"""

This file contains the `Board` class, which implements the rules for the

game Isolation as described in lecture, modified so that the players move

like knights in chess rather than queens.

You MAY use and modify this class, however ALL function signatures must

remain compatible with the defaults provided, and none of your changes will

be available to project reviewers.

"""

import timeit

from copy import deepcopy

from copy import copy

TIME\_LIMIT\_MILLIS = 200

class Board(object):

"""

Implement a model for the game Isolation assuming each player moves like

a knight in chess.

Parameters

----------

player\_1 : object

An object with a get\_move() function. This is the only function

directly called by the Board class for each player.

player\_2 : object

An object with a get\_move() function. This is the only function

directly called by the Board class for each player.

width : int (optional)

The number of columns that the board should have.

height : int (optional)

The number of rows that the board should have.

"""

BLANK = 0

NOT\_MOVED = None

def \_\_init\_\_(self, player\_1, player\_2, width=7, height=7):

self.width = width

self.height = height

self.move\_count = 0

self.\_\_player\_1\_\_ = player\_1

self.\_\_player\_2\_\_ = player\_2

self.\_\_active\_player\_\_ = player\_1

self.\_\_inactive\_player\_\_ = player\_2

self.\_\_board\_state\_\_ = [[Board.BLANK for i in range(width)] for j in range(height)]

self.\_\_last\_player\_move\_\_ = {player\_1: Board.NOT\_MOVED, player\_2: Board.NOT\_MOVED}

self.\_\_player\_symbols\_\_ = {Board.BLANK: Board.BLANK, player\_1: 1, player\_2: 2}

@property

def active\_player(self):

"""

The object registered as the player holding initiative in the

current game state.

"""

return self.\_\_active\_player\_\_

@property

def inactive\_player(self):

"""

The object registered as the player in waiting for the current

game state.

"""

return self.\_\_inactive\_player\_\_

def get\_opponent(self, player):

"""

Return the opponent of the supplied player.

Parameters

----------

player : object

An object registered as a player in the current game. Raises an

error if the supplied object is not registered as a player in

this game.

Returns

----------

object

The opponent of the input player object.

"""

if player == self.\_\_active\_player\_\_:

return self.\_\_inactive\_player\_\_

elif player == self.\_\_inactive\_player\_\_:

return self.\_\_active\_player\_\_

raise RuntimeError("`player` must be an object registered as a player in the current game.")

def copy(self):

""" Return a deep copy of the current board. """

new\_board = Board(self.\_\_player\_1\_\_, self.\_\_player\_2\_\_, width=self.width, height=self.height)

new\_board.move\_count = self.move\_count

new\_board.\_\_active\_player\_\_ = self.\_\_active\_player\_\_

new\_board.\_\_inactive\_player\_\_ = self.\_\_inactive\_player\_\_

new\_board.\_\_last\_player\_move\_\_ = copy(self.\_\_last\_player\_move\_\_)

new\_board.\_\_player\_symbols\_\_ = copy(self.\_\_player\_symbols\_\_)

new\_board.\_\_board\_state\_\_ = deepcopy(self.\_\_board\_state\_\_)

return new\_board

def forecast\_move(self, move):

"""

Return a deep copy of the current game with an input move applied to

advance the game one ply.

Parameters

----------

move : (int, int)

A coordinate pair (row, column) indicating the next position for

the active player on the board.

Returns

----------

`isolation.Board`

A deep copy of the board with the input move applied.

"""

new\_board = self.copy()

new\_board.apply\_move(move)

return new\_board

def move\_is\_legal(self, move):

"""

Test whether a move is legal in the current game state.

Parameters

----------

move : (int, int)

A coordinate pair (row, column) indicating the next position for

the active player on the board.

Returns

----------

bool

Returns True if the move is legal, False otherwise

"""

row, col = move

return 0 <= row < self.height and \

0 <= col < self.width and \

self.\_\_board\_state\_\_[row][col] == Board.BLANK

def get\_blank\_spaces(self):

"""

Return a list of the locations that are still available on the board.

"""

return [(i, j) for j in range(self.width) for i in range(self.height)

if self.\_\_board\_state\_\_[i][j] == Board.BLANK]

def get\_player\_location(self, player):

"""

Find the current location of the specified player on the board.

Parameters

----------

player : object

An object registered as a player in the current game.

Returns

----------

(int, int)

The coordinate pair (row, column) of the input player.

"""

return self.\_\_last\_player\_move\_\_[player]

def get\_legal\_moves(self, player=None):

"""

Return the list of all legal moves for the specified player.

Parameters

----------

player : object (optional)

An object registered as a player in the current game. If None,

return the legal moves for the active player on the board.

