```
import numpy as np
from queue import PriorityQueue
from random import shuffle
from time import time
```

LINK TO COLAB : https://colab.research.google.com/drive/1-lqY1pH6IJlpbET94Fjj9LjN3B1A_t-z

```
class PuzzleNode:

    '''
    The PuzzleNode class. Each object is a given state of an NxN grid.
    Each object has a state, and the ability to hold a parent.
    There are several methods as well, that can return the legal moves
    from that state and also a new state from a given move. Each component
    is explained in detail below.
    '''
```

```
    def __init__(self, edge, parent = None, state = None,
                 gval = None, fval = None, pruned = False):
        # Frequently used attributes of the shape of the state
        self.edge = edge
        self.size = edge**2 -1

        # Used for print out of the state
        self.max_str_size = len(str(self.size))

        # Used to trace back the solution
        self.parent = parent

        # Gval is the number of steps to reach the state
        self.gval = gval

        # Fval is the gval + the result of the heurisitc function (hval)
        self.fval = fval

        # Indicates if the node has been pruned for removal (this happens
```

```
        # when there is another node with the same state but lower gval.)
        self.pruned = pruned

        # Creates a random state if none is specified
        if type(state) == type(None) :
            lis = list(range(0,self.size+1))
            shuffle(lis)
            self.state = np.array(
                [[lis[self.edge*j+i] for i in range(self.edge)] for j in range

        # Intializes the state
        else:
            self.state = np.array(state)

    # Allows the objects to be compared on fval
    # for use in the frontier
    # This itegrates directly into the PriorityQueue class
    # which is quite helpful
    def __lt__(self,other):
        return self.fval < other.fval

    # This prints the state in an easily readable form
    def __str__(self):
        string = ''
        for i in range(self.edge):
            for j in range(self.edge):
                #If there are numbers with different number of digits
                #the blanks are filled in so the print out looks good
                string+=str(self.state[i][j]).ljust(self.max_str_size)
                string+=' '
            string += '\n'
        return string

    # Find the location of the zero (or blank)
    def find_zero(self):
        index = np.where(self.state==0)
        return (index[0][0],index[1][0])

    # Returns the state as a tuple of tuples
```

```
    # This is used so that the state can be inputed into a hash function
    # which we use to see if a state has been previosuly visited
    def tuple_state(self):
        return tuple(tuple(p) for p in self.state)

    # Determines if an individual move is not out of bounds
    # (off the edge) of the grid
    def valid(self,move):
        bad_moves = [-1,self.edge]
        if move[0] in bad_moves or move[1] in bad_moves:
            return False
        else:
            return True

    # Gets all the possible moves
    # First gets the moves in all 4 directions
    # Then checks with the valid function if they
    # don't move off the grid.
    # Returns a list of all legal moves.
    def get_moves(self):
        zero = self.find_zero()
        possible_moves = [(zero[0]+x,zero[1]+y) for x,y in [(0,1),(0,-1),(1,0)
        good_moves =[move if self.valid(move) else None for move in possible_m
        good_moves = list(filter(None, good_moves))
        return good_moves

    # Given a move, returns the new state after that move is made
    def move(self, move):
        if move in self.get_moves():
            new_state = np.array(self.state)
            zero = self.find_zero()
            new_state[zero[0]][zero[1]] = self.state[move[0]][move[1]]
            new_state[move[0]][move[1]] = 0
            return new_state
        else:
            print ("INVALID!")
```

```python
def errorTestOne(n,state):
    """
    Tests if a given input is of the correct shape
    and includes the correct numbers
    """
    # Checks that the number of rows and columns is correct
    if len(state) != n or len(state[0]) !=n :
        return True

    #Checks that all neccesary numbers appear
    if set(np.concatenate(state, axis=0)) != set(range(n**2)):
        return True
    return False

def inversions(state):
    """
    Counts the number of inversions. An inversion
    is when the state, expressed as one list,
    has a number a>b where a appears before b in the
    list.
    """
    # make the array one list and drop the 0
    lis =np.concatenate( state, axis=0)
    lis = list(filter(lambda a: a != 0, lis))

    # count the inversions
    inv_count = 0
    for i in range(len(lis)-1):
        for j in range(i+1,len(lis)):
            if lis[i]>lis[j]:
                inv_count+=1
    return inv_count

def isSolveable(state):
    """
    Determines if a given state is solveable.

