CS4613
Chris Gao, Jaeha Huh
4/11/21

Project 1: 15-Puzzle

How to run the project:

1. Have the input text file in the same directory as the python file
2. Run the python file with the specific input text file by changing the file name
    a.
```
def main():
    f = open("test.txt", "r")

    list_of_nums = []
    for line in f:
        line_strip = line.strip()
        line_list = line_strip.split()
        for char_num in line_list:
            list_of_nums.append(int(char
```
3. After the python program is completed, an output file should be produced named “output_test.txt”, or whatever the output file was decided to be
    a.
```
def print_path(self, initial_state, goal_state, queue_max_length):
    # 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]

    f = open("output_test.txt", "w")
```
4. The program could be ran on a terminal with no additional parameters, and it will produce both an output text file with more specific information being outputted on the terminal (such as runtime, number of nodes popped, visual aid of path to goal node, action taken, and depth level)
    a. Note that the program was tested using Pycharm
5. Reminder: This code was written in reverse by accident, so a down action (according to the code) produces a 3 because 3 is up (which happened because we were focused on

the neighboring numbers moving to the location of the zero, and not the zero location moving to another location)

Source codes:

```
#cs4613
#Chris Gao, Jaeha Huh
#4/11/21

import numpy as np
import time

class Node():
    def __init__(self, state, parent, action, depth, step_cost, path_cost, heuristic_cost):
        self.state = state # stores the current state of the node
        self.parent = parent # stores the parent of the node
        self.action = action # stores the action that the node will perform
        self.depth = depth # stores the depth of the tree
        self.step_cost = step_cost # stores the cost of each step (not used on this proj)
        self.path_cost = path_cost # stores the path cost to reach the node
        self.heuristic_cost = heuristic_cost # stores the heruistic cost of the current node

        self.move_up = None  # stores the reference to the next node that after blank is moved up
        self.move_left = None  # stores the reference to the next node that after blank is moved left
        self.move_down = None  # stores the reference to the next node that after blank is moved down
        self.move_right = None  # stores the reference to the next node that after blank is moved right
        self.move_up_right = None  # stores the reference to the next node that after blank is moved up right
        self.move_up_left = None  # stores the reference to the next node that after blank is moved up left
        self.move_down_right = None  # stores the reference to the next node that after blank is moved down right
        self.move_down_left = None  # stores the reference to the next node that after blank is moved down left

    def print_path(self, initial_state, goal_state, queue_max_length):
        # create FILO stacks to place the trace
        state_trace = [self.state] # stores the trace of the path to goal node
        action_trace = [self.action] # stores the action taken by the path to the goal node
        depth_trace = [self.depth] # stores the depth of each node to the goal node
        step_cost_trace = [self.step_cost] # stores the step cost of each node from path to goal node
        path_cost_trace = [self.path_cost] # stores the path cost of each node from the root to goal ndoe
        heuristic_cost_trace = [self.heuristic_cost] # stores the heuristic cost from root to goal node

        f = open("output_test.txt", "w")

        # add node information as tracing back up the tree
        while self.parent: # adds all the values from the current node by tracking parents
            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)

        for num in initial_state: # writing the initial state to the output file
            for inside in num:
                f.write(str(inside))
                f.write(" ")
            f.write("\n")

        f.write("\n")
```

```
    for num in goal_state: # writing the goal state to the output file
        for inside in num:
            f.write(str(inside))
            f.write(" ")
        f.write("Wn")

    f.write("Wn")

    f.write(str(len(step_cost_trace)-1))
    f.write("Wn")
    f.write(str(queue_max_length))
    f.write("Wn")

    action_list = []  # action list used for writing to file
    f_n_cost_list = []  # f_n cost used for writing to file

    # print out the path
    step_counter = 0
    while state_trace: # used to print out the information about path to goal node
