8 Puzzle Solver

```
Apoorva Kaushik
March 31, 2022
CS 4613 Project 1
```

Running the solver

1. Download 8_puzzle_solver.py
2. Place it in a directory with the formatted input text files containing the 8 puzzles that are being solved. It is set up to automatically run the following three puzzles. (This can be changed by altering the main function of the code)
   - Input1.txt
   - Input2.txt
   - Input3.txt
3. Open your shell in the directory with all files.
4. Run the following command in the shell using Python 3. (An example is shown below.) Alternatively run the Python file through your preferred IDE.

   `python3 8_puzzle_solver.py`

5. Files containing the solutions will be generated as follows. Where `#` will be the number of the input passed to the program. The `h1` shows that the *sum of Manhattan distances of the tiles from their goal positions* was used as the heuristic and `h2` shows that *Nilsson's sequence score* was used as the heuristic.
   - Output`#`h1.txt
   - Output`#`h2.txt

Outputs

```
Output1h1.txt
4 1 6
8 3 5
2 0 7

8 4 6
0 1 5
2 3 7


4
```

```
11
U U L D
4 4 4 4 4
```

Output1h2.txt

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

8 4 6
0 1 5
2 3 7


4
15
U U L D
31 19 22 22 4
```

Output2h1.txt

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

1 2 0
7 5 3
4 8 6


12
56
L D R U L L D R D R U U
10 10 12 12 12 12 12 12 12 12 12 12 12
```

Output2h2.txt

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

1 2 0
7 5 3
```

```
4 8 6


12
54
L D R U L L D R D R U U
49 43 45 45 45 27 45 39 27 30 30 30 12
```

Output3h1.txt

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

1 2 3
4 0 7
6 5 8


25
1336
U R D L D R R U L D L U U R D D L U U R D D R U L
19 19 19 21 21 21 23 23 23 23 23 23 25 25 25 25 25 25 25 25 25 25 25 25 25
25
```

Output3h2.txt

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

1 2 3
4 0 7
6 5 8


25
96
U R D R D L L U R R D L L U U R D R D L L U U R D
58 52 52 51 48 54 50 56 50 43 46 46 52 52 52 52 46 43 40 40 40 40 40 40 34
25
```

Source Code

```
"""
Apoorva Kaushik
8 Puzzle Solver
CS 4613 Project 1
"""
import heapq
import copy

MOVES = ["L", "R", "U", "D"]
NILLSON_ORDER = [(0,0), (0,1), (0,2), (1,2), (2,2), (2,1), (2,0),(1,0),
(0,0)]
VALID = [0, 1, 2]
COLS = 3

class Node:
    def __init__(self, arrangement, parent = None, is_manhattan = True,
last_move = None, goal_node = None, path_cost = 0):
        """
        Parameters:
            arrangement: 2D list representing the puzzle arrangement
            parent: Node object representing the parent of this node
            is_manhattan: boolean representing whether to use manhattan
distance or S value
            last_move: string representing the last move made to get to
this node
            goal_node: Node object representing the goal state of the
puzzle
            path_cost: integer representing the path cost of this node
        Return: None
        Function initializes the node with the given parameters
        """
        self.puzzle = arrangement
        self.path_cost = path_cost
        self.last_move = last_move
        self.parent = parent
        if goal_node != None:
            if is_manhattan:
                heuristic = self.manhattan_distance(goal_node)
            else:
                heuristic = self.manhattan_distance(goal_node) + (3 *
self.S_value(goal_node))
            self.fn_value = path_cost + heuristic

    def __eq__(self, other = None):
        """
        Parameters:
            other: Node to compare with self
        Return: True or False equality based on the puzzle arrangement
        """
        if other == None or self == None:
            return False
        else:
            return self.puzzle == other.puzzle
```

