 11 12
13 10 14 15
```

```
>> print(ns.print_board(original=True))
 1  2  3  4
 5  6  7  8
13  9 11 12
 0 10 14 15
```

generate_initial_board()

Function `generate_initial_board()` and `generate_game()` are used to generate the board for the game. You should not use these, unless you want to change the way boards are generated (by modifying the source code). Think of these as `private` functions.

get_possible_moves()

Function `get_possible_moves()` may be useful for you if you are building a puzzle solver.

Given a current `nsquare` board, it returns a list of possible moves, where the empty tile can go. Outputs array where `u=up`, `d=down`, `l=left`, `r=right`.

```
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
13  9 11 12
 0 10 14 15
>> ns.get_possible_moves()
['u', 'r']
```

We can see that tile zero can only move `up` or `right`.

swap()

Function `swap()` is supposed to be a private function.

You should use `move_up()`, `move_down()`, `move_left()`, `move_right()` instead of `swap()`.

move_up(), move_down(), move_left(), move_right()

Functions `move_up()`, `move_down()`, `move_left()`, `move_right()` move the empty tile on the board.

> **Important**: these functions assume that it is possible to move in that direction in the first place. Use `get_possible_moves()` before using the `move` functions.

```
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
```

```
 1  2  3  4
 5  6  7  8
13  9 11 12
 0 10 14 15
>> ns.get_possible_moves()
['u', 'r']
>> ns.move_up()
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
 0  9 11 12
13 10 14 15
>> ns.get_possible_moves()
['u', 'd', 'r']
>> ns.move_right()
>> ns.get_possible_moves()
['u', 'd', 'l', 'r']
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
 9  0 11 12
13 10 14 15
```

get_key()

Returns the key to go from original board, to solution state.

copy_board()

Function `copy_board(new_board, board_type)` copies the puzzle board from current board, to array `new_board`.

If `board_type='o'`, it copies the original board. If `board_type='c'`, it copies the current board.

```
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
13  9 11 12
 0 10 14 15
>> ns.move_up()
>> ns.move_up()
>> ns.move_up()
>> print(ns.print_board())
 0  2  3  4
 1  6  7  8
```

```
 5  9 11 12
13 10 14 15
>> arr1 = [1, 2, 3]
>> arr2 = [1, 2, 3]
>> ns.copy_board(arr1, 'o')
>> ns.copy_board(arr2, 'c')
>> arr1
[[1, 2, 3, 4], [5, 6, 7, 8], [13, 9, 11, 12], [0, 10, 14, 15]]
>> arr2
[[0, 2, 3, 4], [1, 6, 7, 8], [5, 9, 11, 12], [13, 10, 14, 15]]
```

It may be useful to use `copy_board()` to copy boards into `nsquare.board` or `nsquare.board_original`.

get_tile()

Function `get_tile(x,y)` returns the tile value from board at position `(x,y)`. Indexing starts at 0, from top left corner.

get_empty_tile()

Function `get_empty_tile()` returns array `[row, col]` with the row and column of the empty tile.

```
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
13  9 11 12
 0 10 14 15
>> ns.get_empty_tile()
[3, 0]
```

reverse_solution()

Function `reverse_solution(s)` is a private function that reverses a solution.

It is useful when you know the moves to go from board state `A` to state `B`, and you want the moves to go back from `B` to `A`.

check_solution()

Function `check_solution(solution)` checks if moves in `solution` are a solution to the *original* puzzle (not the current board in `nsquare`).

```
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
```

```
13  9 11 12
 0 10 14 15
>> ns.get_key()
['u', 'r', 'd', 'r', 'r']
>> ns.check_solution(['u', 'r', 'd', 'r', 'r'])
True
>> ns.check_solution(['u', 'r'])
False
```

find_empty_tile()

Function `find_empty_tile()` returns the location of the empty tile, `[row, col]`.

reset_board()

Function `reset_board()` resets the board to original state.

```
>> ns = nsquare.nsquare(4, 1)
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
13  9 11 12
 0 10 14 15
>> ns.move_up()
>> ns.move_up()
>> ns.move_right()
>> ns.move_up()
>> print(ns.print_board())
 1  0  3  4
 6  2  7  8
 5  9 11 12
13 10 14 15
>> ns.reset_board()
>> print(ns.print_board())
 1  2  3  4
 5  6  7  8
13  9 11 12
 0 10 14 15
```

Disclaimer: code was tested, but may still contain some bugs.

Written in Summer 2024