AI ASSIGNMENT 1

21K-3153

Q1:

Turing argued that machines could not have souls. There are still a number of ethical and philosophical discussions taking place on this topic, but it is a fact that with the rise of LLMs and chatbots trained on those LLMs, AI is very close to being able to mimic human speech, blurring the line between machine and human, making itself near indistinguishable from those who have souls.

Since he wrote his paper, huge and drastic improvements have been made in the field of AI, with the rise of LLMs, generative AI and much more.

Turing’s prediction for the year 2000 did not prove to be true. Nowadays, even with very impressive chatbots and conversational AI, there is still a sense of artificiality when conversing with an AI. Achieving human level understanding is still a way off.

Q2:

**Ping Pong:**

Digitally, computers can play a theoretical perfect match of ping pong that can last forever due to advancements in technology, AI, physics mechanics, and collision mechanics. The computers can predict the ball’s trajectory and respond appropriately.

In the physical world, robots can play against humans efficiently but not perfectly. While computers excel in trajectory prediction and decision making, they lack in real time response and find it hard to match the response time of humans. However, with future advancements, this can be remedied as well.

**Writing an intentionally funny story:**

With the rise of Large Language Models and popular AI tools like ChatGPT, NovelAI, Gemini by Google and many more, computers have learnt to perfectly mimic multiple writing styles, including comedy, drama, and many other genres. Yes, computers can efficiently write an intentionally funny story.

**Translating spoken English into spoken Urdu in real time:**

Computers are very capable of translating one language to another in real time in real time with the help of widespread free translators like Google Translate and DeepL. However, translating English to spoken Urdu is more of a resource issue than a computer issue as the world’s translation models do not have enough data in Urdu to provide an accurate translation.

**Driving in the center of Karachi:**

Recently, self-driving autonomous cars are becoming more and more prevalent due to their intelligence and excellent results in testing, as evident by Tesla’s cars and Google’s Waymo taxis. However, those vehicles are tested in areas with proper roads and a population that follows driving laws. In Karachi, the vehicle’s object detection and accident-avoidance protocols would be severely pressured and might not yield satisfactory results.

**Giving competent legal advice in a specialized area of law:**

Nowadays, LLMs like ChatGPT and Gemini are trained on billions of parameters containing knowledge about many subjects, including law. However, when faced with a question LLMs do not have the answer to, they are known to hallucinate and make up factually incorrect information. Thus, it is okay to ask AI for law advice, but it is best to get a second opinion from a real lawyer and not blindly trust the computer.

The difficulties are mentioned in the answers themselves.

Q3:

Environment: A Tic Tac Toe game

The environment would be **fully observable, deterministic, stochastic, static, discrete, multiagent.**

The agent would be a **goal-based agent,** determined to win the game or draw if winning is impossible.

Q4:

**Playing soccer:**

**P:** Scoring a goal/ winning a game

**E:** A football field, smooth, artificial grass

**A:** Athlete’s feet to kick the ball

**S:** Athlete’s eyes to survey the field, survey players, find the ball

**Shopping for used AI books on the internet:**

**P:** Number of AI books found and bought should be satisfactory to the user

**E:** Websites that sell used books, online marketplaces

**A:** Screen, search queries, filtering options

**S:** Keyboard, search queries, user submitted reviews

**Exploring the subsurface of Arabian Sea.:**

**P:** Complete exploration of the sea

**E:** The Arabian Sea

**A:** Rotors to move the submarine through the sea, camera to observe the environment, sonars to send out sound waves across the environment

**S:** Submarine controls, machinery to operate the multiple sensors and rotors.

**Knitting a sweater:**

**P:** A completely knitted sweater with no deformities

**E:** Wool, yarn and a space or table to knit

**A:** The knitting needles which pierces and shapes the wool to produce a sweater

**S:** Hands to control the knitting needles, eyes to judge how much to knit, fingers to feel the texture of the sweater

**Performing a high jump:**

**P:** Performing a jump which is higher than the average jump perfectly

**E:** A spacious space to jump with an open sky, high ceilings and soft carpets

**A:** Athlete’s body and feet which help him jump up. His feet which push him off the ground

**S:** Athlete's muscles, eyesight and judgement to gauge when to how, how to jump and how much pressure to put on their calves, feet and other relevant muscles.

Q5:

**An agent that senses only partial information about the state cannot be perfectly rational: True.**

Perfect rationality means always choosing the option which yields the maximum performance. Without being fully aware of the environment the agent is in, there is a chance that the agent will not be aware of all its options and thus not choose the most optimized one.

