Richard Messina rdm420 N13468622
CS4613 Project 1 - 8 Puzzle Problem Solver

INSTRUCTIONS

The included file, *solver.py*, contains the source code to my solution. It was written for Python 3.7.2, but most versions of Python 3 should work. Instructions for running this script can be found as follows:

```
> python3 solver.py --help
usage: solver.py [-h] -i INPUT -o OUTPUT -f FUNCTION

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Input file name
  -o OUTPUT, --output OUTPUT
                        Output file name
  -f FUNCTION, --function FUNCTION
                        The heuristic function to use. "1"=sum of Manhattan
                        distances of tiles from their goal position. "2"=
                        sum of Manhattan distances + 2 * # linear conflicts
```

Therefore, to run this file against certain inputs, you can follow this example:

```
> python3 solver.py -i InputFile.txt -o OutputFile.txt -f 1
```

RESULTING OUTPUT FILES

Output for Input1.txt, heuristic 1

```
7 1 6
8 3 5
2 0 4

8 7 6
1 0 5
2 3 4

5
12
U U L D R
5 5 5 5 5 5
```

Output for Input1.txt, heuristic 2

```
7 1 6
8 3 5
2 0 4

8 7 6
1 0 5
2 3 4

5
12
U U L D R
5 5 5 5 5 5
```

Output for Input2.txt, heuristic 1

```
2 6 0
1 3 4
7 5 8

1 2 3
4 5 6
7 8 0

10
27
L D R U L L D R D R
10 10 10 10 10 10 10 10 10 10 10
```

Output for Input2.txt, heuristic 2

```
2 6 0
1 3 4
7 5 8

1 2 3
4 5 6
7 8 0

10
24
L D R U L L D R D R
10 10 10 10 10 10 10 10 10 10 10
```

Output for Input3.txt, heuristic 1

```
5 4 3
2 6 7
1 8 0

1 2 3
4 5 6
7 8 0

22
1139
U L D R U U L L D D R U U R D L U L D R R D
12 12 14 14 16 18 20 20 20 20 20 20 20 20 20 22 22 22 22 22 22 22 22
```

Output for Input3.txt, heuristic 2

```
5 4 3
2 6 7
1 8 0

1 2 3
4 5 6
7 8 0

22
666
U L U R D D L U R U L L D D R U U L D D R R
14 14 14 16 18 18 20 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22
```

Output for Input4.txt, heuristic 1

```
8 7 3
0 4 5
6 2 1

1 2 3
4 5 6
7 8 0

23
988
U R D D R U L D L U U R D R D L L U U R D R D
17 17 19 19 19 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23
```

Output for Input4.txt, heuristic 2

```
8 7 3
0 4 5
6 2 1

1 2 3
4 5 6
7 8 0

23
251
U R D D R U L D L U U R D R D L L U U R D R D
21 19 21 23 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23
```