Returns

----------

list<(int, int)>

The list of coordinate pairs (row, column) of all legal moves

for the player constrained by the current game state.

"""

if player is None:

player = self.active\_player

return self.\_\_get\_moves\_\_(self.\_\_last\_player\_move\_\_[player])

def apply\_move(self, move):

"""

Move the active player to a specified location.

Parameters

----------

move : (int, int)

A coordinate pair (row, column) indicating the next position for

the active player on the board.

Returns

----------

None

"""

row, col = move

self.\_\_last\_player\_move\_\_[self.active\_player] = move

self.\_\_board\_state\_\_[row][col] = self.\_\_player\_symbols\_\_[self.active\_player]

self.\_\_active\_player\_\_, self.\_\_inactive\_player\_\_ = self.\_\_inactive\_player\_\_, self.\_\_active\_player\_\_

self.move\_count += 1

def is\_winner(self, player):

""" Test whether the specified player has won the game. """

return player == self.inactive\_player and not self.get\_legal\_moves(self.active\_player)

def is\_loser(self, player):

""" Test whether the specified player has lost the game. """

return player == self.active\_player and not self.get\_legal\_moves(self.active\_player)

def utility(self, player):

"""

Returns the utility of the current game state from the perspective

of the specified player.

/ +infinity, "player" wins

utility = | -infinity, "player" loses

\ 0, otherwise

Parameters

----------

player : object (optional)

An object registered as a player in the current game. If None,

return the utility for the active player on the board.

Returns

----------

float

The utility value of the current game state for the specified

player. The game has a utility of +inf if the player has won,

a value of -inf if the player has lost, and a value of 0

otherwise.

"""

if not self.get\_legal\_moves(self.active\_player):

if player == self.inactive\_player:

return float("inf")

if player == self.active\_player:

return float("-inf")

return 0.

def \_\_get\_moves\_\_(self, move):

"""

Generate the list of possible moves for an L-shaped motion (like a

knight in chess).

"""

if move == Board.NOT\_MOVED:

return self.get\_blank\_spaces()

r, c = move

directions = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),

(1, -2), (1, 2), (2, -1), (2, 1)]

valid\_moves = [(r+dr,c+dc) for dr, dc in directions if self.move\_is\_legal((r+dr, c+dc))]

return valid\_moves

def print\_board(self):

"""DEPRECATED - use Board.to\_string()"""

return self.to\_string()

def to\_string(self):

"""Generate a string representation of the current game state, marking

the location of each player and indicating which cells have been

blocked, and which remain open.

"""

p1\_loc = self.\_\_last\_player\_move\_\_[self.\_\_player\_1\_\_]

p2\_loc = self.\_\_last\_player\_move\_\_[self.\_\_player\_2\_\_]

out = ''

for i in range(self.height):

out += ' | '

for j in range(self.width):

if not self.\_\_board\_state\_\_[i][j]:

out += ' '

elif p1\_loc and i == p1\_loc[0] and j == p1\_loc[1]:

out += '1'

elif p2\_loc and i == p2\_loc[0] and j == p2\_loc[1]:

out += '2'

else:

out += '-'

out += ' | '

out += '\n\r'

return out

def play(self, time\_limit=TIME\_LIMIT\_MILLIS):

"""

Execute a match between the players by alternately soliciting them

to select a move and applying it in the game.

Parameters

----------

time\_limit : numeric (optional)

The maximum number of milliseconds to allow before timeout

during each turn.

Returns

----------

(player, list<[(int, int),]>, str)

Return multiple including the winning player, the complete game

move history, and a string indicating the reason for losing

(e.g., timeout or invalid move).

"""

move\_history = []

curr\_time\_millis = lambda: 1000 \* timeit.default\_timer()

while True:

legal\_player\_moves = self.get\_legal\_moves()

game\_copy = self.copy()

move\_start = curr\_time\_millis()

time\_left = lambda : time\_limit - (curr\_time\_millis() - move\_start)

curr\_move = self.active\_player.get\_move(game\_copy, legal\_player\_moves, time\_left)

move\_end = time\_left()

# print move\_end

if curr\_move is None:

curr\_move = Board.NOT\_MOVED

if self.active\_player == self.\_\_player\_1\_\_:

move\_history.append([curr\_move])

else:

move\_history[-1].append(curr\_move)

if move\_end < 0:

return self.\_\_inactive\_player\_\_, move\_history, "timeout"

if curr\_move not in legal\_player\_moves:

return self.\_\_inactive\_player\_\_, move\_history, "illegal move"

self.apply\_move(curr\_move)