**CHATBOT**

from chatterbot import ChatBot

from chatterbot.trainers import ListTrainer

# Create a new chatbot instance

chatbot = ChatBot('PizzaBot')

# Create a trainer to train the chatbot

trainer = ListTrainer(chatbot)

# Define a custom corpus of conversation pairs

custom\_corpus = [

("What pizzas do you offer?", "We offer a variety of pizzas, including..."),

("How much is a Margherita pizza?", "A Margherita pizza costs $10."),

("Do you have gluten-free options?", "Yes, we have gluten-free pizza options."),

("What are your delivery hours?", "We deliver from 11:00 AM to 10:00 PM."),

("Can I order extra cheese on my pizza?", "Yes, you can customize your pizza with extra cheese."),

("What's the phone number to place an order?", "You can call us at (123) 456-7890."),

("How long does the delivery usually take?", "Delivery times may vary, depending on..."),

("Do you offer any discounts or promotions?", "Yes, we have special deals on certain days.")

]

# Train the chatbot with the custom corpus

trainer.train(custom\_corpus)

# Start the conversation

print("PizzaBot: Hello! How can I assist you with your pizza order today? Type 'bye' to exit.")

while True:

user\_input = input("You: ")

if user\_input.lower() == 'bye':

print("PizzaBot: Goodbye! Enjoy your pizza.")

break

response = chatbot.get\_response(user\_input)

print("PizzaBot:", response)

**BFS**

"""All possible path problem for romanian cities"""

GRAPH = {\

            'Arad': {'Sibiu': 140, 'Zerind': 75, 'Timisoara': 118},\

            'Zerind': {'Arad': 75, 'Oradea': 71},\

            'Oradea': {'Zerind': 71, 'Sibiu': 151},\

            'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu': 80},\

            'Timisoara': {'Arad': 118, 'Lugoj': 111},\

            'Lugoj': {'Timisoara': 111, 'Mehadia': 70},\

            'Mehadia': {'Lugoj': 70, 'Drobeta': 75},\

            'Drobeta': {'Mehadia': 75, 'Craiova': 120},\

            'Craiova': {'Drobeta': 120, 'Rimnicu': 146, 'Pitesti': 138},\

            'Rimnicu': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},\

            'Fagaras': {'Sibiu': 99, 'Bucharest': 211},\

            'Pitesti': {'Rimnicu': 97, 'Craiova': 138, 'Bucharest': 101},\

            'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},\

            'Giurgiu': {'Bucharest': 90},\

            'Urziceni': {'Bucharest': 85, 'Vaslui': 142, 'Hirsova': 98},\

            'Hirsova': {'Urziceni': 98, 'Eforie': 86},\

            'Eforie': {'Hirsova': 86},\

            'Vaslui': {'Iasi': 92, 'Urziceni': 142},\

            'Iasi': {'Vaslui': 92, 'Neamt': 87},\

            'Neamt': {'Iasi': 87}\

        }

def bestfirst(source, destination):

    """Optimal path from source to destination using straight line distance heuristic

    :param source: Source city name

    :param destination: Destination city name

    :returns: Heuristic value, cost and path for optimal traversal

    """

    # HERE THE STRAIGHT LINE DISTANCE VALUES ARE IN REFERENCE TO BUCHAREST AS THE DESTINATION

    straight\_line = {\

                        'Arad': 366,\

                        'Zerind': 374,\

                        'Oradea': 380,\

                        'Sibiu': 253,\

                        'Timisoara': 329,\

                        'Lugoj': 244,\

                        'Mehadia': 241,\

                        'Drobeta': 242,\

                        'Craiova': 160,\

                        'Rimnicu': 193,\

                        'Fagaras': 176,\

                        'Pitesti': 100,\

                        'Bucharest': 0,\

                        'Giurgiu': 77,\

                        'Urziceni': 80,\

                        'Hirsova': 151,\

                        'Eforie': 161,\

                        'Vaslui': 199,\

                        'Iasi': 226,\

                        'Neamt': 234\

                    }

    from queue import PriorityQueue

    priority\_queue, visited = PriorityQueue(), {}

    priority\_queue.put((straight\_line[source], 0, source, [source]))

    visited[source] = straight\_line[source]

    while not priority\_queue.empty():

        (heuristic, cost, vertex, path) = priority\_queue.get()

        if vertex == destination:

            return heuristic, cost, path

        for next\_node in GRAPH[vertex].keys():

            current\_cost = cost + GRAPH[vertex][next\_node]

            heuristic =  straight\_line[next\_node]

            if not next\_node in visited or visited[next\_node] >= heuristic:

                visited[next\_node] = heuristic

                priority\_queue.put((heuristic, current\_cost,next\_node, path + [next\_node]))

def main():

