**Exercise#01**

import heapq

def astar(graph, start, goal, heuristic):

"""

Find the shortest path from start to goal in a weighted graph using A\* search algorithm.

:param graph: the weighted graph (dictionary of vertices and their edges with weights)

:param start: the starting vertex

:param goal: the goal vertex

:param heuristic: the heuristic function (dictionary of estimated distances from each vertex to the goal)

:return: the shortest path from start to goal

"""

queue = [(0, [start])]

visited = set()

while queue:

(cost, path) = heapq.heappop(queue)

vertex = path[-1]

if vertex == goal:

return path

if vertex in visited:

continue

visited.add(vertex)

for (neighbor, edge\_cost) in graph[vertex]:

if neighbor not in visited:

neighbor\_cost = cost + edge\_cost

neighbor\_heuristic = heuristic[neighbor]

neighbor\_f = neighbor\_cost + neighbor\_heuristic

neighbor\_path = path + [neighbor]

heapq.heappush(queue, (neighbor\_f, neighbor\_path))

return None

graph = {

'Ar': [('Zerind', 75), ('Sibiu', 140), ('Timisoara', 118)],

'Zerind': [('Ar', 75), ('Oradea', 71)],

'Oradea': [('Zerind', 71), ('Sibiu', 151)],

'Sibiu': [('Ar', 140), ('Oradea', 151), ('Fagaras', 99), ('Rimnicu Vilcea', 80)],

'Timisoara': [('Ar', 118), ('Lugoj', 111)],

'Lugoj': [('Timisoara', 111), ('Mehadia', 70)],

'Mehadia': [('Lugoj', 70), ('Drobeta', 75)],

'Drobeta': [('Mehadia', 75), ('Craiova', 120)],

'Craiova': [('Drobeta', 120), ('Rimnicu Vilcea', 146), ('Pitesti', 138)],

'Rimnicu Vilcea': [('Sibiu', 80), ('Craiova', 146), ('Pitesti', 97)],

'Fagaras': [('Sibiu', 99), ('Bucharest', 211)],

'Pitesti': [('Rimnicu Vilcea', 97), ('Craiova', 138), ('Bucharest', 101)],

'Bucharest': [('Fagaras', 211), ('Pitesti', 101), ('Giurgiu', 90), ('Urziceni', 85)],

'Giurgiu': [('Bucharest', 90)],

'Urziceni': [('Bucharest', 85), ('Hirsova', 98), ('Vaslui', 142)],

'Hirsova': [('Urziceni', 98), ('Eforie', 86)],

'Eforie': [('Hirsova', 86)],

'Vaslui': [('Urziceni', 142), ('Iasi', 92)],

'Iasi': [('Vaslui', 92), ('Neamt', 87)],

'Neamt': [('Iasi', 87)]

}

h1 = {

'Ar': 366,

'Bucharest': 0,

'Craiova': 160,

'Drobeta': 242,

'Eforie': 161,

'Fagaras': 176,

'Giurgiu': 77,

'Hirsova': 151,

'Iasi': 226,

'Lugoj': 244,

'Mehadia': 241,

'Neamt': 234,

'Oradea': 380,

'Pitesti': 100,

'Rimnicu Vilcea': 193,

'Sibiu': 253,

'Timisoara': 329,

'Urziceni': 80,

'Vaslui': 199,

'Zerind': 374

}

print(astar(graph, 'Ar', 'Bucharest', h1))

def a\_star(graph, start, goal, heuristic):

"""

A\* algorithm implementation.

:param graph: dictionary representing the graph

:param start: starting node

:param goal: goal node

:param heuristic: function that takes two nodes and returns estimated cost between them

:return: list of nodes representing the optimal path from start to goal

"""

start\_path = [start]

cost = 0

f = cost + heuristic(start, goal)

queue = [(f, cost, start\_path)]

visited = set()

while queue:

f, cost, path = heapq.heappop(queue)

current\_node = path[-1]

if current\_node == goal:

return path

if current\_node not in visited:

visited.add(current\_node)

for neighbor, neighbor\_cost in graph[current\_node]:

if neighbor not in path:

new\_path = path + [neighbor]

new\_cost = cost + neighbor\_cost

f = new\_cost + heuristic(neighbor, goal)

heapq.heappush(queue, (f, new\_cost, new\_path))

return None

def straight\_line\_distance(node, goal):

h = h1[node] - h1[goal]

return h

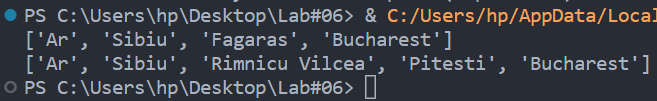
start\_node = 'Ar'

goal\_node = 'Bucharest'

path = a\_star(graph, start\_node, goal\_node, straight\_line\_distance)

print(path)

Output:



**Exercise#02**

import random

def drawBoard(board):

# This function prints out the board that is passed to it.

# "board" is a list of 10 strings representing the board (ignore index 0)

print()

print(' | |')

print(' '+board[7]+' | ' + board[8]+' | '+board[9])

print(' | |')

print('-----------')

print(' '+board[4]+' | ' + board[5]+' | '+board[6])

print(' | |')

print('-----------')

print(' | |')

print(' '+board[1]+' | ' + board[2]+' | '+board[3])

print(' | |')

def inputPlayerLetter():

