Lingfeng Fan(lf2606)
Daoyu Li(dl5321)

Project 1: 26-Puzzle Problem

Program Usage

- Run in command-line interface/terminal:
- python3 <main.py path> <input file path> <desired output file path>
- Example: python3 main.py /Users/Desktop/Input3.txt /Users/Desktop/Ouput3.txt

—-----------------------

Output1.txt:

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

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

1 2 3
4 13 5
6 7 8

9 10 11
15 12 14
24 16 17

18 19 20
21 0 23
25 22 26
```

```
6
23
D W S D E N
6 6 6 6 6 6 6
```

—----------------------------

Output2.txt:

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

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

1 10 2
4 5 3
6 7 8

9 13 11
21 12 14
15 16 17

18 0 20
24 19 22
25 26 23


13
44
E N W D S W D S E E N W N
13 13 13 13 13 13 13 13 13 13 13 13 13 13
```

—---------------------------

Output3.txt:

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

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

0 2 3
1 7 14
6 8 5

12 9 10
4 13 11
21 16 17

18 19 20
22 25 23
15 24 26

16
59
S E N D N W W S D E S W U N U N
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
```

—------------------------------

Source code:

```
import heapq
import sys
```

```
class PuzzleState:
   instance_count = 0

   def __init__(self, state, parent=None, action=None, path_cost=0,
goal=None):
       PuzzleState.instance_count += 1
       self.state = state
       self.parent = parent
       self.action = action
       self.path_cost = path_cost

       if goal is not None:
           # find heuristic h(n)
           self.heuristic_cost = self.manhattan_distance(goal)
       Else:
           # Goal state has h(n) = 0
           self.heuristic_cost = 0

       self.total_cost = self.path_cost + self.heuristic_cost  #
evaluation f(n) = g(n) + h(n)

       # print("total_cost: ", self.total_cost)
       # print("heuristic cost: ", self.heuristic_cost)
       # print("path_cost: ", self.path_cost)
       # print("\n")


   def manhattan_distance(self, goal):
       distance = 0

       # h(n) = Sum of Manhattan distances of the tiles from their
goal positions -> heuristic function
       for i in range(27):
           if self.state[i] != '0':    # '0' represents the blank
position
```

```
               # layer 1: 0-8; layer 2: 9-17; layer 3: 18-26
               # coordinate x = i % 3
               # coordinate y = i // 3 % 3
               # coordinate z = i // 9
               current_pos = (i % 3, (i // 3) % 3, i // 9)     # [0,
1, 2]

               goal_index = goal.state.index(self.state[i])    # find
the index of current tile in goal state
               goal_pos = (goal_index % 3, (goal_index // 3) % 3,
goal_index // 9)

               # Manhattan Dist = |delta(x)| + |delta(y)| +
|delta(z)|
            distance += abs(current_pos[0] - goal_pos[0]) +
abs(current_pos[1] - goal_pos[1]) + abs(
                   current_pos[2] - goal_pos[2])

       return distance

   # required function: override the less-than (<) operator for
comparing PuzzleState objects
   def __lt__(self, other):
       # determine the criteria used to compare nodes in the priority
queue
       return self.total_cost < other.total_cost

   # Goal nodes always hold h(n) = 0
   def is_goal(self):
       return self.heuristic_cost == 0

   @classmethod
   def get_instance_count(cls):
       return cls.instance_count
```

```
def get_children(parent_state, goal_state):
    children = []
    zero_index = parent_state.state.index('0') # the position of blank
position

    # Define move directions based on current blank position
    directions = {'E': 1, 'W': -1, 'N': -3, 'S': 3, 'U': -9, 'D': 9}

    # remove impossible actions
    if zero_index % 3 == 2:     # If the empty tile is at the right
edge, remove 'E'
        del directions['E']

    if zero_index % 3 == 0:     # If the empty tile is at the left
edge, remove 'W'
        del directions['W']

    if zero_index % 9 < 3:      # If the empty tile is at the top
edge, remove 'N'
        del directions['N']

    if zero_index % 9 > 5:      # If the empty tile is at the bottom
edge, remove 'S'
        del directions['S']

    for action, move in directions.items():
        # calculate new index for blank position
        new_zero_index = zero_index + move
        # determine if the empty tile can perform "Up" or "Down"
movement
        if 0 <= new_zero_index < len(parent_state.state):
            # Swap tiles when new blank index is valid
            new_state = parent_state.state[:] # shallow copy
            new_state[zero_index], new_state[new_zero_index] =
new_state[new_zero_index], new_state[zero_index]

