Program 5

from copy import deepcopy

import numpy as np

import time

# takes the input of current states and evaluvates the best path to goal state

def bestsolution(state):

bestsol = np.array([], int).reshape(-1, 9)

count = len(state) - 1

while count != -1:

bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)

count = (state[count]['parent'])

return bestsol.reshape(-1, 3, 3)

# this function checks for the uniqueness of the iteration(it) state, weather it has been previously traversed or not.

def all(checkarray):

set=[]

for it in set:

for checkarray in it:

return 1

else:

return 0

# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state

def manhattan(puzzle, goal):

a = abs(puzzle // 3 - goal // 3)

b = abs(puzzle % 3 - goal % 3)

mhcost = a + b

return sum(mhcost[1:])

# will calcuates the number of misplaced tiles in the current state as compared to the goal state

def misplaced\_tiles(puzzle,goal):

mscost = np.sum(puzzle != goal) - 1

return mscost if mscost > 0 else 0

#3[on\_true] if [expression] else [on\_false]

# will indentify the coordinates of each of goal or initial state values

def coordinates(puzzle):

pos = np.array(range(9))

for p, q in enumerate(puzzle):

pos[q] = p

return pos

# start of 8 puzzle evaluvation, using Manhattan heuristics

def evaluvate(puzzle, goal):

steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8], 3),('left', [0, 3, 6], -1),('right', [2, 5, 8], 1)],

dtype = [('move', str, 1),('position', list),('head', int)])

dtstate = [('puzzle', list),('parent', int),('gn', int),('hn', int)]

# initializing the parent, gn and hn, where hn is manhattan distance function call

costg = coordinates(goal)

parent = -1

gn = 0

hn = manhattan(coordinates(puzzle), costg)

state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.

dtpriority = [('position', int),('fn', int)]

priority = np.array( [(0, hn)], dtpriority)

while 1:

priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

position, fn = priority[0]

priority = np.delete(priority, 0, 0)

# sort priority queue using merge sort,the first element is picked for exploring remove from queue what we are exploring

puzzle, parent, gn, hn = state[position]

puzzle = np.array(puzzle)

# Identify the blank square in input

blank = int(np.where(puzzle == 0)[0])

gn = gn + 1

c = 1

start\_time = time.time()

for s in steps:

c = c + 1

if blank not in s['position']:

# generate new state as copy of current

openstates = deepcopy(puzzle)

openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]

# The all function is called, if the node has been previously explored or not

if ~(np.all(list(state['puzzle']) == openstates, 1)).any():

end\_time = time.time()

if (( end\_time - start\_time ) > 2):

print(" The 8 puzzle is unsolvable ! \n")

exit

# calls the manhattan function to calcuate the cost

hn = manhattan(coordinates(openstates), costg)

# generate and add new state in the list

q = np.array([(openstates, position, gn, hn)], dtstate)

state = np.append(state, q, 0)

# f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal state

fn = gn + hn

q = np.array([(len(state) - 1, fn)], dtpriority)

priority = np.append(priority, q, 0)

# Checking if the node in openstates are matching the goal state.

if np.array\_equal(openstates, goal):

print(' The 8 puzzle is solvable ! \n')

return state, len(priority)

return state, len(priority)