```python
import numpy as np
import time
class Node():
    def __init__(self, state, parent, action, depth, step_cost, path_cost, heuristic_cost):
        self.state = state
        self.parent = parent  # Starting with the parent node
        self.action = action  # move up, left, down, right
        self.depth = depth  # depth of the node in the tree
        self.step_cost = step_cost  # g(n), the cost to take the step
        self.path_cost = path_cost  # accumulated g(n), the cost to reach the current node
        self.heuristic_cost = heuristic_cost  # h(n), cost to reach goal state from the current node

        # children node
        self.move_up = None
        self.move_left = None
        self.move_down = None
        self.move_right = None

    # see if moving down is valid
    def try_move_down(self):
        # index of the empty tile
        zero_index = [i[0] for i in np.where(self.state == 0)]
        if zero_index[0] == 0:
            return False
        else:
            up_value = self.state[zero_index[0] - 1, zero_index[1]]  # value of the upper tile
            new_state = self.state.copy()
            new_state[zero_index[0], zero_index[1]] = up_value
            new_state[zero_index[0] - 1, zero_index[1]] = 0
            return new_state, up_value

    # see if moving right is valid
    def try_move_right(self):
        zero_index = [i[0] for i in np.where(self.state == 0)]
        if zero_index[1] == 0:
            return False
        else:
            left_value = self.state[zero_index[0], zero_index[1] - 1]  # value of the left tile
            new_state = self.state.copy()
            new_state[zero_index[0], zero_index[1]] = left_value
            new_state[zero_index[0], zero_index[1] - 1] = 0
            return new_state, left_value

    # see if moving up is valid
    def try_move_up(self):
        zero_index = [i[0] for i in np.where(self.state == 0)]
        if zero_index[0] == 2:
            return False
```

```python
    else:
        lower_value = self.state[zero_index[0] + 1, zero_index[1]]  # value of the lower tile
        new_state = self.state.copy()
        new_state[zero_index[0], zero_index[1]] = lower_value
        new_state[zero_index[0] + 1, zero_index[1]] = 0
        return new_state, lower_value

# see if moving left is valid
def try_move_left(self):
    zero_index = [i[0] for i in np.where(self.state == 0)]
    if zero_index[1] == 2:
        return False
    else:
        right_value = self.state[zero_index[0], zero_index[1] + 1]  # value of the right tile
        new_state = self.state.copy()
        new_state[zero_index[0], zero_index[1]] = right_value
        new_state[zero_index[0], zero_index[1] + 1] = 0
        return new_state, right_value

# return user specified heuristic cost
def get_h_cost(self, new_state, goal_state, heuristic_function, path_cost, depth):
    if heuristic_function == 'num_misplaced':
        return self.h_misplaced_cost(new_state, goal_state)
    elif heuristic_function == 'manhattan':
        return self.h_manhattan_cost(new_state, goal_state)
    # since this game is made unfair by setting the step cost as the value of the tile being moved
    # to make it fair,  all the step cost are 1
    # made it a best-first-search with manhattan heuristic function
    elif heuristic_function == 'fair_manhattan':
        return self.h_manhattan_cost(new_state, goal_state) - path_cost + depth

# return heuristic cost: number of misplaced tiles
def h_misplaced_cost(self, new_state, goal_state):
    cost = np.sum(new_state != goal_state) - 1  # minus 1 to exclude the empty tile
    if cost > 0:
        return cost
    else:
        return 0  # when all tiles matches

# return heuristic cost: sum of Manhattan distance to reach the goal state
def h_manhattan_cost(self, new_state, goal_state):