    """Main function"""

    print('ENTER SOURCE :', end=' ')

    source = input().strip()

    print('ENTER GOAL :', end=' ')

    goal = input().strip()

    if source not in GRAPH or goal not in GRAPH:

        print('ERROR: CITY DOES NOT EXIST.')

    else:

               print('\nBFS PATH:')

               heuristic, cost, optimal\_path = bestfirst(source, goal)

               print('PATH COST =', cost)

               print(' -> '.join(city for city in optimal\_path))

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**A\***

class Node:

def \_\_init\_\_(self, data, level, fval):

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

x, y = self.find(self.data, '\_')

val\_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]

children = []

for i in val\_list:

child = self.shuffle(self.data, x, y, i[0], i[1])

if child is not None:

child\_node = Node(child, self.level+1, 0)

children.append(child\_node)

return children

def shuffle(self, puz, x1, y1, x2, y2):

if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

temp\_puz = []

temp\_puz = self.copy(puz)

temp = temp\_puz[x2][y2]

temp\_puz[x2][y2] = temp\_puz[x1][y1]

temp\_puz[x1][y1] = temp

return temp\_puz

else:

return None

def copy(self, root):

temp = []

for i in root:

t = []

for j in i:

t.append(j)

temp.append(t)

return temp

def find(self, puz, x):

for i in range(0, len(self.data)):

for j in range(0, len(self.data)):

if puz[i][j] == x:

return i, j

class Puzzle:

def \_\_init\_\_(self, size):

self.n = size

self.open = []

self.closed = []

def accept(self):

puz = []

for i in range(0, self.n):

temp = input().split(" ")

puz.append(temp)

return puz

def f(self, start, goal):

return self.h(start.data, goal) + start.level

def h(self, start, goal):

temp = 0

for i in range(0, self.n):

for j in range(0, self.n):

if start[i][j] != goal[i][j] and start[i][j] != '\_':

temp += 1

return temp

def process(self):

print("Enter the start state matrix \n")

start = self.accept()

print("Enter the goal state matrix \n")

goal = self.accept()

start = Node(start, 0, 0)

start.fval = self.f(start, goal)

self.open.append(start)

print("\n\n")

while True:

cur = self.open[0]

print("")

print(" | ")

print(" | ")

print(" \\\'/ \n")

for i in cur.data:

for j in i:

print(j, end=" ")

print("")

if self.h(cur.data, goal) == 0:

break

for i in cur.generate\_child():

i.fval = self.f(i, goal)

self.open.append(i)

self.closed.append(cur)

del self.open[0]

self.open.sort(key=lambda x: x.fval, reverse=False)

puz = Puzzle(3)

puz.process()

**tic-tat-toe**

import random

board = [' ' for \_ in range(9)]

HUMAN = 'X'

COMPUTER = 'O'

WINNING\_COMBINATIONS = [(0, 1, 2), (3, 4, 5), (6, 7, 8),

(0, 3, 6), (1, 4, 7), (2, 5, 8),

(0, 4, 8), (2, 4, 6)]

def print\_board(board):

for row in [board[i:i + 3] for i in range(0, 9, 3)]:

print(' | '.join(row))

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

def check\_win(board, player):

for combo in WINNING\_COMBINATIONS:

if all(board[i] == player for i in combo):

return True

return False

def is\_board\_full(board):

return ' ' not in board

def get\_empty\_squares(board):

return [i for i, val in enumerate(board) if val == ' ']

def minimax(board, depth, maximizing):

if check\_win(board, COMPUTER):

return 1

elif check\_win(board, HUMAN):

return -1

elif is\_board\_full(board):

return 0

if maximizing:

best\_score = -float('inf')

for move in get\_empty\_squares(board):

board[move] = COMPUTER

score = minimax(board, depth + 1, False)

board[move] = ' '

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = float('inf')

for move in get\_empty\_squares(board):

board[move] = HUMAN

score = minimax(board, depth + 1, True)

board[move] = ' '

best\_score = min(score, best\_score)

return best\_score

def make\_computer\_move(board):

best\_score = -float('inf')

best\_move = None

for move in get\_empty\_squares(board):

board[move] = COMPUTER

score = minimax(board, 0, False)

board[move] = ' '

if score > best\_score:

best\_score = score

best\_move = move

board[best\_move] = COMPUTER

def main():

print\_board(board)

while True:

if not is\_board\_full(board):

try:

human\_move = int(input("Enter your move (1-9): ")) - 1

if 0 <= human\_move < 9 and board[human\_move] == ' ':

board[human\_move] = HUMAN

else:

print("Invalid move. Try again.")

continue

except ValueError:

print("Invalid input. Please enter a number between 1 and 9")

continue

else:

print("It's a draw!")

break

if check\_win(board, HUMAN):

print\_board(board)

print("You win!")

break

if not is\_board\_full(board):

make\_computer\_move(board)

print\_board(board)

else:

print("It's a draw!")

break

if check\_win(board, COMPUTER):

print("Computer wins!")

break

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

main()