    Rules adapted from
    https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/
```

```
and
http://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.ht

For a given N x N grid, the puzzle is solveable if:
    if N is odd, the inversions must be even
    if N is even, the inversions + row distance is even

We'll first start with an odd n. We'll use a 3 x 3 grid for
an example.

We first show that any legal move will leave the
same parity of inversions. We'll imgaine the state as one
list, of length n^2 (9 in our example). The blank can move +/-1
and +/-n, as those are moving the blank left, right, up or down
on the original square. Moving it +/- 1 doesn't change the number
of inversions, as the order of the numbers is not switching.
Moving it +/- n will not change the parity, as 1 number is jumping
ahead (or behind) over n-1 (2 in our case) numbers, thus the number
of inversions will change by +/- (n-1), and n is odd, leaving the
parity the same.

Next, we know that the goal state has an even (0) number of inversions.
Thus, any state which is reachable by legal moves from the goal state
(a solveable state) will have an even number of inversions

We'll next handle the even n case. We have already shown that moving
the blank +/- 1 will have no impact on the parity. The other legal moves
are moving +/- n, where n is even. Thus, a number is jumping over an odd
number of tiles, which will changes the parity of inversions by an odd num
To account for this, (as we want the parity to stay the same for all legal
we include the row distance, defined as the blank tile's distance from row
(equivalent to the row number). When making a +/- n move, the inversions w
change by an odd number and the row distance will change by an odd number,
so the inversions + row distance will change by an even number and maintai
the same parity.

The goal state has an even (inversions + row_distance), so all states whic
solveable will have an even (inversions + row_distance).
```

```
    NOTE: The logic and reasoning behind this system was described in the abov
    linked webpages. However, there appears to be a mistake, where they come t
    the wrong conclusions about an even n grid's sign. This can be shown by
    applying there rule to trivially solveable states, but they conclude that
    the state is unsolveable.

    """
    n = len(state)
    inv_count = inversions(state)
    # if n is odd
    if n % 2 != 0:
        if inv_count % 2 == 0:
            return True
        else:
            return False
    # if n is even
    else :
        zero_dist = np.where(np.array(state)==0)[0][0]
        if (zero_dist+inv_count) % 2 == 0:
            return True
        else:
            return False


def h0(state):
    """
    Heuristic 0: Misplaced Tiles

    This is used as a heuristic function for the A* search algorithm.
    The use of heuristic functions will be explained in more detail later.

    A valid heurisitic function must be admissible and consistent.
    An admissible heuristic fucntion is one that never overestimates the
    cost to reach the goal. A consistent heuristic is one where for every
    node n and every child node n' of n generated by a valid action, the
    heuristic cost at n must be no greater than the cost to
    get to n' + the heuristic cost at n'. This basically means no
    parent will have a cost greater than its child's cost + cost of actions
    to get to the child. The consistency of the Misplaced Tiles
```

```
    heuristic is proven in Russel and Norvig.


    This Heuristic function counts the number of misplaced tiles.
    It counts the tiles that are the same (not including the blank
    tile) and subsequently finds the number that are misplaced.

    """
    edge = len(state)

    # Makes np.arrays of the current state and goal state
    state_np = np.array(state)
    goal_np = np.array([[edge*j+i for i in range(edge)] for j in range(edge)])

    # checks the number of tiles that are the same
    same_tiles = len(np.where(state_np==goal_np)[0])

    #If all tiles are the same, return 0
    if same_tiles == edge**2:
        return 0
    else:
        # The next few lines are neccesary because
        # 0 is not a tile, and thus shouldn't be in the misplaced tile
        # count. However, numpy doesn't make this distinction.
        # To fix this, if 0 is in the correct position, it returns
        # the size of the puzzle + 1 - same_tiles, which is the correct
        # number of misplaced tiles in this case
        if state_np[0][0] == 0:
            return edge**2-same_tiles
        else:
            # and if 0 is not in the correct position, it returns the
            # size of the puzzle - same_tiles, which is the correct
            # number of misplaced tiles in this alternate case.
            return edge**2-1-same_tiles

def h1(state):
    '''
    Heuristic1 : Manhattan Distance
```

```
    The consistency of this heuristic was proven in Russel and Norvig.