**There exist task environments in which no pure reflex agent can behave rationally: True.**

Reflex agents can only make decisions based on their current precept and cannot take past precepts into account. This is why rationality is impossible as considering past precepts could lead to more optimized decisions, which a reflex agent is incapable of doing. Thus, they cannot be rational.

**There exists a task environment in which every agent is rational: False.**

Even in extremely simple task environments, such as a vacuum cleaner, an agent can still get stuck in the corner. In an even simpler environment, such as “walking straight”, an agent could slip, or not use the correct parts of their body.

Even if one agent is rational in a specific task environment, that does not mean that **every** agent will be.

**Suppose an agent selects its action uniformly at random from the set of possible actions. There exists a deterministic task environment in which this agent is rational: True.**

A deterministic task environment means an environment where the next state is determined by the current state and the action taken during that state.

Suppose an optimal decision is unknown for a deterministic environment. Through randomly selected actions, the agent would eventually find the action which is the most optimal. Their wrong guesses would not impact the environment as it is deterministic. Thus, the agent would be rational, and every random choice made would be optimal.

**It is possible for a given agent to be perfectly rational in two distinct task environments: True.**

An advanced agent placed into two simple distinct task environments would be rational in both.

An agent that would be rational at Tic Tac Toe could play one match of Tic Tac Toe, choose the most optimal decisions, go to play another match (a different environment) and choose the most optimal decisions again.

Q6:

Breath First Search:

graph = {

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

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

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

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

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

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

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

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

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

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

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

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

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

    'Urziceni' : {'Bucharest':85, 'Hirsova':98},

    'Giurgiu' : {'Bucharest':90},

    'Neamt' : {'Iasi':87},

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

    'Vaslui' : {'Iasi':92, 'Urziceni':98},

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

    'Eforie' : {'Hirsova':86}

}

visited = [] # List for visited nodes.

queue = []     # Initialize a queue

def bfs(visited, graph, node,dest): # Function for BFS

    if(node==dest):

        print("Source and destination are same.\n")

        return 0

    visited.append(node)

    queue.append(node)

    TotalCost = 0  # Initialize cost here

    cost = []

    while queue:          # Creating loop to visit each node

        m = queue.pop(0)

        print(m, end="    ")

        if m == dest:

            TotalCost += cost[0]

            return TotalCost

        if m != node:

            TotalCost += cost[0]

            cost = cost[1:]

        for neighbour, edge\_cost in graph[m].items():

            if neighbour not in visited:

                visited.append(neighbour)

                cost.append(edge\_cost)  # Update cost with edge cost

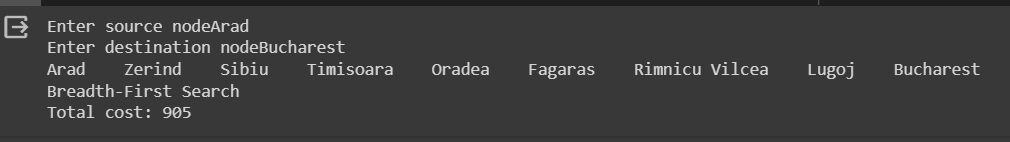
                queue.append(neighbour)

    return TotalCost

src = input("Enter source node")

dest= input("Enter destination node")

print(f"\nBreadth-First Search\nTotal cost: {bfs(visited, graph, src,dest)}")



Uniform Cost Search:

def uniform\_cost\_search(graph, start, goal):

    # Priority queue to store nodes to be explored

    frontier = [(0, start, [])]  # (cost, node, path)

    # Set to keep track of explored nodes

    explored = set()

    # Dictionary to store the cost to reach a node

    cost\_so\_far = {start: 0}

    while frontier:

        # Pop the node with the lowest cost from the priority queue

        current\_cost, current\_node, current\_path = heapq.heappop(frontier)

        print(current\_node)

        # Check if the current node is the goal

        if current\_node == goal:

            return current\_path + [current\_node], current\_cost

        # Mark the current node as explored

        explored.add(current\_node)

        # Explore neighbors of the current node

        for neighbor, cost in graph[current\_node].items():

            new\_cost = current\_cost + cost

            if neighbor not in explored and (neighbor not in cost\_so\_far or new\_cost < cost\_so\_far[neighbor]):

                cost\_so\_far[neighbor] = new\_cost

                heapq.heappush(frontier, (new\_cost, neighbor, current\_path + [current\_node]))

    # If goal is not reachable

    return None, float('inf')

# Example usage:

start\_node = input("Enter source node")

goal\_node = input("Enter dest node")

route, cost = uniform\_cost\_search(graph, start\_node, goal\_node)

if route:

    print("Route:", ' -> '.join(route))

    print("Cost:", cost)

else:

    print("Goal is not reachable from the start node.")