# Lets the player type which letter they want to be their mark

# Returns a list with the player's letter as the first item, and the computer's letter as the second.

# For simplification, keeping X as the player's letter and O as the computer's letter

return ['X', 'O']

def whoGoesFirst():

# for simplification letting the computer go first

return 'computer'

def playAgain():

# This function returns True if the player wants to play again, otherwise it returns False.

print('Do you want to play again? (yes or no)')

return input().lower().startswith('y')

def makeMove(board, letter, move):

# This function simply marks the planned move (Location of the board with the player's letter.

board[move] = letter

def isWinner(bo, le):

# Given a board and a player's letter, this function returns True if that player has won.

# We use bo instead of board and le instead of letter so we don't have to type as much.

return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the top

(bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle

(bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom

(bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side

(bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle

# down the right side

(bo[9] == le and bo[6] == le and bo[3] == le) or

(bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal

(bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal

def getBoardCopy(board):

# Make a duplicate of the board list and return it the duplicate

dupeBoard = []

for i in board:

dupeBoard.append(i)

return dupeBoard

def isSpaceFree(board, move):

# Return true if the passed move is free on the passed board.

return board[move] == ''

def getPlayerMove(board):

# Let the player type in his move

move = ''

while move not in '1 2 3 4 5 6 7 8 9'.split() or not isSpaceFree(board, int(move)):

print('What is your next move? (1-9)')

move = input()

return int(move)

def chooseRandomMoveFromList(board, movesList):

# Returns a valid move from the passed list on the passed board.

# Returns None if there is no valid move.

possibleMoves = []

for i in movesList:

if isSpaceFree(board, i):

possibleMoves.append(i)

if len(possibleMoves) != 0:

return random.choice(possibleMoves)

else:

return None

def getComputerMove(board, computerLetter):

# Given a board and the computer's letter, determine where to move and return that move.

# define the player's letter

if computerLetter == 'X':

playerLetter = 'O'

else:

playerLetter = 'X'

# define the maximize and minimize functions

def maximize(board):

# if the game is over, return the score

if isWinner(board, computerLetter):

return 10

elif isWinner(board, playerLetter):

return -10

elif board.count('') == 0:

return 0

# if the game is not over, evaluate all possible moves and return the maximum score

else:

maxEval = -float('inf')

for i in range(1, 10):

if isSpaceFree(board, i):

copy = getBoardCopy(board)

makeMove(copy, computerLetter, i)

eval = minimize(copy)

maxEval = max(maxEval, eval)

return maxEval

def minimize(board):

# if the game is over, return the score

if isWinner(board, computerLetter):

return 10

elif isWinner(board, playerLetter):

return -10

elif board.count('') == 0:

return 0

# if the game is not over, evaluate all possible moves and return the minimum score

else:

minEval = float('inf')

for i in range(1, 10):

if isSpaceFree(board, i):

copy = getBoardCopy(board)

makeMove(copy, playerLetter, i)

eval = maximize(copy)

minEval = min(minEval, eval)

return minEval

# call the maximize function on each possible move and return the move with the highest score

bestMove = None

maxEval = -float('inf')

for i in range(1, 10):

if isSpaceFree(board, i):

copy = getBoardCopy(board)

makeMove(copy, computerLetter, i)

eval = minimize(copy)

if eval > maxEval:

maxEval = eval

bestMove = i

return bestMove

def isBoardFull(board):

# Return True if every space on the board has been taken. Otherwise returns False.

for i in range(1, 10):

if isSpaceFree(board, i):

return False

return True

def getPossibleMoves(board):

# return a list of all possible moves

moves = []

for i in range(1, len(board)):

if board[i] == '':

moves.append(i)

return moves

def getState(board, computerLetter, playerLetter):

# get the current state

state = ''

for i in range(1, 10):

if board[i] == computerLetter:

state += '1'

elif board[i] == playerLetter:

state += '2'

else:

state += '0'

return state

def chooseMove(qTable, state):

# randomly select a move from the list of possible moves

if state in qTable:

possibleMoves = qTable[state]

move = random.choice(possibleMoves)

else:

move = random.randint(1, 9)

return move

def computerVsHuman():

# function for computer vs human simulation using a lookup table approach

board = ['']\*10

computerLetter, playerLetter = 'X', 'O'

turn = whoGoesFirst()

print('The ' + turn + ' will go first.')

# initialize the lookup table

qTable = {}

# draw all the moves

while True:

if turn == 'computer':

# get the current state

state = getState(board, computerLetter, playerLetter)

# choose a move using the lookup table

move = chooseMove(qTable, state)

# make the move

makeMove(board, computerLetter, move)

# print the move made by computer

print('Computer has made a move. Board is:')

drawBoard(board)

# check for a win

if isWinner(board, computerLetter):

drawBoard(board)

print('Computer has won the game!')

break

else:

if isBoardFull(board):

drawBoard(board)

print('The game is a tie!')

break

else:

turn = 'player'

else:

# get the player's move

move = getPlayerMove(board)

# make the move

makeMove(board, playerLetter, move)

# print the move made by the player

print('Player has made a move. Board is:')

drawBoard(board)

# check for a win

if isWinner(board, playerLetter):

drawBoard(board)

print('Player has won the game!')

break

else:

if isBoardFull(board):

drawBoard(board)

print('The game is a tie!')

break

else:

turn = 'computer'

computerVsHuman()

Output:

