Eliahou Mayor

A\* 8 Tile Puzzle program

To run the program, just run:

python3 A\*.py

from the directory that the program is in, or open it up in your favorite IDE and hit run.

The program will ask you for the path to the input file, enter it and then hit enter.

The program will then ask you to choose between the Manhattan Distance Heuristic or 2\* Linear Conflicts + the Manhattan Distance Heuristic. Enter A for the first or B for the second, followed by the enter key.

The program will then produce the solution output file in the directory it is running in, with the title solution.txt. If you wish to keep this file, make sure you rename it before running the program again.

SOURCE CODE:

# A\* Search Algorithm for 8 Square Problem

from enum import Enum

from queue import PriorityQueue

# Different Actions you can take

class Actions(Enum):

U = 1

D = 2

L = 3

R = 4

# function reads the initial and goal states and allows the user to select from 2 heuristics

def readProblem(initialState, goalState):

print("Enter the path for the file that contains the problem:")

filepath = input()

f = open(filepath, "r") # open file

readInitialState = False # bool whether we read initial state yet

# one loop to read both initial and goal states

for line in f:

if(line == "\n"):

readInitialState = True # new line between the states, means its time to read the goal state

continue

a,b,c = line.split() # split 3 numbers and assign them to array

if(readInitialState):

# read initial state already so add 3 numbers to goal state

goalState.append(int(a))

goalState.append(int(b))

goalState.append(int(c))

else:

# add 3 states to initial state

initialState.append(int(a))

initialState.append(int(b))

initialState.append(int(c))

f.close() # close file

algorithm = input("Enter A to use the Manhattan Distances Heuristic or B to use the 2\* Linear Conflicts + Manhattan Distance Heuristic: \n")

return algorithm

#return 1

# This function applies an action to a state and returns a boolean whether the action was valid or not

def applyAction(action, state):

actionLegal = True # bool for whether valid action

i = state.index(0) # find the index of the empty tile

if action == Actions.U:

if i - 3 < 0: #out of bounds, you are on the top row, cannot go up

actionLegal = False

return actionLegal

state[i],state[i-3] = state[i-3],state[i] # swap empty tile with one on top of it

elif action == Actions.D:

if i + 3 > 8: #out of bounds, you are on the bottom row, cannot go down

actionLegal = False

return actionLegal

state[i],state[i+3] = state[i+3],state[i] # swap empty tile with one on below it

elif action == Actions.L:

if i == 0 or i == 3 or i == 6: #out of bounds, you are on the leftmost column, cannot go left

actionLegal = False

return actionLegal

state[i],state[i-1] = state[i-1],state[i] # swap empty tile with one on to the left of it

else:

if i == 2 or i == 5 or i == 8: #out of bounds, you are on the rightmost column, cannot go right

actionLegal = False

return actionLegal

state[i],state[i+1] = state[i+1],state[i] # swap empty tile with one on to the right of it

return actionLegal

# calculate manhattan distance heuristic of state and return it

def heuristicManhattanDistance(state,goalState):

sum = 0

for i in range(1,9): # for each tile from 1-9

distance = abs(state.index(i) - goalState.index(i)) # find the distance between the indices from initial to goal state

sum += (distance // 3) + (distance % 3) # go as up/down as you can and as left/right as you can (tells you number of moves necessary in either direction)

return sum

# calculate 2\*linear conflicts + manhattan distance

def heuristicLinearConflict(state,goalState):

manhattanDistance = heuristicManhattanDistance(state,goalState)

stateLines = []

goalLines = []

# get all the columns into the list "Lines"

i = 0

while i < 9:

stateLines.append([state[i],state[i+1],state[i+2]])

goalLines.append([goalState[i],goalState[i+1],goalState[i+2]])

i+=3

# get all the rows into the list "Lines"

for i in range(3):

stateLines.append([state[0+i],state[3+i],state[6+i]])

goalLines.append([goalState[0+i],goalState[3+i],goalState[6+i]])

# count the number of linear conflicts

linearConflicts = 0

for i in range(len(goalLines)): # for each row/column

intersection = []

for number in stateLines[i]:

if number in goalLines[i] and number != 0: # find all the numbers in the row/column that appear in both initial and goal state

dict1 = {}

dict1["number"] = number # record the tile number

dict1["stateIndex"] = stateLines[i].index(number) # record the position in current state for that value

dict1["goalIndex"] = goalLines[i].index(number) # record the position in goal state for that value

intersection.append(dict1) # add the dict to intersection list

if(len(intersection) > 1): # if there is more than 1 intersection (2 numbers or more that are in both lines)

j = k = 0

# for each pair of numbers

for j in range(len(intersection)):

for k in range(j+1,len(intersection)):

# if the order between the 2 numbers are reversed between the current state and the goal state

if(intersection[j]["stateIndex"] < intersection[k]["stateIndex"] and intersection[j]["goalIndex"] > intersection[k]["goalIndex"]):

linearConflicts +=1

return manhattanDistance + (linearConflicts\*2)

# returns the value of right heuristic based on user's initial input for that state

def algorithm(algo,state,goalState):

if algo == "A":

return heuristicManhattanDistance(state,goalState)

elif algo == "B":

return heuristicLinearConflict(state,goalState)