        action = action_trace.pop()
        total_cost = str(path_cost_trace.pop() + heuristic_cost_trace.pop())

        if action:
            action_list.append(action[-1])
        f_n_cost_list.append(str(total_cost))

        print('step',step_counter)
        print(state_trace.pop())
        print('action=',action, ', depth=',str(depth_trace.pop()),
        ', step cost=',str(step_cost_trace.pop()), ', total_cost=',
        total_cost, 'Wn')

        step_counter += 1

    for action in action_list: # write the list of actions to the txt file
        f.write(action)
        f.write(" ")

    f.write("Wn")

    for f_n in f_n_cost_list: # write the list of f_n to the txt file
        f.write(f_n)
        f.write(" ")
```

```
def try_move_left(self):  # trys to move the left number to the blank position
    zero_index=[i[0] for i in np.where(self.state==0)] # obtains the zero index

    if zero_index[1] == 3: # edge of matrix
        return False

    else:
        left_val = self.state[zero_index[0], zero_index[1]+1] # perform swap between zero and left number
        new_state = self.state.copy()
        new_state[zero_index[0],zero_index[1]] = left_val
        new_state[zero_index[0], zero_index[1]+1] = 0
        return new_state, left_val  # returns the new_state after swap as well as the left_val

def try_move_up(self):
    zero_index=[i[0] for i in np.where(self.state==0)]  # obtains the zero index

    if zero_index[0] == 3:
        return False

    else:
        low_value = self.state[zero_index[0]+1, zero_index[1]] # perform swap between zero and up number
        new_state = self.state.copy()
        new_state[zero_index[0],zero_index[1]] = low_value
        new_state[zero_index[0]+1, zero_index[1]] = 0
        return new_state, low_value

# determines if space can move right
def try_move_right(self):
    zero_index=[i[0] for i in np.where(self.state==0)]  # obtains the zero index

    if zero_index[1] == 0: # if right of the matrix
        return False

    else:
        left_value = self.state[zero_index[0], zero_index[1]-1] # perform swap between zero and right number
        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

# determines whether moving down is possible
def try_move_down(self):
    zero_index=[i[0] for i in np.where(self.state==0)] # obtains the zero index

    if zero_index[0] == 0: # moving down is not possible
        return False

    else:
        up_value = self.state[zero_index[0]-1, zero_index[1]] # perform swap between zero and down number
        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
```

```
def try_move_up_left(self):
    zero_index=[i[0] for i in np.where(self.state==0)] # obtains the zero index

    if zero_index[0] == 3 or zero_index[1] == 3: # if space is top row or first column, return false
        return False

    else:
        up_left_val = self.state[zero_index[0]+1, zero_index[1]+1]  # perform swap between zero and up left number
        new_state = self.state.copy()
        new_state[zero_index[0],zero_index[1]] = up_left_val
        new_state[zero_index[0]+1, zero_index[1]+1] = 0
        return new_state, up_left_val

def try_move_up_right(self):
    zero_index=[i[0] for i in np.where(self.state==0)] # obtains the zero index

    if zero_index[0] == 3 or zero_index[0] == 3: # if space is top row or last column, return false
        return False

    else:
        up_right_val = self.state[zero_index[0]+1, zero_index[1]-1] # perform swap between zero and up right
        new_state = self.state.copy()
        new_state[zero_index[0],zero_index[1]] = up_right_val
        new_state[zero_index[0]+1, zero_index[1]-1] = 0
        return new_state, up_right_val


def try_move_down_right(self):
    zero_index=[i[0] for i in np.where(self.state==0)] # obtains the zero index

    if zero_index[0] == 0 or zero_index[1] == 0: # if space is bottom row, last column
        return False

    else:
        down_right_val = self.state[zero_index[0]-1, zero_index[1]-1] # perform swap between zero and down right
        new_state = self.state.copy()
        new_state[zero_index[0],zero_index[1]] = down_right_val
        new_state[zero_index[0]-1, zero_index[1]-1] = 0
        return new_state, down_right_val

def try_move_down_left(self):
    zero_index=[i[0] for i in np.where(self.state==0)] # obtains the zero index

    if zero_index[0] == 0 or zero_index[1] == 3: # if space is top row or last column, return false
        return False

    else:
        down_left_val = self.state[zero_index[0]-1, zero_index[1]+1] # perform swap between zero and down left
        new_state = self.state.copy()
        new_state[zero_index[0],zero_index[1]] = down_left_val