            # path cost + 1 for child node
            child_state = PuzzleState(new_state, parent_state, action,
parent_state.path_cost + 1, goal = goal_state)
            children.append(child_state)

    return children


def a_star_search(start_state, goal_state):
    nodes_number = 1
    open_set = []       # nodes that have been discovered but not yet
evaluated
    closed_set = set()  # nodes that have already been evaluated

    # add root node to the open set/heap
    heapq.heappush(open_set, (start_state.total_cost, start_state))


    while open_set:
        # check lowest f(n) in the priority queue
        _, current_state = heapq.heappop(open_set)

        # return when a goal node is found, call reconstruct_path to
build up actions and f(n) lists
        if current_state.is_goal():
            return reconstruct_path(current_state, nodes_number)

        # evaluated nodes are added into closed set
        closed_set.add(tuple(current_state.state))

        for child in get_children(current_state, goal_state):
            # add non-evaluated nodes into open set
```

```
           if tuple(child.state) not in closed_set:
               heapq.heappush(open_set, (child.total_cost, child))
               nodes_number += 1


# only called when a goal node is found, traceback actions and f(n)
def reconstruct_path(state, nodes_number):
   actions = []
   fn = []
   depth = 0

   while state.action is not None:  # root exclusive
       actions.append(state.action)    # d
       fn.append(state.total_cost)      # d
       state = state.parent
       depth += 1

   fn.append(state.total_cost)      # d+1, root node counted

   # reverse the actions & f(n) lists since we'are appending them
from child to parent
   return actions[::-1], fn[::-1], depth, nodes_number


# get initial puzzle state and goal puzzle state from input
def read_puzzle_input(file_path):
   start = []
   goal = []

   with open(file_path, 'r') as file:
       lines = file.read().splitlines()

   # three layers of the initial state: line 0-10
   for i in range(0, 11):
       if i == 3 or i == 7:    # skip empty lines in input file
```

```
            continue
        for int in lines[i].split(" "):
            start.append(int)

    # three layers of the goal state: line 12-22
    for i in range(12, 23):
        if i == 15 or i == 19:  # skip empty lines in input file
            continue
        for int in lines[i].split(" "):
            goal.append(int)

    return start, goal


if __name__ == '__main__':

    # start = [1, 2, 3, 4, '0', 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
    # goal = [1, 2, 3, 4, 5, 8, 6, 7, 17, 9, 10, 11, 12, 13, 14, 15,
25, 16, 18, 19, 20, '0', 21, 23, 24, 22, 26]

    # example input_file_path = '/Users/Desktop/Input3.txt'  ->
replace with own input file path
    # example output_file_path = '/Users/Desktop/Output3.txt'  ->
replace with own desired output file path

    # USAGE: python3 main.py <input txt file path> <desired output txt
file path>

    input_file_path = sys.argv[1]
    output_file_path = sys.argv[2]

    start, goal = read_puzzle_input(input_file_path)
    # print(start, goal)
    # Initialize the start and goal states
    goal_state = PuzzleState(goal)
    start_state = PuzzleState(start, goal=goal_state)

    # Perform the A* search
    actions, fn, depth, nodes_number= a_star_search(start_state,
goal_state)

    print("\n")

    print("actions: ", actions)
    print("f(n) values of the nodes along the solution path: ", fn)
    print("total number of nodes generated: ", nodes_number)
    print("depth level - shallowest goal node found: ", depth)

    # print(actions, fn, depth, nodes_number)

    # Step 1: Read the input file
    with open(input_file_path, 'r') as file:
        input_content = file.readlines()

    # Step 2 and 3: Write the input file content to the output file
    with open(output_file_path, 'w') as file:
        file.writelines(input_content)

        file.writelines("\n")
        file.writelines("\n")
        file.write(str(depth)) # line 25

        file.writelines("\n")
        file.write(str(nodes_number)) # line 26

        file.writelines("\n")
        file.writelines(" ".join(action for action in actions)) # line
27
```

```
        file.writelines("\n")
        file.write(" ".join(str(f) for f in fn)) # line 28


# print(f"Total PuzzleState instances created:
{PuzzleState.get_instance_count()}")
```