    current = new_state
    # digit and coordinates they are supposed to be
    goal_position_dic = {1: (0, 0), 2: (0, 1), 3: (0, 2), 8: (1, 0), 0: (1, 1), 4: (1, 2), 7: (2, 0), 6: (2, 1),
                         5: (2, 2)}
    sum_manhattan = 0
    for i in range(3):
        for j in range(3):
```

```python
            if current[i, j] != 0:
                sum_manhattan += sum(abs(a - b) for a, b in zip((i, j), goal_position_dic[current[i, j]]))
    return sum_manhattan

# once the goal node is found, trace back to the root node and print out the path
def print_path(self):
    # create FILO stacks to place the trace
    state_trace = [self.state]
    action_trace = [self.action]
    depth_trace = [self.depth]
    step_cost_trace = [self.step_cost]
    path_cost_trace = [self.path_cost]
    heuristic_cost_trace = [self.heuristic_cost]

    # add node information as tracing back up the tree
    while self.parent:
        self = self.parent

        state_trace.append(self.state)
        action_trace.append(self.action)
        depth_trace.append(self.depth)
        step_cost_trace.append(self.step_cost)
        path_cost_trace.append(self.path_cost)
        heuristic_cost_trace.append(self.heuristic_cost)

    # print out the path
    step_counter = 0
    while state_trace:
        print('step', step_counter)
        print(state_trace.pop())
        print('action=', action_trace.pop(), ', depth=', str(depth_trace.pop()), \
              ', step cost=', str(step_cost_trace.pop()), ', total_cost=', \
              str(path_cost_trace.pop() + heuristic_cost_trace.pop()), '\n')

        step_counter += 1

def breadth_first_search(self, goal_state):
    start = time.time()

    queue = [self]  # queue of found but unvisited nodes, FIFO
    queue_num_nodes_popped = 0  # number of nodes popped off the queue, measuring time performance
    queue_max_length = 1  # max number of nodes in the queue, measuring space performance

    depth_queue = [0]  # queue of node depth
    path_cost_queue = [0]  # queue for path cost
    visited = set([])  # record visited states

    while queue:
```

```
        # update maximum length of the queue
        if len(queue) > queue_max_length:
            queue_max_length = len(queue)

        current_node = queue.pop(0)  # select and remove the first node in the queue
        queue_num_nodes_popped += 1

        current_depth = depth_queue.pop(0)  # select and remove the depth for current node
        current_path_cost = path_cost_queue.pop(0)  # # select and remove the path cost for reaching current node
        visited.add(
            tuple(current_node.state.reshape(1, 9)[0]))  # avoid repeated state, which is represented as a tuple

        # when the goal state is found, trace back to the root node and print out the path
        if np.array_equal(current_node.state, goal_state):
            current_node.print_path()
            print('BFS:')
            print('Time-Execution:', str(queue_num_nodes_popped), 'nodes popped off the queue.')
            print('Space-Execution:', str(queue_max_length), 'nodes in the queue at its max.')
            print('Time-complexity: %0.2fs' % (time.time() - start))
            return True

        else:           # see if moving upper tile down is a valid move
         if current_node.try_move_down():
            new_state, up_value = current_node.try_move_down()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                # create a new child node
                current_node.move_down = Node(state=new_state, parent=current_node, action='down',
                                              depth=current_depth + 1, \
                                              step_cost=up_value, path_cost=current_path_cost + up_value,
                                              heuristic_cost=0)