        new_state[zero_index[0]-1, zero_index[1]+1] = 0
        return new_state, down_left_val
```

```
def get_h_cost(self, new_state, goal_pos_dic): # gets the heuristic function cost
    curr = new_state # current state

    sum_chessboard = 0 # initialize herusitic value

    for i in range(4):
        for j in range(4):
            if curr[i, j] != 0:
                x_diff = abs(i - goal_pos_dic[curr[i, j]][0]) # get the aboslute difference between x coord
                y_diff = abs(j - goal_pos_dic[curr[i, j]][1]) # get the aboslute difference between y coord
                maximum = max(x_diff, y_diff) # obtain the maximun distance of x or y
                sum_chessboard += maximum # add that valeu to the sum

    return sum_chessboard


def a_star_search(self, initial_state, goal_state, goal_state_dic): # inital state used for output file
    start = time.time()

    queue = [(self,0)] # queue of (found but unvisited nodes, path cost+heuristic cost), ordered by the second element
    queue_num_nodes_popped = 0 # # of nodes popped off the queue, measures time performance
    queue_max_length = 1 # max number of nodes in queue, measures space performance

    depth_queue = [(0,0)] # queue of node depth, (depth, path_cost+heuristic cost)
    path_cost_queue = [(0,0)] # queue for path cost, (path_cost, path_cost+heuristic cost)
    visited = set([]) # use set to record the visited nodes

    while queue:
        # sort queue based on path_cost+heuristic cost, in ascending order, performed on 3 lists
        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[1])

        print(path_cost_queue)
        # each time update maximum length of the queue
        if len(queue) > queue_max_length:
            queue_max_length = len(queue)

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

        queue_num_nodes_popped += 1
        current_depth = depth_queue.pop(0)[0]  # choose and remove the depth for current node
        current_path_cost = path_cost_queue.pop(0)[
            0]  # select and remove the path cost for reaching current node
        visited.add(
            tuple(current_node.state.reshape(1, 16)[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(initial_state, goal_state, queue_max_length)

            print('Time performance: ',str(queue_num_nodes_popped),'nodes popped off the queue.')
            print('Space performance: ', str(queue_max_length),'nodes in the queue at its max.')
            print('Time spent: %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()

                if tuple(new_state.reshape(1,16)[0]) not in visited: # if node has not be visited already
                    path_cost=current_path_cost+1 # update current path cost
                    depth = current_depth+1 # update the current depth
                    # get heuristic cost
                    h_cost = self.get_h_cost(new_state,goal_state_dic)  # obtain the heuristic cost of new node
                    # create a new child node

                    total_cost = path_cost+h_cost  # updates the total cost

                    # create new node, and add the node to the queue

                    current_node.move_down = Node(state=new_state,parent=current_node,action='down 3',depth=depth,
                                    step_cost=up_value,path_cost=path_cost,heuristic_cost=h_cost)
                    queue.append((current_node.move_down, total_cost))
                    depth_queue.append((depth, total_cost))
                    path_cost_queue.append((path_cost, total_cost))

            # 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(1,16)[0]) not in visited: # if node has not be visited already
                    path_cost=current_path_cost+1 # update current path cost
                    depth = current_depth+1 # update the current depth
                    # get heuristic cost
                    h_cost = self.get_h_cost(new_state,goal_state_dic)  # obtain the heuristic cost of new node
                    # create a new child node
                    total_cost = path_cost+h_cost  # updates the total cost

                    # create new node, and add the node to the queue

                    current_node.move_up = Node(state=new_state,parent=current_node,action='right 1',depth=depth,
                                    step_cost=left_value,path_cost=path_cost,heuristic_cost=h_cost)
                    queue.append((current_node.move_up, total_cost))
                    depth_queue.append((depth, total_cost))
                    path_cost_queue.append((path_cost, total_cost))
```

```
            # 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(1,16)[0]) not in visited: # if node has not be visited already