SOURCE CODE (contents of *solver.py*)

```
# Richard Messina rdm420 N13468622
# CS4613 Project 1 - 8 Puzzle Problem Solver
# Python 3.7.2

import argparse
from copy import deepcopy
from queue import PriorityQueue


class EightPuzzleQueue(PriorityQueue):
    # priority queue which prioritizes nodes by f(n) value
    def put_puzzle_node(self, node):
        if node is not None:
            super().put((node.fn_value, node))

    def get_puzzle_node(self):
        return super().get()[1]


class EightPuzzleNode(object):
    def __init__(self, positions, level, operation, fn, parent, ct,
generated_set):
        self.positions = positions
        self.level = level
        self.operation = operation
        self.fn_value = fn(self)
        self.parent = parent

        self._ct = ct
        self._fn = fn
        self._generated = generated_set

        # whenever this constructor is called, increment the total number
of nodes generated
        # and mark this node as generated
        ct(self)

    def __lt__(self, other):
        return self.fn_value < other.fn_value

    def __hash__(self):
        return self._hash_from_positions(self.positions)

    def _hash_from_positions(self, positions):
        return hash(tuple(positions.items()))

    def _find_piece(self, target_pos):
        for piece, pos in self.positions.items():
            if pos == target_pos:
                return piece
        return None

    def _is_valid_pos(self, pos):
```

```
        return 0 <= pos[0] <= 2 and 0 <= pos[1] <= 2

    # moves the blank position by the modifier specified
    def _move(self, operation, modifier):
        zero_pos = self.positions['0']
        new_zero_pos = (zero_pos[0] + modifier[0], zero_pos[1] +
modifier[1])

        if not self._is_valid_pos(new_zero_pos):
            return None

        swap_piece = self._find_piece(new_zero_pos)

        new_positions = deepcopy(self.positions)
        new_positions['0'] = new_zero_pos
        new_positions[swap_piece] = zero_pos

        if self._hash_from_positions(new_positions) in self._generated:
            return None

        # only generate the new node if the move is valid and the node has
not been generated yet
        return EightPuzzleNode(new_positions, self.level + 1,
                               operation, self._fn, self, self._ct,
self._generated)

    def up(self):
        return self._move('U', (0, -1))

    def down(self):
        return self._move('D', (0, 1))

    def left(self):
        return self._move('L', (-1, 0))

    def right(self):
        return self._move('R', (1, 0))


class EightPuzzleProblem(object):
    def __init__(self, infile, heuristic_fn):
        self._parse_input(infile)
        self._heuristic_fn = heuristic_fn
        self._generated = set()
        self._num_nodes_generated = 0

    def _ct(self, node):
        self._num_nodes_generated += 1
        self._mark_generated(node)

    def _parse_input(self, infile):
        with open(infile, 'r') as instream:
            data = instream.readlines()
```

```
            self._initial_state = [[p for p in row.split()] for row in
data[:3]]
            self._goal_state = [[p for p in row.split()] for row in
data[4:7]]

            self._goal_positions = self._get_positions(self._goal_state)

    # convert 2d array to hashmap of positions
    def _get_positions(self, state):
        positions = {}
        for row in range(len(state)):
            for col in range(len(state[row])):
                positions[state[row][col]] = (col, row)
        return positions

    # maintain hash set of generated nodes in order to avoid regeneration
    def _mark_generated(self, node):
        self._generated.add(hash(node))

    def _is_goal(self, node):
        for piece, pos in node.positions.items():
            if self._goal_positions[piece] != pos:
                return False
        return True

    def _manhattan_distance_sum(self, node):
        manhattan_distance_sum = 0
        for piece, pos in node.positions.items():
            if piece != '0':
                delta_x = pos[0] - self._goal_positions[piece][0]
                delta_y = pos[1] - self._goal_positions[piece][1]
                manhattan_distance_sum += abs(delta_x) + abs(delta_y)
        return manhattan_distance_sum

    def _linear_conflicts(self, node):
        # maintain linear conflicts in hash set in order to avoid recounts
        linear_conflicts = set()
        for piece, pos in node.positions.items():
            if piece != '0':
                if self._goal_positions[piece][0] == pos[0]:
                    # check for column conflicts
                    current_diff = pos[1] - self._goal_positions[piece][1]
                    for col in range(len(self._goal_state[pos[1]])):
                        other_piece = self._goal_state[pos[1]][col]
                        if other_piece != '0' and other_piece != piece \
                                and node.positions[other_piece][0] ==
pos[0]:
                            other_diff = node.positions[other_piece][1] \
                                - self._goal_positions[other_piece][1]
                            # if the difference of the distances of these
tiles from their
                            # goal positions is >= 2, count as linear
conflict
                            if abs(current_diff - other_diff) >= 2:
```

```
                                linear_conflicts.add(tuple(sorted((piece,
other_piece))))
                if self._goal_positions[piece][1] == pos[1]:
                    # check for row conflicts
                    current_diff = pos[0] - self._goal_positions[piece][0]
                    for other_piece in self._goal_state[pos[1]]:
                        if other_piece != '0' and other_piece != piece \
                                and node.positions[other_piece][1] ==
pos[1]:
                            other_diff = node.positions[other_piece][0] \
                                - self._goal_positions[other_piece][0]
                            # if the difference of the distances of these
tiles from their
                            # goal positions is >= 2, count as linear
conflict
                            if abs(current_diff - other_diff) >= 2:
                                linear_conflicts.add(tuple(sorted((piece,
other_piece))))
        return len(linear_conflicts)

    def _a_star_fn(self):
        if self._heuristic_fn == '1':
            def fn(node):
                return node.level + self._manhattan_distance_sum(node)
        elif self._heuristic_fn == '2':
            def fn(node):
                return node.level + self._manhattan_distance_sum(node) \
                    + (self._linear_conflicts(node) * 2)
        else:
            raise Exception('The value "1" or "2" must be passed for the
function identifier.')
        return fn

    def solve_to_file(self, outfile):
        q = EightPuzzleQueue()

        # root node
        current_node =
EightPuzzleNode(self._get_positions(self._initial_state),
                                       0, None, self._a_star_fn(), None,
self._ct, self._generated)

        # generate child states if valid moves and place into priority
queue based on f(n) value
        while not self._is_goal(current_node):
            q.put_puzzle_node(current_node.up())
            q.put_puzzle_node(current_node.down())
            q.put_puzzle_node(current_node.left())
            q.put_puzzle_node(current_node.right())

            # expand node on frontier with lowest f(n) value
            current_node = q.get_puzzle_node()

        # gather final results
```

```
        soln_level = current_node.level
        operations = []
        fn_values = [current_node.fn_value]
        while current_node.operation is not None:
            operations = [current_node.operation] + operations
            fn_values = [current_node.fn_value] + fn_values
            current_node = current_node.parent

        # write results to specified file
        with open(outfile, 'w') as outstream:
            for initial_row in self._initial_state:
                print(*initial_row, sep=' ', file=outstream)
            outstream.write('\n')

            for goal_row in self._goal_state:
                print(*goal_row, file=outstream)
            outstream.write('\n')

            print(soln_level, file=outstream)
            print(self._num_nodes_generated, file=outstream)
            print(*operations, file=outstream)
            print(*fn_values, file=outstream)


if __name__ == '__main__':
    parser = argparse.ArgumentParser()

    parser.add_argument('-i', '--input', help='Input file name',
required=True)
    parser.add_argument('-o', '--output', help='Output file name',
required=True)
    parser.add_argument('-f', '--function', help='The heuristic function
to use. "1"=sum of ' +
                        'Manhattan distances of tiles from their goal
position. "2"= sum of ' +
                        'Manhattan distances + 2 * # linear conflicts',
required=True)

    args = parser.parse_args()

    EightPuzzleProblem(args.input,
args.function).solve_to_file(args.output)
```