# Name - Konark Raj Mishra

#CS550 - AI

# Professor Marie Roch

# Program 1

import random

import copy

import math

from basicsearch\_lib.board import Board

class TileBoard(Board):

def \_\_init\_\_(self, n, multiple\_solutions=False, force\_state=None,

verbose=False):

""""TileBoard(n, multiple\_solutions

Create a tile board for an n puzzle.

If multipleSolutions are true, the solution need not

have the space in the center. This defaults to False but

is automatically set to True when there is no middle square

force\_state can be used to initialize an n puzzle to a desired

configuration. No error checking is done. It is specified as

a list with n+1 elements in it, 1:n and None in the desired order.

verbose is a boolean for turning on debugging

"""

self.verbose = verbose # not debug state, up to you to use it

self.boardsize = int(math.sqrt(n+1))

self.originalsize = n+1

if math.sqrt(n+1) != self.boardsize:

raise ValueError("Bad board size\n" +

"Must be one less than an odd perfect square 8, 24, ...")

# initialize parent

super().\_\_init\_\_(self.boardsize, self.boardsize)

# Compute solution states

# todo: Set self.goals to a list of solution tuples

goals = [x for x in range(1, self.originalsize)]

goals.append(None)

if force\_state is not None:

# populate the board with the specified input

# self.board = add items specified to the board

self.board = \

[[force\_state.pop(0) for col in range(self.get\_cols())]

for row in range(self.get\_rows())]

else:

# generate random board

self.populate\_board()

self.random\_board()

while(not self.solved()) and (not self.solvable(self.board)):

self.random\_board()

# [(None,1,2,3,...), (1,None,2,3,...), (1,2,None,3,...)]

# Otherwise, must be the last square: [(1,2,3,...,None)]

# todo: Determine inital state and make sure that it is solvable

# todo: Populate the board using self.place

# It would be wise to track the empty square location as well

# as it will make action generation easier

def populate\_board(self):

# create a new\_board with original size of the board i.e 8+1

# loop goes through each tile and creates a new\_board replica

new\_board = [x for x in range(1, self.originalsize)]

new\_board.append(None)

# \ to continue logical statement on the next line

# add every element to the original board

self.board = \

[[new\_board.pop(0) for col in range(self.cols)]

for row in range(self.rows)]

def random\_board(self):

# cast new\_board into a tuple to get a solution state

# state tuple flattens the list out

# convert the flat tuple into a flat list

new\_board = list(self.state\_tuple())

# shuffle the board with random module

random.shuffle(new\_board)

# add items from the shuffled tuple to the original board

self.board = [[new\_board.pop(0) for col in range(self.cols)] for row in range(

self.rows)] # add items by popping them off the list

def solvable(self, tiles, verbose=False):

"""solvable - Determines if a puzzle is solvable

Given a list of tiles, determine if the N-puzzle is solvable.

You do not need to know how to do this, but the calculation

is based on the inversion order.

for each number in the list of tiles,

How many following numbers are less than that one

e.g. [13, 10, 11, 6, 5, 7, 4, 8, 1, 12, 14, 9, 3, 15, 2, None]

Example: Files following 9: [3, 15, 2, None]

Two of these are smaller than 9, so the inversion order

for 9 is 2

A puzzle's inversion order is the sum of the tile inversion

orders. For puzzles with even numbers of rows and columns,

the row number on which the blank resides must be added.

Note that we need not worry about 1 as there are

no tiles smaller than one.

See Wolfram Mathworld for further explanation:

http://mathworld.wolfram.com/15Puzzle.html

and http://www.cut-the-knot.org/pythagoras/fifteen.shtml

This lets us know if a problem can be solved. The inversion

order modulo 2 is invariant across moves. This means that

when we make a legal move, the inversion order will always

be even or odd. The solution state always has an even

inversion order, so any puzzle with an odd inversion

number cannot be solved.

"""

inversionorder = 0

# Make life easy, remove None

reduced = [t for t in tiles if t is not None]

# Loop over all but last (no tile after it)

for idx in range(len(reduced)-1):

value = reduced[idx]

after = reduced[idx+1:] # Remaining tiles

smaller = [x for x in after if x < value]

numtiles = len(smaller)

inversionorder = inversionorder + numtiles

if verbose:

print("idx {} value {} tail {} #smaller {} sum: {}".format(

idx, value, after, numtiles, inversionorder))

# Even number of rows must take the blank position into account

if self.get\_rows() % 2 == 0:

if verbose:

print("Even # rows, adding for position of blank")

inversionorder = inversionorder + \

math.floor(tiles.index(None) / self.boardsize)+1

solvable = inversionorder % 2 == 0 # Solvable if even

return solvable

def \_\_hash\_\_(self):

"\_\_hash\_\_ - Hash the board state"

# Convert state to a tuple and hash

return hash(self.state\_tuple())

def \_\_eq\_\_(self, other):

"\_\_eq\_\_ - Check if objects equal: a == b"

if self.rows != other.rows:

return False

for row in range(self.rows):

for col in range(self.cols):

if self.get(row, col) != other.get(row, col):

return False

return True

# todo: Determine if two board configurations are equivalent

def state\_tuple(self):

"state\_tuple - Return board state as a single tuple"

# create a new tuple

# add tuples to a tuple

new\_tuple = ()

for i in range(self.rows):

new\_tuple += tuple(self.board[i])

return new\_tuple

raise NotImplementedError(

"You must create a tuple based on the board state")

def get\_actions(self):

"Return row column offsets of where the empty tile can be moved"

# return list of lists to move on the board

blank\_index = self.blank\_find()

moves = []

"-1 left or down, 1 right or up, 0 stay"

if blank\_index[0] > 0:

# move left on the column

moves.append([-1, 0])

if blank\_index[0] < self.cols - 1:

# move right on the column

moves.append([1, 0])

if blank\_index[1] > 0:

# move up on the row

moves.append([0, -1])

if blank\_index[1] < self.rows - 1:

# move down on the row

moves.append([0, 1])

return moves

raise NotImplementedError("Return list of valid actions")

def move(self, offset):

"move - Move the empty space by [delta\_row, delta\_col] and return new board"

# Hint: Be sure to use deepcopy

new\_board = copy.deepcopy(self)

# find where None is and return the index

blank\_index = new\_board.blank\_find()

# return indices for moves

blank\_mov\_col = blank\_index[0] + offset[0]

blank\_mov\_row = blank\_index[1] + offset[1]

new\_board.board[blank\_mov\_row][blank\_mov\_col] = None

new\_board.board[blank\_index[1]][blank\_index[0]

] = self.board[blank\_mov\_row][blank\_mov\_col]

return new\_board

raise NotImplementedError("Return new TileBoard with action applied")

def solved(self):

goal\_list = [x for x in range(1, self.originalsize)]

goal\_list.append(None)

goal\_list\_middle = [x for x in range(1, self.originalsize-1)]

goal\_list\_middle.append(None)

goal\_list\_middle.append(self.originalsize-1)

goal\_list\_right = [x for x in range(1, self.originalsize-2)]

goal\_list\_right.append(None)

goal\_list\_right.append(self.originalsize-2)

goal\_list\_right.append(self.originalsize-1)

return self.state\_tuple() == tuple(goal\_list) or self.state\_tuple() == tuple(goal\_list\_middle) or self.state\_tuple() == tuple(goal\_list\_right)

def blank\_find(self):

for row in range(self.get\_rows()):

for col in range(self.get\_cols()):

if self.board[row][col] is None:

return col, row