                queue.append(current_node.move_down)
                depth_queue.append(current_depth + 1)
                path_cost_queue.append(current_path_cost + up_value)

        # see if moving left tile to the right is a valid move
        if current_node.try_move_right():
            new_state, left_value = current_node.try_move_right()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                # create a new child node
                current_node.move_right = Node(state=new_state, parent=current_node, action='right',
                                               depth=current_depth + 1, \
                                               step_cost=left_value, path_cost=current_path_cost + left_value,
                                               heuristic_cost=0)
                queue.append(current_node.move_right)
                depth_queue.append(current_depth + 1)
                path_cost_queue.append(current_path_cost + left_value)
```

```
        # see if moving lower tile up is a valid move
        if current_node.try_move_up():
            new_state, lower_value = current_node.try_move_up()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                # create a new child node
                current_node.move_up = Node(state=new_state, parent=current_node, action='up',
                                            depth=current_depth + 1, \
                                            step_cost=lower_value, path_cost=current_path_cost + lower_value,
                                            heuristic_cost=0)
                queue.append(current_node.move_up)
                depth_queue.append(current_depth + 1)
                path_cost_queue.append(current_path_cost + lower_value)

        # see if moving right tile to the left is a valid move
        if current_node.try_move_left():
            new_state, right_value = current_node.try_move_left()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                # create a new child node
                current_node.move_left = Node(state=new_state, parent=current_node, action='left',
                                              depth=current_depth + 1, \
                                              step_cost=right_value, path_cost=current_path_cost + right_value,
                                              heuristic_cost=0)
                queue.append(current_node.move_left)
                depth_queue.append(current_depth + 1)
                path_cost_queue.append(current_path_cost + right_value)

def depth_first_search(self, goal_state):
    start = time.time()

    queue = [self]  # queue of found but unvisited nodes, FILO
    queue_num_nodes_popped = 0  # number of nodes popped off the queue, measuring time performance
    queue_max_length = 1  # max number of nodes in the queue, measuring space performance

    depth_queue = [0]  # queue of node depth
    path_cost_queue = [0]  # queue for path cost
    visited = set([])  # record visited states

    while queue:
        # update maximum length of the queue
        if len(queue) > queue_max_length:
            queue_max_length = len(queue)

        current_node = queue.pop(0)  # select and remove the first node in the queue
        queue_num_nodes_popped += 1
```

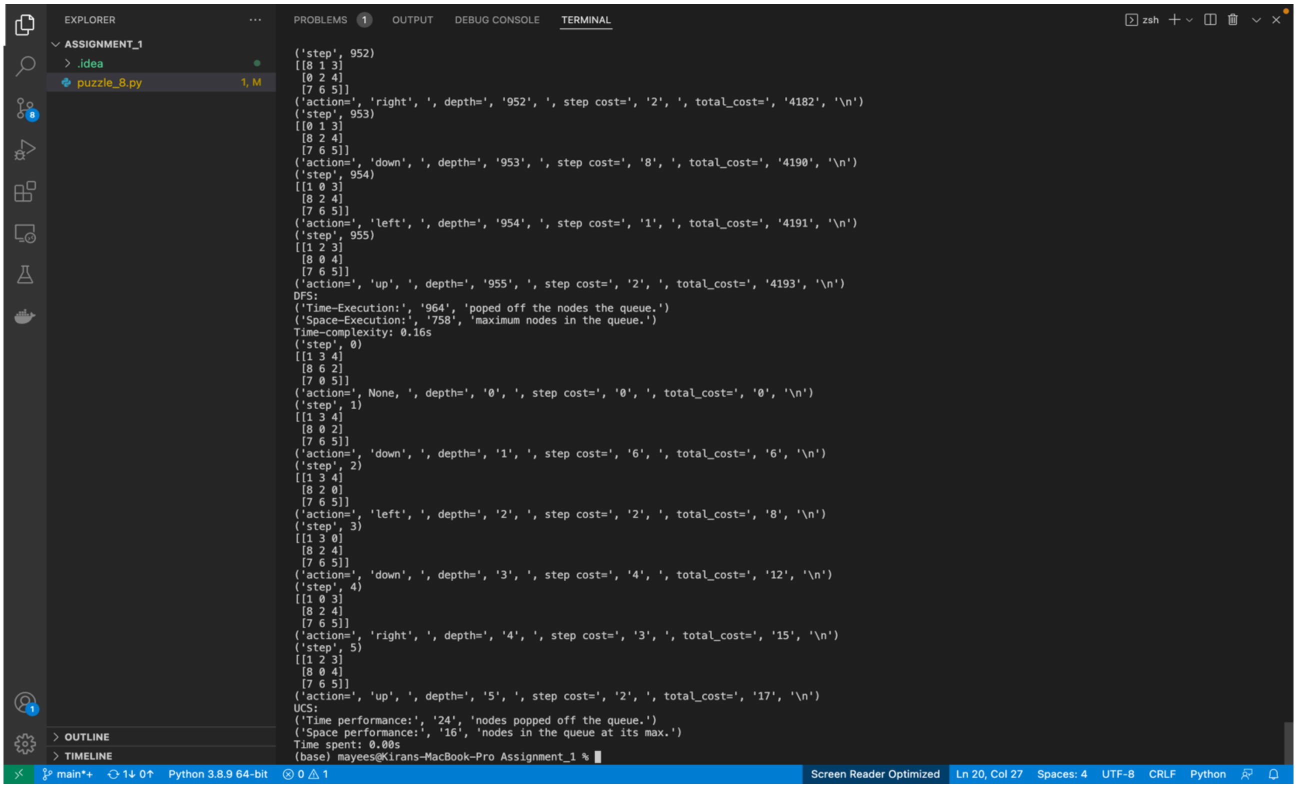
```
    current_depth = depth_queue.pop(0)  # select and remove the depth for current node
    current_path_cost = path_cost_queue.pop(0)  # # select and remove the path cost for reaching current node
    visited.add(tuple(current_node.state.reshape(1, 9)[0]))  # add state, which is represented as a tuple

    # when the goal state is found, trace back to the root node and print out the path
    if np.array_equal(current_node.state, goal_state):
        current_node.print_path()
        print('DFS:')