    The Manhattan Distance of a tile is the number of moves (horizontal
    or vertical) that it takes to get to its goal position. This function
    finds adds the Manhattan Distance of all tiles (except the blank) and
    returns it. It's clear that this is admissable as each tile must be
    moved at least that many movements to get to its goal state.
    '''
    edge = len(state)
    state_np = np.array(state)
    goal_np = np.array([[edge*j+i for i in range(edge)] for j in range(edge)])
    dist =0
    for num in range(1,len(state)**2):
        a =(np.where(state_np == num)[0][0],np.where(state_np == num)[1][0])
        b =(np.where(goal_np == num)[0][0],np.where(goal_np == num)[1][0])
        dist += abs(a[0]-b[0])+abs(a[1]-b[1])
    return dist

def linearConflict(state):
    """
    Returns the number of Linear Conflicts

    Inspiration for this implemenation came from
    https://github.com/Jason-Yuan/8PuzzleGameSovler/blob/master/heuristic_esti

    Two tiles a and b are in linear conflict if they are in the
    same row or column, their goal positions are in the same row
    or column, and the goal position of one is blocked by the other.

    This means that one of them tiles must move out of the way for
    the other to get through and the orignal tile must come back
    (at a minimum ).

    I implemented this by only doing it for ros first, then
    transposing the matrixes and doing it for rows again
    (which tests for columns)
    """

    LinearConflict = 0
```

```
    edge = len(state)
    state_np = np.array(state)
    goal_np = np.array([[edge*j+i for i in range(edge)] for j in range(edge)])
    state_np_t = state_np.transpose()
    goal_np_t = goal_np.transpose()

    # do it for both normal and transpose
    for state, goal in [[state_np,goal_np],[state_np_t,goal_np_t]]:

        for x in range(0,edge):
            counter = 0
            temp = []
            #find the potential conflicts
            for y in range(0,edge):
                if state[x][y] in goal[x]:
                    temp.append((x,y))
                    counter += 1

            #create an easy list to compare conflicts
            goals = [goal[temp[i][0]][temp[i][1]] for i in range(counter)]
            curs = [state[temp[i][0]][temp[i][1]] for i in range(counter)]
            for i in range(len(goals)):
                for j in range(len(curs)):
                    # make sure they are not 0 and not the same
                    if goals[i]!=0and goals[j]!=0and curs[i]!=0and curs[j]!=0:
                        if i != j:
                            #Tests for the conflict. The other order
                            #is accounted for in the loop

                            # The test if if the goal of one is greater than
                            # the other but the position is opposite
                            if goals[i]-goals[j]>0 and curs[i]-curs[j]<0 :
                                LinearConflict +=1
    return LinearConflict


def h2(state):
    """
```

```
    NEW HEURISTIC: Manhattan Distance + Linear Conflict

    This heuristic function combines the manhattan distance with
    the linear conflicts*2. This works because for every linear
    conflict, a tile will have to move out of the way and back again,
    hence the 2 multipler.

    """
    manhattan = h1(state)
    lin_confl = linearConflict(state)
    return manhattan +2*lin_confl



def memoize(f):
    """
    Memoizes the heuristic functions

    Insipiration from this memoized function came from
    https://www.python-course.eu/python3_memoization.php

    A memo_dic is created to store the results of the heuristic
    functions.

    The state and the memo dic are passed to the heuristic
    function, which goes to the memoize decorator.

    The helper function converts the state into a tuple
    in order to be added to the memo_dic. It then checks
    if the state is already in the memo_dic. If not, it adds
    the output of the heuristic function to the memo_dic. It
    then returns the heuristic value of the state from the memo_dic.

    """
    def helper(state,memo_dic):
        tuple_state =tuple(tuple(p) for p in state)
        if tuple_state not in memo_dic:
            memo_dic[tuple_state] = f(state)
        return memo_dic[tuple_state]
```

```
    return helper


h0_memo = memoize(h0)
h1_memo = memoize(h1)

heuristics = [h0,h1,h2]



def goalTest(state):
    """
    Tests if the current state is the goal state for any NxN board.
