Nate Pierce
CIS479 - P1
A* Windy N Puzzle

Output:

```
In [52]: runfile('C:/Users/Nate/Desktop/Winter 2021/CIS479/P1/P1 Nate Pierce.py',
wdir='C:/Users/Nate/Desktop/Winter 2021/CIS479/P1')
The Starting State is:
2   8   3
6   7   4
1   5   0

The Goal State is:
1   2   3
8   0   4
7   6   5

Start Search Tree:
2   8   3
6   7   4
1   5   0

Path cost: g(n) =  2
Manhattan cost: h(n) =  19
Total cost for the node n is f(n) =  21
Node n =  1
2   8   3
6   7   4
1   0   5

Path cost: g(n) =  5
Manhattan cost: h(n) =  17
Total cost for the node n is f(n) =  22
Node n =  2
2   8   3
6   0   4
1   7   5

Path cost: g(n) =  7
Manhattan cost: h(n) =  15
Total cost for the node n is f(n) =  22
Node n =  3
2   8   3
0   6   4
1   7   5

Path cost: g(n) =  8
Manhattan cost: h(n) =  12
Total cost for the node n is f(n) =  20
Node n =  4
2   8   3
1   6   4
0   7   5
```

```
Path cost: g(n) =  10
Manhattan cost: h(n) =  10
Total cost for the node n is f(n) =  20
Node n =  5
2   8   3
1   6   4
7   0   5

Path cost: g(n) =  13
Manhattan cost: h(n) =  9
Total cost for the node n is f(n) =  22
Node n =  6
2   8   3
1   0   4
7   6   5

Path cost: g(n) =  16
Manhattan cost: h(n) =  7
Total cost for the node n is f(n) =  23
Node n =  7
2   0   3
1   8   4
7   6   5

Path cost: g(n) =  18
Manhattan cost: h(n) =  5
Total cost for the node n is f(n) =  23
Node n =  8
0   2   3
1   8   4
7   6   5

Path cost: g(n) =  19
Manhattan cost: h(n) =  2
Total cost for the node n is f(n) =  21
Node n =  9
1   2   3
0   8   4
7   6   5

Path cost: g(n) =  21
Manhattan cost: h(n) =  0
Total cost for the node n is f(n) =  21
Node n =  10
1   2   3
8   0   4
7   6   5

End search tree.
Total solution cost is  21
```

Screenshot:

[Click this link to view screenshot (Google Drive)]

Source Code:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 24 20:46:28 2021

CIS479 AI - P1
@Student Implementation: Nate Pierce
UMID 94712233
This code is adapted N-puzzle algorithm from
http://www.learntosolveit.com/python/algorithm_npuzzle.html to solve the
Windy N puzzle problem.
"""

from operator import add

class State:

    # Constructs the node/state class, given some dimension nsize.
    def __init__(self, nsize):
        self.nsize = nsize
        self.tsize = pow(self.nsize, 2) # Number of nodes is N x N or N^2
        self.goal = [1, 2, 3, 8, 0, 4, 7, 6, 5] # Goal configuration
explicitly defined
        self.start = [2, 8, 3, 6, 7, 4, 1, 5, 0] # Start configuration
        self.config = [] # holds the current state configuration
        self.g = 0 # path cost

    # Prints the state in matrix form.
    def printState(self, st):
        for (index, value) in enumerate(st):
            print(' %s ' % value, end=' ')
            if index in [x for x in range(self.nsize - 1, self.tsize,
self.nsize)]:
                print()
        print()
```

```
    # Helper function to gather movement choices at various positions in
the matrix. The key is just an index in the N x N matrix. Returns a list of
valid movement choices
    # for a single key value. A 1 corresponds to east, -1 to west, 3 to
south, -3 to north. This function only gets called by the expand()
function.
    def getValues(self, key):
        values = [1, -1, self.nsize, -self.nsize]
        valid = []
        for x in values: # x is the array element value - NOT the index
            if 0 <= key + x < self.tsize:
                if x == 1 and key in range(self.nsize - 1, self.tsize,
self.nsize): # conditional for matrix elements in 0th column
                    continue # continue statement returns control to the
beginning of the while loop, it does increment the counter.
                if x == -1 and key in range(0, self.tsize, self.nsize): #
conditional for matrix elements in N column - for this assignment, 2nd
column
                    continue # continue statement returns control to the
beginning of the while loop, it does increment the counter..
                valid.append(x)
        return valid

    # Returns an array of states, where each state in the array is achieved
by swapping the 0 tile with any of the four potential adjacent tiles as
determined
    # by the getValues function. This function gets called only in the
evalFunction function. expstates is the list of expansion nodes.
    def expand(self, st):

        pexpands = {} # dictionary of possible expansions

        # This for loop generates a dictionary of legal moves (hash value -
1 corresponds to east, -1 to west, 3 to south, -3 to north) and
        # key (element in the matrix)
        for key in range(self.tsize): # pexpands is a dictionary that holds
possible expansion moves for all elements in matrix
            pexpands[key] = self.getValues(key)

        pos = st.index(0) # index() returns the index location that
contains the value 0. For our beginning state, this is 8th position
(integer 8)
        moves = pexpands[pos] # moves is an array holding the possible
```

```
moves associated with number 0 on the N puzzle
        expstates = []

        for mv in moves:
            nstate = st[:]
            (nstate[pos + mv], nstate[pos]) = (nstate[pos], nstate[pos +
mv])
            expstates.append(nstate)
        return expstates

    # Checks if current state matches goal state. Return true if so, false
otherwise.
    def goal_reached(self, st):
        return st == self.goal

    # Takes as an argument the current state and determines the total windy
manhattan distance for it, which it returns as an integer mdist.
    def manhattanDistance(self, st):
        mdist = 0

        for elem in st: # elem is the element in the matrix, not the index.
The function index() returns the index for a given element.
            if elem != 0:
                dist = (self.goal.index(elem) - st.index(elem)) # the index
difference between the goal state and the current state
                vert = abs(dist) // self.nsize # the vertical distance
between the matrix elements of the goal and current states
                hor = abs(dist) % self.nsize # the horizontal distance
between the matrix elements of the goal and current states

                if dist > 0: # a positive difference indicates that the
state element needs to be moved up/right of its current state
                    mdist += 1*vert + 2*hor
                if dist < 0: # a negative difference indicates that the
state element needs to be moved down/left of its current state
                    mdist += 3*vert + 2*hor
                if dist == 0: # current state element matches corresponding
goal state element
                    mdist += dist

        return mdist

    # Path cost function G(n) - takes an expansion state node of some node
```

```
n and finds the path cost from the intial state to the
    # expansion state. Returns integer path cost.
    def pathCost(self, exp_st, cur_st):
        pcost = 0

        if exp_st == self.start:
            for elem in exp_st: # elem is the element in the matrix, not
the index. The function index() returns the index for a given element.
                if elem != 0:
                    dist = (cur_st.index(elem) - exp_st.index(elem)) # the
index difference between the goal state and the current state
                    vert = abs(dist) // self.nsize # the vertical distance
between the matrix elements of the goal and current states
                    hor = abs(dist) % self.nsize # the horizontal distance
between the matrix elements of the goal and current states

                    if dist > 0: # a positive difference indicates that the
state element needs to be moved up/right of its current state
                        pcost += 1*vert + 2*hor
                    if dist < 0: # a negative difference indicates that the
state element needs to be moved down/left of its current state
                        pcost += 3*vert + 2*hor
                    if dist == 0: # current state element matches
corresponding goal state element
                        pcost += dist

            return pcost

        else:
            for elem in exp_st: # elem is the element in the matrix, not
the index. The function index() returns the index for a given element.
                if elem != 0:
                    dist = (cur_st.index(elem) - exp_st.index(elem)) # the
index difference between the goal state and the current state
                    vert = abs(dist) // self.nsize # the vertical distance
between the matrix elements of the goal and current states
                    hor = abs(dist) % self.nsize # the horizontal distance
between the matrix elements of the goal and current states

                    if dist > 0: # a positive difference indicates that the
state element needs to be moved up/right of its current state
                        pcost += 1*vert + 2*hor
                    if dist < 0: # a negative difference indicates that the
```

```
state element needs to be moved down/left of its current state
                        pcost += 3*vert + 2*hor
                    if dist == 0: # current state element matches
corresponding goal state element
                        pcost += dist

            return pcost


    # This function gets called in the solve routine. Takes a current
state, expands it, calculates the windy manhattan distance for each of the
    # expanded states, sorts them by minimum distance into a list, and
returns the minimum distance state. If there is a tie, it is resolved by
    # first in first out.
    def evalFunction(self, state):
        exp_sts = self.expand(state.config) # expansion set (frontier) for
the current state
        cur_st = state.config # array with the current state configuration
        hdists = [] # manhattan distance cost array
        gdists = [] # path cost array
        evals = [] # initialize the evaluation array

        # For loop calculates manhattan distance for the set of expansion
states in exp_sts, i.e. NOT the current state passed
        # to this function.
        for st in exp_sts:
            hdists.append(self.manhattanDistance(st))
            gdists.append(self.pathCost(st, cur_st) + self.g)

        evals = list(map(add, hdists, gdists)) # element wise sum of g(n)
and h(n) i.e. the path cost and the manhattan cost for state n
        evals.sort() # sort from min to max
        min_path = evals[0] # select the minmum cost

        # This block selects the optimal expansion state from the list of
available expansion states. The first conditional
        # is the tie break condition. The optimal state is returned to
solve() here.
        if evals.count(min_path) > 1:
            least_paths = [st for st in exp_sts if
(self.manhattanDistance(st) + self.pathCost(st, cur_st) + self.g) ==
min_path]
            self.g = self.pathCost(st, cur_st)
```

```
            return least_paths[0] # FIFO tie breaker
        else:
            for st in exp_sts:
                if (self.manhattanDistance(st) + self.pathCost(st, cur_st)
+ self.g) == min_path:
                    print("Path cost: g(n) = ", (self.pathCost(st, cur_st)
+ self.g))
                    print("Manhattan cost: h(n) = ",
self.manhattanDistance(st) )
                    print("Total cost for the node n is f(n) = ", min_path)

                    self.g = self.pathCost(st, cur_st) + self.g
                    return st


    # Solve takes a pointer to a state object as an argument
    def solve(self, state):
        st = state.start
        state.config = st
        nodeCount = 1

        while not self.goal_reached(state.config):
            state.config = self.evalFunction(state) # st here is the node
in the expansion set calculated with minimum cost
            print("Node n = ", nodeCount)
            self.printState(state.config)
            nodeCount += 1


if __name__ == '__main__':
    state = State(3) # passes the dimension

    print('The Starting State is:')
    start = state.start # start is an array, NOT a state object
    state.printState(start)


    print('The Goal State is:')
    state.printState(state.goal)

    print("Start Search Tree:")
    state.printState(start)
    state.solve(state)
```

```
    print("End search tree.")
    print("Total solution cost is ", state.manhattanDistance(start))
```