                    path_cost=current_path_cost+1  # update current path cost
                    depth = current_depth+1  # update the current depth
                    # get heuristic cost
                    h_cost = self.get_h_cost(new_state,goal_state_dic)  # obtain the heuristic cost of new node
                    # create a new child node
                    total_cost = path_cost+h_cost  # updates the total cost

                    # create new node, and add the node to the queue

                    current_node.move_up = Node(state=new_state,parent=current_node,action='up 7',depth=depth,
                                    step_cost=lower_value,path_cost=path_cost,heuristic_cost=h_cost)
                    queue.append((current_node.move_up, total_cost))
                    depth_queue.append((depth, total_cost))
                    path_cost_queue.append((path_cost, total_cost))


            # 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(1,16)[0]) not in visited: # if node has not be visited already
                    path_cost=current_path_cost+1  # update current path cost
                    depth = current_depth+1  # update the current depth
                    # get heuristic cost
                    h_cost = self.get_h_cost(new_state,goal_state_dic)  # obtain the heuristic cost of new node
                    # create a new child node
                    total_cost = path_cost+h_cost  # updates the total cost

                    # create new node, and add the node to the queue

                    current_node.move_left = Node(state=new_state,parent=current_node,action='left 5',depth=depth,
                                    step_cost=right_value,path_cost=path_cost,heuristic_cost=h_cost)
                    queue.append((current_node.move_left, total_cost))
                    depth_queue.append((depth, total_cost))
                    path_cost_queue.append((path_cost, total_cost))
```

```
        if current_node.try_move_up_left():
            new_state,up_left_value = current_node.try_move_up_left()
            # check if the resulting node is already visited
            if tuple(new_state.reshape(1,16)[0]) not in visited: # if node has not be visited already
                path_cost=current_path_cost+1  # update current path cost
                depth = current_depth+1  # update the current depth
                # get heuristic cost
                h_cost = self.get_h_cost(new_state,goal_state_dic)  # obtain the heuristic cost of new node
                # create a new child node
                total_cost = path_cost+h_cost  # updates the total cost

                # create new node, and add the node to the queue

                current_node.move_up_left = Node(state=new_state,parent=current_node,action='up left 6',depth=depth,
                                step_cost=up_left_value,path_cost=path_cost,heuristic_cost=h_cost)
                queue.append((current_node.move_up_left, total_cost))
                depth_queue.append((depth, total_cost))
                path_cost_queue.append((path_cost, total_cost))

        if current_node.try_move_up_right():
            new_state,up_right_value = current_node.try_move_up_right()

            # check if the resulting node is already visited
            if tuple(new_state.reshape(1,16)[0]) not in visited: # if node has not be visited already
                path_cost=current_path_cost+1  # update current path cost
                depth = current_depth+1   # update the current depth
                # get heuristic cost
                h_cost = self.get_h_cost(new_state,goal_state_dic)  # obtain the heuristic cost of new node
                # create a new child node
                total_cost = path_cost+h_cost  # updates the total cost

                # create new node, and add the node to the queue

                current_node.move_up_right = Node(state=new_state,parent=current_node,action='up right 8',depth=depth,
                                step_cost=up_right_value,path_cost=path_cost,heuristic_cost=h_cost)
                queue.append((current_node.move_up_right, total_cost))
                depth_queue.append((depth, total_cost))
                path_cost_queue.append((path_cost, total_cost))
```

```
        if current_node.try_move_down_left():
            new_state,down_left_value = current_node.try_move_down_left()
            # check if the resulting node is already visited

            if tuple(new_state.reshape(1,16)[0]) not in visited: # if node has not be visited already
                path_cost=current_path_cost+1  # update current path cost
                depth = current_depth+1  # update the current depth