        print('Time-Execution:', str(queue_num_nodes_popped), 'poped off the nodes the queue.')
        print('Space-Execution:', str(queue_max_length), 'maximum nodes in the queue.')
        print('Time-complexity: %0.2fs' % (time.time() - start))
        return True

    else:
        # see if moving upper tile down is a valid move
        if current_node.try_move_down():
            new_state, up_value = current_node.try_move_down()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                # create a new child node
                current_node.move_down = Node(state=new_state, parent=current_node, action='down',
                                              depth=current_depth + 1, \
                                              step_cost=up_value, path_cost=current_path_cost + up_value,
                                              heuristic_cost=0)
                queue.insert(0, current_node.move_down)
                depth_queue.insert(0, current_depth + 1)
                path_cost_queue.insert(0, current_path_cost + up_value)

    # see if moving left tile to the right is a valid move
    if current_node.try_move_right():
        new_state, left_value = current_node.try_move_right()
        # check if the resulting node is already visited
        if tuple(new_state.reshape(1, 9)[0]) not in visited:
            # create a new child node
            current_node.move_right = Node(state=new_state, parent=current_node, action='right',
                                           depth=current_depth + 1, \
                                           step_cost=left_value, path_cost=current_path_cost + left_value,
                                           heuristic_cost=0)
            queue.insert(0, current_node.move_right)
            depth_queue.insert(0, current_depth + 1)
            path_cost_queue.insert(0, current_path_cost + left_value)

    # see if moving lower tile up is a valid move
    if current_node.try_move_up():
        new_state, lower_value = current_node.try_move_up()
        # check if the resulting node is already visited
        if tuple(new_state.reshape(1, 9)[0]) not in visited:
            # create a new child node
```

```python
                current_node.move_up = Node(state=new_state, parent=current_node, action='up',
                                            depth=current_depth + 1, \
                                            step_cost=lower_value, path_cost=current_path_cost + lower_value,
                                            heuristic_cost=0)
                queue.insert(0, current_node.move_up)
                depth_queue.insert(0, current_depth + 1)
                path_cost_queue.insert(0, current_path_cost + lower_value)

        # see if moving right tile to the left is a valid move
        if current_node.try_move_left():
            new_state, right_value = current_node.try_move_left()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                # create a new child node
                current_node.move_left = Node(state=new_state, parent=current_node, action='left',
                                              depth=current_depth + 1, \
                                              step_cost=right_value, path_cost=current_path_cost + right_value,
                                              heuristic_cost=0)
                queue.insert(0, current_node.move_left)
                depth_queue.insert(0, current_depth + 1)
                path_cost_queue.insert(0, current_path_cost + right_value)

# An extension of BFS that's guided by a prioritized queue based on path cost
def uniform_cost_search(self, goal_state):
    start = time.time()

    queue = [
        (self, 0)]  # queue of (found but unvisited nodes, path cost), ordered by path cost(accumulated step cost)