```
    def __ne__(self, other):
        """
        Parameters:
            other: Node to compare with self
        Return: True or False equality based on the puzzle arrangement
        """
        return not(self == other)

    def __lt__(self, other):
        """
        Parameters:
            other: Node to compare with self
        Return: True or False equality based on the evaluiation function
fn value
        """
        return self.fn_value < other.fn_value

    def make_dict(self):
        """
        Parameters:
            None
        Return: dictionary representation of puzzle -> {number : index
position as tuple (x, y)}
        """
        coordinate_dict = {}
        for x, row in enumerate(self.puzzle):
            for y, value in enumerate(row):
                coordinate_dict[value] = (x, y)
        return coordinate_dict

    def manhattan_distance(self, goal):
        """
        Parameters:
            goal: Node object representing the goal state of the puzzle
        Return: manhattan distance between this node and the goal node
        """
        # calculate the manhattan distance between self Node and goal Node
        manhattan_distance = 0
        current_vals = self.make_dict()
        goal_vals = goal.make_dict()
        for key in current_vals:
            curr_x, curr_y = current_vals[key]
            goal_x, goal_y = goal_vals[key]
            if key != "0":
                manhattan_distance += abs(goal_x - curr_x) + abs(goal_y -
curr_y)
            # print("{} at {},{} distance: {}".format(key, curr_x, curr_y,
single_distance))
        #print("manhattan distance: ", manhattan_distance)
        return manhattan_distance

    def S_value(self, goal):
        """
```

```
        Parameters:
            goal: Node object representing the goal state of the puzzle
        Return: sequence score between current nodes state and the goal
state
        """
        goal_vals = goal.make_dict()
        nillson_score = 0
        for i in range(len(NILLSON_ORDER)-1):
            current_coord = NILLSON_ORDER[i]
            current_value = self.puzzle[current_coord[0]]
[current_coord[1]]
            successor_coord = NILLSON_ORDER[i+1]
            successor_value = self.puzzle[successor_coord[0]]
[successor_coord[1]]

            goal_location = goal_vals[current_value]
            if goal_location != (1,1):
                goal_successor_coord =
NILLSON_ORDER[NILLSON_ORDER.index(goal_location)+1]
                goal_successor_value =
goal.puzzle[goal_successor_coord[0]][goal_successor_coord[1]]
                if goal_successor_value != successor_value:
                    nillson_score += 2
                # print("current {} successor {} expected successor
{}".format(current_value, successor_value, goal_successor_value))
            # else:
            #     print("whoops")
            #     print("current {} successor {}".format(current_value,
successor_value))

        # Check the center tile
        if self.puzzle[1][1] != goal.puzzle[1][1]:
            nillson_score += 1
        # print("S_score",nillson_score)
        return nillson_score

class Puzzle:
    def __init__(self, input_filename, output_filename, is_manhattan):
        """
        Parameters:
            input_filename: name of the input file
            output_filename: name of the output file
            is_manhattan: True if manhattan distance is to be used, False
if S value is to be used
        Return: None
        Function initializes the Puzzle object based on the input file
given and heuristic used to
        """
        self.current = None
        self.goal = None
        self.reached = []
        self.frontier = []
        self.num_nodes = 1 # this is the root node
        self.is_manhattan = is_manhattan
```

```
        self.get_input(input_filename, output_filename)
        self.select_next_move(output_filename)

    def seen_node(self, new_node):
        """
        Parameters:
            new_node: Node object to check if it has been seen before
        Return: True or False if the node has been seen before
        """
        seen = False
        for node in self.reached:
            # same arrangement and the old node has a lower path cost
            if node == new_node and node < new_node:
                return True
        return False