    """
    edge = len(state)
    state_np = np.array(state)
    goal_np = np.array([[edge*j+i for i in range(edge)] for j in range(edge)])
    return np.array_equal(goal_np,state_np)


def solvePuzzle(n, state, heuristic, prnt, memo_dic = None):
    """
    Uses the A* Search Algorithm to solve the NxN puzzle.
    The A* algorithm is a form of best-first search whose frontier
    is organized by fval, which is the gval (cost of actions to get
    to the state) + hval, which is the cost of a heuristic function.
    The heuristic function estimates the cost to get to the goal
    state from the state. This allows the algorithm
    to expand nodes that are closer to the goal state first, which
    leads to a dramatic increase in speed.

    This function takes as inputs:

    n - the puzzle dimension (i.e. n x n board)

    state - the starting (scrambled) state of the puzzle, provided
        as a list of lists, with the blank space represented by the
        number 0. For example, for n=3, we could have
        state = [[7 2 4],[5 0 6],[8 3 1]] as in the
```

```
    image shown previously.

heuristic - a handle to a heuristic function. The heuristic
    functions are labeled from 0 up, and are stored in
    a list called heuristics.

prnt - a boolean value that indicates whether
    or not to print the solution

memo_dic - If None, do nothing. If not none, an
    empty or full dictionary which is used in the
    memoization process. The keys are the tuple state
    and the values are the output of a given heuristic
    function at that state

This function returns:

steps - the number of steps to optimally reach the goal
    state from the initial state

frontierSize - the maximum (i.e. worst-case) size of
    the frontier during the search

err - an error code.
    'Error Code - 1' means the state is either the wrong shape
    or does not have the correct numbers

    'Error Code - 2' means that the state is unsolvable

(If prnt is true, the function does not return anything
but instead prints the full path along with with number of steps
and the maximum frontier size)

"""
# Checks if the state is the right shape,
# has the correct numbers, and is solveable
if errorTestOne(n,state):
    return 0,0,"Error Code - 1"
if not isSolveable(state):
```

```
        return 0,0,'Error Code - 2'

    # Creates a memo variable for use later
    memo = False
    if memo_dic != None:
        memo = True

    # Creates a PuzzleNode object with the correct
    # state and size. Starts with a gval of zero (gval
    # is the number of moves to reach this state) and
    # with the appropriate fval, based on the inputted
    # heuristic (and includes the memo option if stipulated)
    if memo == False:
        node = PuzzleNode(edge = n, state = state,
                          gval = 0,
                          fval= heuristic(state))
    else:
        node = PuzzleNode(edge = n, state = state,
                          gval = 0,
                          fval= heuristic(state,memo_dic))

    # Initiates the frontier as a priority queue
    # and puts the starting node in it
    # Remember: The priority queue is organized
    # around the fval of the nodes, as we
    # specified the __lt__ method, which is called
    # when comparing two objects.
    frontier = PriorityQueue()
    frontier.put(node)

    # Initiates the costs database, which is used
    # to see if a node has already been visited
    # and if so, if it had a lower gval
    costs_db = {node.tuple_state(): node}

    # Initiates a counter for the largest the
    # frontier ever gets
    frontier_max_size = frontier.qsize()
```

```
    # Starts the main loop
    while not frontier.empty():

        # takes the node with the higest priority from the
        # queue. This is the node with the lowest fval. The
        # fval is the gval + hval, and so the node with the lowest
        # fval is that which is both estimated to be close to the
        # goal state and close to get to from the current state.
        cur_node = frontier.get()

        # If the state has already been found with a lower
        # gval, ignore the node
        if cur_node.pruned:
            continue

        # Carries out the goal test
        if goalTest(cur_node.state):

            # If the goal test is true, the path
            # is reconstructed from the parents of the nodes
            path = [cur_node]
            while cur_node.parent:
                path.append(cur_node.parent)
                cur_node = cur_node.parent

                # If prnt is False, returns the
                # length of the path, (-1 because we are counting
                # moves, not states, so we shouldn't
                # count the initial state which is in our list)
            if not prnt:
                return len(path)-1, frontier_max_size ,0

                #If prnt is true, reverses the path
                # and prints each state in the correct order
            else:
                print_path = path[::-1]
                print ('There were {0} moves untill completion.'.format(len(pa
                print ('The largest the frontier got was {0} nodes.'.format(fr
                print ('The path was:')
```

```
            for step in print_path:
                print([list(step.state[i]) for i in range(len(step.state))
            return

    # iterates through all legal moves
    for move in cur_node.get_moves():

        # creates the new state the move will produce
        # and a tuple of that state for using to
        # interact with the cost_db
        new_state = cur_node.move(move)
        tuple_state = tuple(tuple(p) for p in new_state)

        #creates the gval for the child node, which is 1
        # more that the parent
        gval = cur_node.gval +1

        # if the state potential child is already in the
        # costs_db, we have already seen it
        if tuple_state in costs_db:

            # If the new child has a lower gval than the
            # old node with the same state, prune the old
            # node
            if costs_db[tuple_state].gval > child.gval:
                costs_db[tuple_state].pruned = True
            else:
                # If not, abandon the potential child
                continue

    # Find the heuristic value of the child
    # Use the memoized heuristic if specified
    if not memo:
        hval = heuristic(new_state)
    else:
        hval = heuristic(new_state,memo_dic)

    # Creates the child node, with the new state
    # gval, fval = gval + hval, and the appropriate parent
```

```
            child = PuzzleNode(edge = n, state = new_state,
                       gval= gval,
                       fval= gval + hval,
                       parent = cur_node)

            # Adds the child to the frontier and also
            # updates the costs_db
            frontier.put(child)
            costs_db[child.tuple_state()] = child

        # Updates the largest the frontier gets if the current
        # frontier is the largest
        frontier_max_size = max(frontier_max_size, frontier.qsize())
```

```
## BASIC TEST

# Tests to see if the function works as specified
# and to see if the prnt option works

test0 = [[7,2,4],[5,0,6],[8,3,1]]

steps, frontierSize, err = solvePuzzle(3, test0, heuristics[1], False)
print ("The Initial Test took {0} steps with a max frontier size of {1}.\n".fo
print ('The Initial Test with "prnt" set to True returned :')
solvePuzzle(3, test0, heuristics[1], True)

## TEST ERROR 1

# Tests to see if the 'Error Code -1 ' functionality works

bad_test0 = [[100,2,4],[5,0,6],[8,3,1]]
bad_test1 = [[7,2,4,5],[0,6],[8,3,1]]
print ('\nThe two bad tests returned:')
steps, frontierSize, err = solvePuzzle(3, bad_test0, heuristics[1], False)
print ("Failed because of", err)
steps, frontierSize, err = solvePuzzle(3, bad_test1, heuristics[1], False)
print ("Failed because of", err)
```

```
## TEST THE HEURISTICS

# Tests to see if the heuristic functions output the
# correct values

test0 = [[7,2,4],[5,0,6],[8,3,1]]
test1 = [[3,9,1,15],[14,11,4,6],[13,0,10,12],[2,7,8,5]]
assert h0(test0) == 8
assert h1(test0) == 18
print ('\nThe heuristic tests on 3x3 grids worked.')

assert h0(test1) == 14
assert h1(test1) == 41
print ('\nThe heuristic tests on 4x4 grids worked.')

## TEST THE isSolveable FUNCTION

# Tests to see if the isSolveable function
# makes the correct conclusions about solveable
# and unsolveable states

test0 =  [[7,2,4],[5,0,6],[8,3,1]]
bad_test0 = [[2,7,4],[5,0,6],[8,3,1]]
test1 =[[3,9,1,15],[14,11,4,6],[13,0,10,12],[2,7,8,5]]
bad_test1 =[[9,3,1,15],[14,11,4,6],[13,0,10,12],[2,7,8,5]]

assert isSolveable(test0) == True
assert isSolveable(bad_test0) == False
assert isSolveable(test1) == True
assert isSolveable(bad_test1) == False

print ('\nThe isSolveable test worked.')
```

```
The Initial Test took 26 steps with a max frontier size of 1167.

The Initial Test with "prnt" set to False returned :
There were 26 moves untill completion.
The largest the frontier got was 1167 nodes.
The path was:
```

```
[[7, 2, 4], [5, 0, 6], [8, 3, 1]]
[[7, 2, 4], [0, 5, 6], [8, 3, 1]]
[[0, 2, 4], [7, 5, 6], [8, 3, 1]]
[[2, 0, 4], [7, 5, 6], [8, 3, 1]]
[[2, 5, 4], [7, 0, 6], [8, 3, 1]]
[[2, 5, 4], [7, 3, 6], [8, 0, 1]]
[[2, 5, 4], [7, 3, 6], [0, 8, 1]]
[[2, 5, 4], [0, 3, 6], [7, 8, 1]]
[[2, 5, 4], [3, 0, 6], [7, 8, 1]]
[[2, 5, 4], [3, 6, 0], [7, 8, 1]]
[[2, 5, 0], [3, 6, 4], [7, 8, 1]]
[[2, 0, 5], [3, 6, 4], [7, 8, 1]]
[[0, 2, 5], [3, 6, 4], [7, 8, 1]]
[[3, 2, 5], [0, 6, 4], [7, 8, 1]]
[[3, 2, 5], [6, 0, 4], [7, 8, 1]]
[[3, 2, 5], [6, 4, 0], [7, 8, 1]]
[[3, 2, 5], [6, 4, 1], [7, 8, 0]]
[[3, 2, 5], [6, 4, 1], [7, 0, 8]]
[[3, 2, 5], [6, 0, 1], [7, 4, 8]]
[[3, 2, 5], [6, 1, 0], [7, 4, 8]]
[[3, 2, 0], [6, 1, 5], [7, 4, 8]]
[[3, 0, 2], [6, 1, 5], [7, 4, 8]]
[[3, 1, 2], [6, 0, 5], [7, 4, 8]]
[[3, 1, 2], [6, 4, 5], [7, 0, 8]]
[[3, 1, 2], [6, 4, 5], [0, 7, 8]]
[[3, 1, 2], [0, 4, 5], [6, 7, 8]]
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]

The two bad tests returned:
Failed because of Error Code - 1
Failed because of Error Code - 1

The heuristic tests on 3x3 grids worked.

The heuristic tests on 4x4 grids worked.

The isSolveable test worked.
```

```
test0 = [[7,2,4],[5,0,6],[8,3,1]]
```

```
test1 = [[7,0,8],[4,6,1],[5,3,2]]
test2 = [[2,3,7],[1,8,0],[6,5,4]]
tests = [test0,test1,test2]

heuristics = [h0,h1,h2]

for i in range(len(heuristics)):
    print ('\n\nHeuristic {0}'.format(i))
    for j in range(len(tests)):
        print ('Test {0}'.format(j))
        steps, frontierSize, err = solvePuzzle(3, tests[j], heuristics[i], Fal
        print ("Took {0} steps with a max frontier size of {1}.\n".format(step
```

```
Heuristic 0
Test 0
Took 26 steps with a max frontier size of 15253.

Test 1
Took 25 steps with a max frontier size of 12573.

Test 2
Took 17 steps with a max frontier size of 558.



Heuristic 1
Test 0
Took 26 steps with a max frontier size of 1167.

Test 1
Took 25 steps with a max frontier size of 1341.

Test 2
Took 17 steps with a max frontier size of 75.



Heuristic 2
Test 0
```

```
Took 26 steps with a max frontier size of 719.

Test 1
Took 25 steps with a max frontier size of 433.

Test 2
Took 17 steps with a max frontier size of 74.
```

I'll first explain why the second heuristic function dominates the other two. It is already given that the Manhattan distance dominates missing tiles, so I just have to show this dominates the Manhattan distance. The second heuristic functions is Manhattan Distance +(Linear Conflits x2), because this is the minimum number of moves that must happen, as we both must habe the tiles move back to there original spot (the manhattan distance), and tiles must move out of the way of others and back again (the linear conflicts). Thus, the heuristic is always greater for H2 than H1.

The results of the test confirm this. All 3 heuristic functions arrived at the same number of steps for each test. Each time, heuristic 1 beat heuristic 0 in frontier size (and subsequently time) and each time heuristic 2 beat both of the others. Thus, we can conclude that heuristic 2 is the optimal heuristic for our purposes.

```
### MEMOIZING TEST

# Tests the basic heuristic functions
# on the various test cases for speed.

# Each heuristic function is tested in its
# normal version, a memoized version with an
# empty memo_dic, and a memoized version with
# a full memo_dic.

heuristics = [h0,h1,h0_memo,h1_memo]

test0 = [[7,2,4],[5,0,6],[8,3,1]]
test1 = [[7,0,8],[4,6,1],[5,3,2]]
test2 = [[2,3,7],[1,8,0],[6,5,4]]
tests = [test0,test1,test2]
repeats = 5
```

```
for j in range(int(len(heuristics)/2)):
    print ("Heuristic{0} :".format(j))
    for i in range(len(tests)):
        print ("\nTest{0} :".format(i))
        start = time()
        for k in range(repeats):
            steps, frontierSize, err = solvePuzzle(3, tests[i],
                                                   heuristics[j], prnt = False
        end = time()
        print ('\nNormal Heuristic:')
        print ('On average, it took {0} seconds to execute.'.format(round((end

        start = time()
        for k in range(repeats):
            memo_dic_0 = {}
            steps, frontierSize, err = solvePuzzle(3, tests[i],
                                                   heuristics[j+2], prnt = Fal
        end = time()
        print ('\nMemo Heuristic with empty memo_dic:')
        print ('On average, it took {0} seconds to execute.'.format(round((end

        memo_dic_0 = {}
        start = time()
        for k in range(repeats):
            steps, frontierSize, err = solvePuzzle(3, tests[i],
                                                   heuristics[j+2], prnt = Fal
        end = time()
        print ('\nMemo Heuristic with full memo_dic:')
        print ('On average, it took {0} seconds to execute.'.format(round((end
    print('\n')
```

```
Heuristic0 :

Test0 :

Normal Heuristic:
On average, it took 4.027 seconds to execute.
```

```
Memo Heuristic with empty memo_dic:
On average, it took 4.226 seconds to execute.

Memo Heuristic with full memo_dic:
On average, it took 3.885 seconds to execute.

Test1 :

Normal Heuristic:
On average, it took 3.135 seconds to execute.

Memo Heuristic with empty memo_dic:
On average, it took 3.363 seconds to execute.

Memo Heuristic with full memo_dic:
On average, it took 3.022 seconds to execute.

Test2 :

Normal Heuristic:
On average, it took 0.131 seconds to execute.

Memo Heuristic with empty memo_dic:
On average, it took 0.117 seconds to execute.

Memo Heuristic with full memo_dic:
On average, it took 0.115 seconds to execute.


Heuristic1 :

Test0 :

Normal Heuristic:
On average, it took 0.402 seconds to execute.

Memo Heuristic with empty memo_dic:
On average, it took 0.435 seconds to execute.
```

```
Memo Heuristic with full memo_dic:
On average, it took 0.257 seconds to execute.

Test1 :

Normal Heuristic:
On average, it took 0.464 seconds to execute.

Memo Heuristic with empty memo_dic:
On average, it took 0.484 seconds to execute.

Memo Heuristic with full memo_dic:
On average, it took 0.302 seconds to execute.

Test2 :

Normal Heuristic:
On average, it took 0.024 seconds to execute.

Memo Heuristic with empty memo_dic:
On average, it took 0.028 seconds to execute.

Memo Heuristic with full memo_dic:
On average, it took 0.019 seconds to execute.
```

The above cell tested the memoization of our basic heuristic functions. It runs each test 5 times and averages them. It first runs the normal heuristic, than the memoized version with an empty dictionary, then the memoized version with a full dic (its empty the first time but full after). We can see that speed implications of the memoization. Across the board, running the memoized version with an empty dictionary increases the running time, which makes sense as there is lots of additonal writing. However once we have a full dictionary with the heuristic outputs, we get a faster running time. This makes sense and shows the benfit of our memoization. It could be improved even more by running it with many different intial states to get an even larger memo_dic and then running it on a novel state.