Michelle Lin (netid: ml7188) and Tianzuo Liu (netid: tl3119)

To run the program, in terminal (at the same path with source code file):

$python Puzzles.py (filename.txt)

Output for Input1.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
22
D W S D E N
6 6 6 6 6 6 6

Output for Input2.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
43
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

Output for Input3.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
58
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:

```
# Author: Michelle Lin (netid: ml7188) and Tianzuo Liu (netid:
tl3119)
import argparse
from typing import List
import copy


# Define the heuristic function (Sum of Manhattan distance for
each node)
def manhattan_distance(state, goal):
    sum_distance = 0
    for x in range(3):
        for y in range(3):
            for z in range(3):
                value = state[x][y][z]
                if value != 0:
                    goal_x, goal_y, goal_z =
find_tile_position(goal, value)
                    sum_distance += abs(x - goal_x) + abs(y -
goal_y) + abs(z - goal_z)
    return sum_distance

# Helper function to find the position of a tile in the grid
def find_tile_position(state, tile):
    for x in range(3):
        for y in range(3):
            for z in range(3):
```

```
                if state[x][y][z] == tile:
                    return x, y, z

def find_reachable_states(state):
    # Possible actions: East (E), West (W), North (N), South
(S), Up (U) and Down (D)
    reachable_states = []
    reachable_states_actions = []
    # East:
    if find_tile_position(state, 0)[2] != 2: # not in the third
column of any level of tiles
        reachable_states += [get_state(state, "E")]
        reachable_states_actions += ["E"]
    # West:
    if find_tile_position(state, 0)[2] != 0: # not in the first
column of any level of tiles
        reachable_states += [get_state(state, "W")]
        reachable_states_actions += ["W"]
    # North:
    if find_tile_position(state, 0)[1] != 0: # not in the first
row of any level of tiles
        reachable_states += [get_state(state, "N")]
        reachable_states_actions += ["N"]
    # South:
    if find_tile_position(state, 0)[1] != 2: # not in the third
row of any level of tiles
        reachable_states += [get_state(state, "S")]
        reachable_states_actions += ["S"]
    # Up:
    if find_tile_position(state, 0)[0] != 0: # not in the first
level of tiles
        reachable_states += [get_state(state, "U")]
        reachable_states_actions += ["U"]
    # Down:
    if find_tile_position(state, 0)[0] != 2: # not in the third
level of tiles
        reachable_states += [get_state(state, "D")]
        reachable_states_actions += ["D"]

    return reachable_states, reachable_states_actions

def get_state(state, action):
    tile_position = find_tile_position(state, 0)
    new_state = copy.deepcopy(state)
    if action == "E": # go to the right, possible sample input:
([0, 1, 2] or [1, 0, 2])
```

```
        if tile_position[2] == 0: # [0, 1, 2]
            # new_state = state
            new_state[tile_position[0]][tile_position[1]] =
[state[tile_position[0]][tile_position[1]][1], 0,
state[tile_position[0]][tile_position[1]][2]]
        elif tile_position[2] == 1: # [1, 0, 2]
            # new_state = state
            new_state[tile_position[0]][tile_position[1]] =
[state[tile_position[0]][tile_position[1]][0],
state[tile_position[0]][tile_position[1]][2], 0]
    elif action == "W": # go to the left, possible sample input:
([1, 0, 2] or [1, 2, 0])
        if tile_position[2] == 1: # [1, 0, 2]
            # new_state = state
            new_state[tile_position[0]][tile_position[1]] = [0,
state[tile_position[0]][tile_position[1]][0],
state[tile_position[0]][tile_position[1]][2]]
        elif tile_position[2] == 2: # [1, 2, 0]
            # new_state = state
            new_state[tile_position[0]][tile_position[1]] =
[state[tile_position[0]][tile_position[1]][0], 0,
state[tile_position[0]][tile_position[1]][1]]
    elif action == "S": # slide up on the same level
        # new_state = state
        new_state[tile_position[0]][tile_position[1]]
[tile_position[2]] = new_state[tile_position[0]]
[tile_position[1]+1][tile_position[2]]
        new_state[tile_position[0]][tile_position[1]+1]
[tile_position[2]] = 0
    elif action == "N": # slide down on the same level
        # new_state = state
        new_state[tile_position[0]][tile_position[1]]
[tile_position[2]] = new_state[tile_position[0]]
[tile_position[1]-1][tile_position[2]]
        new_state[tile_position[0]][tile_position[1]-1]
[tile_position[2]] = 0
    elif action == "U": # go up a level, possible input: on
level 1 or 2
        # new_state = state
        new_state[tile_position[0]][tile_position[1]]
[tile_position[2]] = state[tile_position[0]-1][tile_position[1]]
[tile_position[2]]
        new_state[tile_position[0]-1][tile_position[1]]
[tile_position[2]] = 0
    elif action == "D": # go down a level, possible input: on
level 0 or 1
```

```
        # new_state = state
        new_state[tile_position[0]][tile_position[1]]