    queue_num_nodes_popped = 0  # number of nodes popped off the queue, measuring time performance
    queue_max_length = 1  # max number of nodes in the queue, measuring space performance

    depth_queue = [(0, 0)]  # queue of node depth, (depth, path cost)
    path_cost_queue = [0]  # queue for path cost
    visited = set([])  # record visited states

    while queue:
        # sort queue based on path cost, in ascending order
        queue = sorted(queue, key=lambda x: x[1])
        depth_queue = sorted(depth_queue, key=lambda x: x[1])
        path_cost_queue = sorted(path_cost_queue, key=lambda x: x)

        # update maximum length of the queue
        if len(queue) > queue_max_length:
            queue_max_length = len(queue)

        current_node = queue.pop(0)[0]  # select and remove the first node in the queue

        queue_num_nodes_popped += 1
```

```python
current_depth = depth_queue.pop(0)[0]  # select and remove the depth for current node
current_path_cost = path_cost_queue.pop(0)  # # select and remove the path cost for reaching current node
visited.add(
    tuple(current_node.state.reshape(1, 9)[0]))  # avoid repeated state, which is represented as a tuple

# when the goal state is found, trace back to the root node and print out the path
if np.array_equal(current_node.state, goal_state):
    current_node.print_path()
    print('UCS:')
    print('Time-Execution:', str(queue_num_nodes_popped), 'nodes popped off the queue.')
    print('Space-Execution:', str(queue_max_length), 'nodes in the queue at its max.')
    print('Time-complecity: %0.2fs' % (time.time() - start))
    return True

else:
    # see if moving upper tile down is a valid move
    if current_node.try_move_down():
        new_state, up_value = current_node.try_move_down()
        # check if the resulting node is already visited
        if tuple(new_state.reshape(1, 9)[0]) not in visited:
            # create a new child node
            current_node.move_down = Node(state=new_state, parent=current_node, action='down',
                                          depth=current_depth + 1, \
                                          step_cost=up_value, path_cost=current_path_cost + up_value,
                                          heuristic_cost=0)
            queue.append((current_node.move_down, current_path_cost + up_value))
            depth_queue.append((current_depth + 1, current_path_cost + up_value))
            path_cost_queue.append(current_path_cost + up_value)

# see if moving left tile to the right is a valid move
if current_node.try_move_right():
    new_state, left_value = current_node.try_move_right()
    # check if the resulting node is already visited
    if tuple(new_state.reshape(1, 9)[0]) not in visited:
        # create a new child node
        current_node.move_right = Node(state=new_state, parent=current_node, action='right',
                                       depth=current_depth + 1, \
                                       step_cost=left_value, path_cost=current_path_cost + left_value,
                                       heuristic_cost=0)
        queue.append((current_node.move_right, current_path_cost + left_value))
        depth_queue.append((current_depth + 1, current_path_cost + left_value))
        path_cost_queue.append(current_path_cost + left_value)

# see if moving lower tile up is a valid move
if current_node.try_move_up():
    new_state, lower_value = current_node.try_move_up()
    # check if the resulting node is already visited
    if tuple(new_state.reshape(1, 9)[0]) not in visited:
```

```
                # create a new child node
                current_node.move_up = Node(state=new_state, parent=current_node, action='up',
                                            depth=current_depth + 1, \