                # get heuristic cost
                h_cost = self.get_h_cost(new_state,goal_state_dic)  # obtain the heuristic cost of new node
                # create a new child node
                total_cost = path_cost+h_cost  # updates the total cost

                # create new node, and add the node to the queue

                current_node.move_down_left = Node(state=new_state,parent=current_node,action='down left 4',depth=depth,
                                step_cost=down_left_value,path_cost=path_cost,heuristic_cost=h_cost)
                queue.append((current_node.move_down_left, total_cost))
                depth_queue.append((depth, total_cost))
                path_cost_queue.append((path_cost, total_cost))

        if current_node.try_move_down_right():
            new_state,down_right_value = current_node.try_move_down_right()
            # check if the resulting node is already visited

            if tuple(new_state.reshape(1,16)[0]) not in visited: # if node has not be visited already
                path_cost=current_path_cost+1  # update current path cost
                depth = current_depth+1  # update the current depth
                # get heuristic cost
                h_cost = self.get_h_cost(new_state,goal_state_dic) # obtain the heuristic cost of new node
                # create a new child node
                total_cost = path_cost+h_cost # updates the total cost

                #create new node, and add the node to the queue

                current_node.move_down_right = Node(state=new_state,parent=current_node,action='down right 2',depth=depth,
                                step_cost=down_right_value,path_cost=path_cost,heuristic_cost=h_cost)

                queue.append((current_node.move_down_right, total_cost))
                depth_queue.append((depth, total_cost))
                path_cost_queue.append((path_cost, total_cost))
```

```
    def path_print(self):
        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]

        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)

        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(), 'Wn'))))

            step_counter += 1
```

```
#test = np.array([1, 2, 3, 4, 5, 6, 0, 7, 8, 9, 10, 11, 12, 13, 14, 15]).reshape(4, 4)
#initial_state = test
#goal_state = np.array([1, 2, 0, 4, 5, 6, 9, 3, 8, 13, 10, 7, 12, 14, 11, 15])

f = open("input3.txt", "r")

def main():
    list_of_nums = [] # stores the numbers of the input fileW
    for line in f:
        line_strip = line.strip()
        line_list = line_strip.split()
        for char_num in line_list:
            list_of_nums.append(int(char_num))

    input = list_of_nums[0:16] # split the list of nubmers into 2, representing input state and goal state
    output = list_of_nums[16:32]

    initial_state = np.array(input).reshape(4, 4) # reshape the arrays into matrices
    goal_state = np.array(output).reshape(4, 4)

    goal_state_dic = {} # stores a dictionary of the indices of the goal states

    for i in range(16): # addo the dictionary of the goal state indices
        result = np.where(goal_state==i)
        goal_state_dic[i] = (result[0][0], result[1][0])

    f.close() # close the file
    root_node = Node(state=initial_state, parent=None, action=None, depth=0, step_cost=0, path_cost=0, heuristic_cost=0) # create initial node
    root_node.a_star_search(initial_state, goal_state, goal_state_dic) # perform a star search


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

Output text files:

Input1.txt:

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

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

6
27
6 5 8 1 2 3
0 6 6 6 6 6 6
```

Screen shot (output of input1) :

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

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

6
27
6 5 8 1 2 3
0 6 6 6 6 6 6
```

Input2.txt:

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

9 3 7 1
13 5 4 6
12 15 0 10
```

```
14 2 11 8

11
100
1 6 8 2 3 4 6 4 7 7 1
0 11 11 11 11 11 11 11 11 11 11 11
```

Screen shot (output of input2) :

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

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

11
100
1 6 8 2 3 4 6 4 7 7 1
0 11 11 11 11 11 11 11 11 11 11 11
```

Input3.txt:

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

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

16
167
7 5 4 5 3 2 1 8 6 7 4 6 3 2 1 2
0 15 15 16 16 16 16 16 16 16 16 16 16 16 16 16 16
```

Screen shot (output of input3) :

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

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

16
167
7 5 4 5 3 2 1 8 6 7 4 6 3 2 1 2
0 15 15 16 16 16 16 16 16 16 16 16 16 16 16 16 16