[tile_position[2]] = state[tile_position[0]+1][tile_position[1]]
[tile_position[2]]
        new_state[tile_position[0]+1][tile_position[1]]
[tile_position[2]] = 0
    else:
        raise Exception("Invalid action")
    return new_state

# Define the A* search node class
class Node:
    def __init__(self, state: List[List[List[int]]], depth: int,
cost: int, parent, action):
        self.state = state
        self.depth = depth
        self.cost = cost
        self.parent = parent
        self.action = action

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

# Define the A* search algorithm
def astar_search(initial_state, goal_state):
    #DECLARE A NODE: Node(state, depth, cost, parent)
    initial_node = Node(initial_state, 0,
manhattan_distance(initial_state, goal_state), None, None)

    frontier = [initial_node] # current frontier
    reached = [initial_node.state] # reached states

    nodes_generated_count = 0

    while len(frontier) > 0:
        current_cost = 10000000000000

        for i in range(len(frontier)): # loop through the nodes
in the frontier
            if frontier[i].cost + frontier[i].depth <
current_cost:
                current_node = frontier[i]
                current_node_index = i
                current_cost = frontier[i].cost +
frontier[i].depth
```

```
        frontier.pop(current_node_index) # pop the current_node
from the frontier

        if current_node.state == goal_state: # check goal state
before expansion
            print("solution found!!!")
            generate_output_file(current_node,
nodes_generated_count)
            return None

        reachable_states, reachable_states_actions =
find_reachable_states(current_node.state) # expand

        for i in range(len(reachable_states)):
            if reachable_states[i] not in reached:
                new_node = Node(reachable_states[i],
current_node.depth+1, manhattan_distance(reachable_states[i],
goal_state), parent = current_node, action =
reachable_states_actions[i])
                nodes_generated_count += 1
                frontier.append(new_node)
                reached.append(new_node.state)
    print("solution not found!!! ")
    return None

def is_goal(state, goal_state):
    if state == goal_state:
        return True
    else:
        return False

def generate_output_file(node, nodes_generated_count):
    # copy paste the initial state and goal state to output file
    parser = argparse.ArgumentParser()
    parser.add_argument('filename')
    cmdline = parser.parse_args()

    # Read input file and create initial and goal states
    with open(cmdline.filename, 'r') as file:
        lines = file.read().splitlines()

    new_file_name = cmdline.filename[:-4] + "solution.txt"
    f = open(new_file_name, "w")
    for i in lines:
        f.write(i)
        f.write("\n")
```

```
    # Line 24 is a blank line
    f.write ("\n")

    # Line 25 is the depth level d of the shallowest goal node
as found by the A* algorithm
    # (assume the root node is at level 0.)
    f.write(str(node.depth))
    f.write ("\n")

    # Line 26 is the total number of nodes N generated in your
tree (including the root node.)
    f.write(str(nodes_generated_count))
    f.write ("\n")

    # Line 27 contains the solution (a sequence of actions from
root node to goal node) represented by A's.
    solution = []
    solution_path_cost = []
    while node.parent != None: # trace back the solution from
leaf node
        solution.append(node.action)
        solution_path_cost.append(node.cost + node.depth) # f(n)
        node = node.parent
    solution_path_cost.append(node.cost + node.depth) # d+1,
parent node f(n)
    solution.reverse() # make the list solution in reverse (so
that it comes from root node)
    for i in range(len(solution)):
        f.write(solution[i])
        f.write(" ")
    f.write ("\n")

    # Line 28 contains the f(n) values of the nodes along the
solution path,
    # from the root node to the goal node, separated by blank
spaces.
    solution_path_cost.reverse()
    for i in range(len(solution_path_cost)):
        f.write(str(solution_path_cost[i]))
        f.write(" ")

# Main function
def main() -> None:
    # Parse command-line arguments
    parser = argparse.ArgumentParser()
```

```
    parser.add_argument('filename')
    cmdline = parser.parse_args()

    # Read input file and create initial and goal states
    with open(cmdline.filename, 'r') as file:
        lines = file.read().splitlines()

    initial_state = [[list(map(int, line.split())) for line in
lines[:3]]]  # Initial state
    initial_state += [[list(map(int, line.split())) for line in
lines[4:7]]]
    initial_state += [[list(map(int, line.split())) for line in
lines[8:11]]]

    goal_state = [[list(map(int, line.split())) for line in
lines[12:15]]]  # Goal state
    goal_state += [[list(map(int, line.split())) for line in
lines[16:19]]]
    goal_state += [[list(map(int, line.split())) for line in
lines[20:23]]]

    # Find the solution using A* search
    astar_search(initial_state, goal_state)

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