    def try_moves(self):
        """
        Parameters:
            None
        Return: None
        This function will try all possible moves and add them to the
frontier if they have not been seen
        """
        # Try all moves L, R, U, D
        row, col = self.current.make_dict()["0"]
        # print("Current")
        # print(self.current)
        for move in MOVES:
            new_node = None
            original = copy.deepcopy(self.current.puzzle)
            if move == "L" and col - 1 in VALID:
                original[row][col], original[row][col - 1] = original[row]
[col - 1], original[row][col]
                new_node = Node(original, self.current, self.is_manhattan,
move, self.goal, self.current.path_cost + 1)
            elif move == "R" and col + 1 in VALID:
                original[row][col], original[row][col + 1] = original[row]
[col + 1], original[row][col]
                new_node = Node(original, self.current, self.is_manhattan,
move, self.goal, self.current.path_cost + 1)
            elif move == "U" and row - 1 in VALID:
                original[row][col], original[row - 1][col] = original[row
- 1][col], original[row][col]
                new_node = Node(original, self.current, self.is_manhattan,
move, self.goal, self.current.path_cost + 1)
            elif move == "D" and row + 1 in VALID:
                original[row][col], original[row + 1][col] = original[row
+ 1][col], original[row][col]
                new_node = Node(original, self.current, self.is_manhattan,
move, self.goal, self.current.path_cost + 1)
            #print(new_node)
            # NOTE: that a new node is only created when the move in valid
            # NOTE: Only add when not seen
```

```
            if new_node != None:
                if self.seen_node(new_node) == False:
                    heapq.heappush(self.frontier, new_node)
                    self.reached.append(new_node)
                    self.num_nodes += 1

    def trace_back(self):
        """
        Parameters:
            self: Puzzle object
        Return:
            moves: string of moves to get to the goal state
            f_vals: string of f values for each node
        This function will trace back the path from the goal node to the
root node.
        """
        moves = ""
        f_vals = ""
        curr_node = self.current
        while curr_node.parent != None:
            moves = curr_node.last_move + " " + moves
            f_vals = str(curr_node.fn_value) + " " + f_vals
            # print(curr_node)
            curr_node = curr_node.parent
        f_vals = str(curr_node.fn_value) + " " + f_vals
        # print(moves)
        # print(f_vals)
        return moves, f_vals

    def select_next_move(self, output_filename):
        """
        Parameters:
            output_filename: string representing the name of the output
file
        Return: None
        This function will select the next move to make based on the
heuristic function until a solution is found
        """
        output_file = open(output_filename, 'a')

        self.try_moves() # expand the first node
        not_solved = True
        while not_solved:
            if len(self.frontier) == 0:
                break
            self.current = heapq.heappop(self.frontier)
            if self.current == self.goal:
                not_solved = False
                print("We solved the puzzle!")
                moves, f_vals = self.trace_back()
                output_file.write(str(self.current.path_cost) + "\n")
                output_file.write(str(self.num_nodes) + "\n")
                output_file.write(moves + "\n")
                output_file.write(f_vals)
```

```
            else:
                self.try_moves()
        output_file.close()

    def get_input(self, input_filename, output_filename):
        """
        Parameters:
            input_filename: string representing the name of the input file
            output_filename: string representing the name of the output
file
        Return: None
        This function will read the input file and create the initial and
goal nodes
        """
        try:
            input_file = open(input_filename, 'r')
        except FileNotFoundError:
            print("{} not found".format(input_filename))
            return
        output_file = open(output_filename, 'w')
        is_goal = False
        goal = []
        start = []
        for line in input_file:
            output_file.write(line)
            if line == "\n":
                is_goal = True
            else:
                line = line.strip().split(" ")
                if is_goal:
                    goal.append(line)
                else:
                    start.append(line)
        self.goal = Node(goal)
        self.current = Node(start, None, self.is_manhattan, None,
self.goal, 0)
        output_file.write("\n\n")
        input_file.close()
        output_file.close()

def main():
    """
    Parameters:
        None
    Return: None
    This function will run the program with the three input text files
each with two heuristics
    """
    Puzzle("Input1.txt", "Output1h1.txt", True)
    Puzzle("Input1.txt", "Output1h2.txt", False)

    Puzzle("Input2.txt", "Output2h1.txt", True)
    Puzzle("Input2.txt", "Output2h2.txt", False)
```

```
    Puzzle("Input3.txt", "Output3h1.txt", True)
    Puzzle("Input3.txt", "Output3h2.txt", False)

if __name__ == "__main__":
    main()
```