CS GY 6613 - Artificial Intelligence 1
Project: 26 - Puzzle

Group Members:
- Swaran Iyappan - si2320
- Saumya Parag Phadkar- sp7778

Instructions on how to run the program:
- In the same folder as the source code, create and populate a file “input.txt”.
- Run the file “python 26-puzzle.py”
- The output file “output.txt” will get created or, if it already exists, overwritten in the same folder.

Source Code:
```python
'''

AI Assignment - 26 Puzzle Problem

sp7778 - Saumya Parag Phadkar
si2320 - Swaran Iyappan

'''
import numpy as np
import heapq


'''

manhattanDistance
Used to calculate our heuristic
I/P:
    currentState - State from which we are trying to reach our goal state
    goalState    - State which we are trying to reach
O/P:
    distance     - manhattan distance between current and goalState

'''
def manhattanDistance(currentState, goalState):
```

```
    distance = 0
    for i in range(3):
        for j in range(3):
            for k in range(3):
                if(currentState[i][j][k] != 0 and currentState[i][j][k] != 
goalState[i][j][k]):
                    x, y, z = np.where(goalState == currentState[i][j][k])
                    distance += abs(x[0] - i) + abs(y[0] - j) + abs(z[0] - 
k)
    return distance



'''
moves - A list of moves and their respective change of position
'''
moves = {
    'U': [-1, 0, 0],
    'D': [1, 0, 0],
    'N': [0, -1, 0],
    'S': [0, 1, 0],
    'E': [0, 0, 1],
    'W': [0, 0, -1]
}



'''
validMovesList
Used to calculate valid moves from the current state
I/P:
    state       - Current state for which we are trying to find valid 
moves
O/P:
    validMoves  - returns a list of valid moves, for eg: ['U', 'D', 'E', 
'S']

'''
def validMovesList(state):
    # x, y, z denote where we can find the blank in our board
```

```
    x, y, z = np.where(state == 0)
    x = x[0]
    y = y[0]
    z = z[0]
    validMoves = list()
    for move in moves:
        # Iterating all the moves and checking whether the move will is
valid or not by ensuring
        # that the blank does not leave the 3*3*3 board
        if(x + moves[move][0] >= 0 and x + moves[move][0] < 3 and
           y + moves[move][1] >= 0 and y + moves[move][1] < 3 and
           z + moves[move][2] >= 0 and z + moves[move][2] < 3):
            validMoves.append(move)
    return validMoves

'''
generateMoves
Used to generate the valid boards from the current state and list of valid
moves
I/P:
    state           - Current state for which we are trying to generate
the moves
    validMoves      - A list of valid moves, for eg: ['U', 'D', 'E', 'S']
O/P:
    generatedMoves  - A list of all valid boards after the moves are
completed
'''
def generateMoves(state, validMoves):
    generatedMoves = []
    for move in validMoves:
        currentState = np.copy(state)
        # blankX, blankY, blankZ are positions of the blank state
        blankX, blankY, blankZ = np.where(state == 0)
        blankX, blankY, blankZ = blankX[0], blankY[0], blankZ[0]
        moveAsCoodinates = moves[move]
        # numX, numY, numZ are the positions of the number which we are
swapping with blank state
        numX = blankX + moveAsCoodinates[0]
        numY = blankY + moveAsCoodinates[1]
        numZ = blankZ + moveAsCoodinates[2]
```

```
        currentState[numX][numY][numZ],
currentState[blankX][blankY][blankZ] =
currentState[blankX][blankY][blankZ], currentState[numX][numY][numZ]
        generatedMoves.append(currentState)
    return generatedMoves



'''
aStar
Used to generate the valid boards from the current state and list of valid
moves
I/P:
    startState      - State from which we are starting, i.e, our initial
state
    goalState       - State which we are trying to reach, i.e, our goal
state
O/P:
    currentPath     - Path of the goal state
    currentG        - Path cost of the goal state
    nodesGenerated  - Total number of nodes generated
    currentPathF    - f values of the nodes in the goal path
'''
def aStar(startState, goalState):

    # movesPriorityQueue is a priority queue with the following data:
    # 0 - f(node)
    # 1 - g(node)
    # 2 - Path
    # 3 - node path f
    # 4 - node represented as 3*3*3 matrix

    goalStateFlat = tuple(goalState.flatten())

    # Reached map
    reached = set()

    startingCost = 0
    startingPath = ""
```

```python
    nodesGenerated = 0

    f = manhattanDistance(startState, goalState)
    startingListF = [f]

    movesPriorityQueue = []

    heapq.heappush(movesPriorityQueue, (f, startingCost, startingPath,
startingListF, tuple(startState.flatten())))

    nodesGenerated += 1

    while(movesPriorityQueue):
        currentMove = heapq.heappop(movesPriorityQueue)

        currentStateFlat = currentMove[4]
        currentState = np.reshape(currentStateFlat, (3, 3, 3))
        currentF = currentMove[0]
        currentG = currentMove[1]
        currentPath = currentMove[2]
        currentPathF = currentMove[3]

        reached.add(currentStateFlat)

        if currentStateFlat == goalStateFlat:
            return currentPath, currentG, nodesGenerated, currentPathF

        validMoves = validMovesList(currentState)

        childrenMoves = generateMoves(currentState, validMoves)

        currentG = currentG + 1

        for i in range(len(childrenMoves)):
            child = childrenMoves[i]
            h = manhattanDistance(child, goalState)
            f = currentG + h
            child = tuple(child.flatten())
            # Computing the child path cost list
            childPathF = list(currentPathF)
```

```
            childPathF.append(f)
            # We add the child only if it is not already in reached
            if not child in reached:
                nodesGenerated += 1
                heapq.heappush(movesPriorityQueue, (f, currentG,
currentPath + validMoves[i], childPathF, child))

"""
    createArray -  Used to read input file and create the 3D arrays for
the initial and goal states
    O/P:
    startState  - State from which we starting the A* algorithm to reach
the goal state
    goalState   - State which we are trying to reach
"""

def createArray():
    temp = []
    startState = []
    goalState = []
    lines = input.readlines()
    for i in range(len(lines)):
        output.write(lines[i])
        if lines[i] != '\n':
            temp.append([int(x) for x in lines[i].strip().split(" ")])
            if i == len(lines)-1 or lines[i+1] == "\n":
                if i < 12:
                    startState.append(temp)
                else:
                    goalState.append(temp)
                temp = []
    return(np.array(startState), np.array(goalState))


# Reading and writing the output files
input = open("input.txt", "r+")
output = open("output.txt", "w+")

state, goal = createArray()
```

```
path, pathCost, totalNodesGenerated, pathF = aStar(state, goal)

output.write('\n\n')
output.write(str(pathCost)+'\n')
output.write(str(totalNodesGenerated)+'\n')
output.write(" ".join(path))
output.write("\n")
output.write(' '.join([str(v) for v in pathF]))
```

Outputs:

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
```