A screenshot of a computer program

Description automatically generated

Greedy Best First Search

heuristics = {

    "Arad" : 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

}

def greedy\_best\_first\_search(graph, heuristics, start, goal):

    visited = set()

    path = []

    queue = [(heuristics[start], 0, start)]  # (heuristic cost, actual cost, node)

    while queue:

        queue.sort(reverse=True)

        h\_cost, g\_cost, node = queue.pop()

        if node not in visited:

            visited.add(node)

            path.append(node)

            if node == goal:

                return path, g\_cost

            for neighbor, cost in graph[node].items():

                if neighbor not in visited:

                    queue.append((heuristics[neighbor], g\_cost + cost, neighbor))

    return None, None

# Example usage:

start\_node = input("Enter source node")

goal\_node = input("Enter dest node")

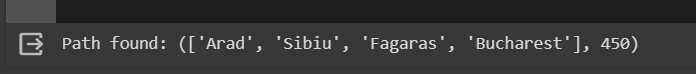
path = greedy\_best\_first\_search(graph, heuristics, start\_node, goal\_node)

if path:

    print("Path found:", path)

else:

    print("Path not found.")



Iterative Deepening Depth First Search

def dfs(graph, start, goal, depth\_limit):

    visited = set()

    stack = [(start, 0, [])]

    while stack:

        node, cost, path = stack.pop()

        print(f"Depth: {len(path)}, Exploring: {node}, Path: {path + [node]}, Cost: {cost}")

        if node == goal:

            return path + [node], cost  # If the goal is reached, return the path and total cost

        if len(path) < depth\_limit:  # Check if depth limit is not exceeded

            visited.add(node)

            for neighbor, edge\_cost in graph[node].items():

                if neighbor not in visited:

                    stack.append((neighbor, cost + edge\_cost, path + [node]))  # Add neighbors to stack with updated cost and path

    return None, None  # Return None if no path is found within the depth limit

def iddfs(graph, start, goal, max\_depth):

    for depth\_limit in range(max\_depth + 1):  # Iterate over increasing depth limits

        print(f"\nDepth Limit: {depth\_limit}")

        result, total\_cost = dfs(graph, start, goal, depth\_limit)  # Apply DFS with current depth limit

        if result is not None:

            return result, total\_cost  # If a path is found, return it

    return None, None  # If no path is found within any depth limit, return None

start\_node = input("Enter source node")

goal\_node = input("Enter dest node")

max\_depth = 10

path, total\_cost = iddfs(graph, start\_node, goal\_node, max\_depth)

if path:

    print("\nPath found:", ' -> '.join(path))

    #print("Total Cost:", total\_cost)

else:

    print("\nPath not found within depth limit.")

A screenshot of a computer

Description automatically generated

BFS can find the optimal solution as it goes level by level, however the cost of reaching the destination is increased compared to other algorithms. It also has a larger space complexity compared to the other algorithms.

IDDFS, Greedy Best First Search and Uniform Search all gave the same path with the same cost.

Out of these three, greedy best first search does not ensure optimal results as it is dependent on the heuristic function used. IDDFS and Uniform Search do guarantee optimal results. However, Greedy can be considered the most time efficient as it adopts a greedy strategy.

IDDFS combines both benefits of DFS and BFS, thus has a lower space complexity than the others.

N Queen:

global N

def printSolution(board):

  for i in range(N):

    for j in range(N):

      print (board[i][j],end=' ')

    print()

def isSafe(board, row, col):

  # Check this row on left side

  for i in range(col):

    if board[row][i] == 1:

      return False

  # Check upper diagonal on left side

  for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

    if board[i][j] == 1:

      return False

  # Check lower diagonal on left side

  for i, j in zip(range(row, N, 1), range(col, -1, -1)):

    if board[i][j] == 1:

      return False

  return True

def solveNQUtil(board, col):

  # base case: If all queens are placed

  # then return true

  if col >= N:

    return True

  # Consider this column and try placing

  # this queen in all rows one by one

  for i in range(N):

    if isSafe(board, i, col):

      # Place this queen in board[i][col]

      board[i][col] = 1

      # recur to place rest of the queens

      if solveNQUtil(board, col + 1) == True:

        return True

      # If placing queen in board[i][col

      # doesn't lead to a solution, then

      # queen from board[i][col]

      board[i][col] = 0

  # if the queen can not be placed in any row in

  # this column col then return false

  return False

def solveNQ(n):

  board = [[0] \* N for \_ in range(N)]

  solveNQUtil(board,0)

  printSolution(board)

# driver program to test above function

N = int(input("Enter a number between 4 and 8"))

solveNQ(N)