# writes the solution to a file formatted accordingly

def writeSolution(initialState, goalState,depth,nodesGenerated,moves,fn):

f = open("solution.txt","w+")

i = 0

# print initial state

for number in initialState:

if i == 2 or i == 5 or i ==8 :

f.write(str(number) + "\n")

else:

f.write(str(number) + " ")

i += 1

f.write("\n")

i = 0

# print goal state

for number in goalState:

if i == 2 or i == 5 or i ==8 :

f.write(str(number) + "\n")

else:

f.write(str(number) + " ")

i += 1

f.write("\n")

# print depth

f.write(str(depth))

f.write("\n")

# print number of nodes generated

f.write(str(nodesGenerated))

f.write("\n")

# print the moves to get from initial to goal state

f.write(" ".join(moves))

f.write("\n")

# print the f(n) values for each state on the path

f.write(" ".join(fn))

f.write("\n")

f.close()

def AStar():

"""

Runs the algorithm

"""

# empty arrays for initial state

initialState = []

goalState= []

algo= readProblem(initialState, goalState) # read the problem

came\_from = {} # dict to remember which state came from which

cost\_so\_far = {} # dict to remember the depth of each node

initialPriority = 0 + algorithm(algo,initialState,goalState) # calculate priority

came\_from[str(initialState)] = {"state": None, "Action" : None, "Priority:" : initialPriority} # put initial state in dict with no parent state, no action and no priority since it's the initial state

cost\_so\_far[str(initialState)] = 0 # cost to get to initial is 0

nodesGenerated = 1 # count number of nodes generated

frontier = PriorityQueue() # queue ordered on priority

frontier.put((initialPriority,initialState)) # put initial state in the queue

while not frontier.empty():

state = frontier.get()[1] # get smallest f(n) value in queue

#print("retrieved state: ")

#print(state)

if state == goalState: # if goal, we are done

break

for action in Actions: # try every action

prevCost = cost\_so\_far[str(state)] # cost to get here

nextState = state.copy() # make a copy to be the next state

if( not applyAction(action,nextState)): # if next action is not legal

continue # just skip it

prevCost += 1 # +1 action to get from previous state to next state

# if we didnt get here or the cost to get here is lower than the other path we took to get to this state

if str(nextState) not in cost\_so\_far or prevCost < cost\_so\_far[str(nextState)]:

nodesGenerated += 1

priority = prevCost + algorithm(algo,nextState,goalState) # calculate priority

#print(priority)

cost\_so\_far[str(nextState)] = prevCost # place depth

frontier.put((priority,nextState)) # put it in the queue

came\_from[str(nextState)] = {"state": state, "Action" : action, "Priority" : priority} # remember where we came from and how

# if we made it here, we are done so process the output

curr = state

moves = [came\_from[str(curr)]["Action"].name] # get last move in array

fn = [str(came\_from[str(curr)]["Priority"])] # get last f(n) in array

while curr != initialState: # iterate backwards through dict tracing back the states to get action and f(n)

curr = came\_from[str(curr)]["state"]

if came\_from[str(curr)]["Action"] != None: # if you arent the root

moves.append(came\_from[str(curr)]["Action"].name) # add the action to array

fn.append(str(came\_from[str(curr)]["Priority"])) # add the f(n)

fn.append(str(initialPriority)) # add the f(n) for the root

# reverse the array to correct order

moves.reverse()

fn.reverse()

depth = cost\_so\_far[str(state)] # get depth from dict

writeSolution(initialState, goalState,depth,nodesGenerated,moves,fn) # format solution in file

return # we are done

if \_\_name\_\_ == "\_\_main\_\_":

AStar() # run AStar algo

Output File1 A

7 1 6

8 3 5

2 0 4

8 7 6

1 0 5

2 3 4

5

12

U U L D R

5 5 5 5 5 5

Output File1 B

7 1 6

8 3 5

2 0 4

8 7 6

1 0 5

2 3 4

5

12

U U L D R

5 5 5 5 5 5

Output File2 A

2 6 0

1 3 4

7 5 8

1 2 3

4 5 6

7 8 0

10

33

L D R U L L D R D R

10 10 10 10 10 10 10 10 10 10 10

Output File2 B

2 6 0

1 3 4

7 5 8

1 2 3

4 5 6

7 8 0

10

30

L D R U L L D R D R

10 10 10 10 10 10 10 10 10 10 10

Output File3 A

5 4 3

2 6 7

1 8 0

1 2 3

4 5 6

7 8 0

22

1471

U L U L D D R U U L D D R R U L L D R U R D

10 12 12 12 12 12 12 14 16 16 16 16 18 18 18 20 22 22 22 22 22 22 22

Output File3 B

5 4 3

2 6 7

1 8 0

1 2 3

4 5 6

7 8 0

22

866

U L U L D D R U U L D D R R U L L D R U R D

10 14 14 14 14 14 14 16 18 18 18 18 18 18 18 20 22 22 22 22 22 22 22

Output File4 A

8 7 3

0 4 5

6 2 1

1 2 3

4 5 6

7 8 0

23

1378

U R D D R U L D L U U R D R D L L U U R D R D

15 15 15 17 17 17 19 21 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23

Output File4 B

8 7 3

0 4 5

6 2 1

1 2 3

4 5 6

7 8 0

23

851

U R D D R U L D L U U R D R D L L U U R D R D

15 15 15 17 17 17 19 21 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23