                                            step_cost=lower_value, path_cost=current_path_cost + lower_value,
                                            heuristic_cost=0)
                queue.append((current_node.move_up, current_path_cost + lower_value))
                depth_queue.append((current_depth + 1, current_path_cost + lower_value))
                path_cost_queue.append(current_path_cost + lower_value)

        # see if moving right tile to the left is a valid move
        if current_node.try_move_left():
            new_state, right_value = current_node.try_move_left()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                # create a new child node
                current_node.move_left = Node(state=new_state, parent=current_node, action='left',
                                              depth=current_depth + 1, \
                                              step_cost=right_value, path_cost=current_path_cost + right_value,
                                              heuristic_cost=0)
                queue.append((current_node.move_left, current_path_cost + right_value))
                depth_queue.append((current_depth + 1, current_path_cost + right_value))
                path_cost_queue.append(current_path_cost + right_value)

test = np.array([1,2,3,8,6,4,7,5,0]).reshape(3,3)
easy = np.array([1,3,4,8,6,2,7,0,5]).reshape(3,3)
medium = np.array([2,8,1,0,4,3,7,6,5]).reshape(3,3)
hard = np.array([5,6,7,4,0,8,3,2,1]).reshape(3,3)

initial_state = easy
goal_state = np.array([1,2,3,8,0,4,7,6,5]).reshape(3,3)
print (initial_state,'\n')
print (goal_state)

root_node = Node(state=initial_state,parent=None,action=None,depth=0,step_cost=0,path_cost=0,heuristic_cost=0)
# search level by level with queue
root_node.breadth_first_search(goal_state)
# search as far as possible along each branch before backtracking, using stack
root_node.depth_first_search(goal_state)
# search based on path cost, using priority queue
root_node.uniform_cost_search(goal_state)
```

Output:


EXPLORER
ASSIGNMENT_1
.idea
puzzle_8.py
1, M
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL
zsh
('step', 952)
[[8 1 3]
 [0 2 4]
 [7 6 5]]
('action=', 'right', ', depth=', '952', ', step cost=', '2', ', total_cost=', '4182', '\n')
('step', 953)
[[0 1 3]
 [8 2 4]
 [7 6 5]]
('action=', 'down', ', depth=', '953', ', step cost=', '8', ', total_cost=', '4190', '\n')
('step', 954)
[[1 0 3]
 [8 2 4]
 [7 6 5]]
('action=', 'left', ', depth=', '954', ', step cost=', '1', ', total_cost=', '4191', '\n')
('step', 955)
[[1 2 3]
 [8 0 4]
 [7 6 5]]
('action=', 'up', ', depth=', '955', ', step cost=', '2', ', total_cost=', '4193', '\n')
DFS:
('Time-Execution:', '964', 'poped off the nodes the queue.')
('Space-Execution:', '758', 'maximum nodes in the queue.')
Time-complexity: 0.16s
('step', 0)
[[1 3 4]
 [8 6 2]
 [7 0 5]]
('action=', None, ', depth=', '0', ', step cost=', '0', ', total_cost=', '0', '\n')
('step', 1)
[[1 3 4]
 [8 0 2]
 [7 6 5]]
('action=', 'down', ', depth=', '1', ', step cost=', '6', ', total_cost=', '6', '\n')
('step', 2)
[[1 3 4]
 [8 2 0]
 [7 6 5]]
('action=', 'left', ', depth=', '2', ', step cost=', '2', ', total_cost=', '8', '\n')
('step', 3)
[[1 3 0]
 [8 2 4]
 [7 6 5]]
('action=', 'down', ', depth=', '3', ', step cost=', '4', ', total_cost=', '12', '\n')
('step', 4)
[[1 0 3]
 [8 2 4]
 [7 6 5]]
('action=', 'right', ', depth=', '4', ', step cost=', '3', ', total_cost=', '15', '\n')
('step', 5)
[[1 2 3]
 [8 0 4]
 [7 6 5]]
('action=', 'up', ', depth=', '5', ', step cost=', '2', ', total_cost=', '17', '\n')
UCS:
('Time performance:', '24', 'nodes popped off the queue.')
('Space performance:', '16', 'nodes in the queue at its max.')
Time spent: 0.00s
(base) mayees@Kirans-MacBook-Pro Assignment_1 %
OUTLINE
TIMELINE
main*+
1↓ 0↑
Python 3.8.9 64-bit
0 1
Screen Reader Optimized
Ln 20, Col 27
Spaces: 4
UTF-8
CRLF
Python