National University of Computer & Emerging Sciences, Karachi

Spring -2023
AI-2002 Artificial Intelligence
Assignment#1

Roll No#20K-0190 **Name: Mohamamd Usama**

Question 01:

In his 1950 paper on AI, "Computing Machinery and Intelligence," Turing addressed potential objections to his proposed business plan and intelligence test. Let's examine the arguments that still hold weight and the accuracy of his refutations.

Theological Challenge: Theological challenges suggest that robots are unable to possess intelligence as they lack souls. Turing responds that it is more a matter of opinion than fact. However, some people with strong religious beliefs may find this argument persuasive and still oppose the concept of artificial intelligence.

Mathematical Objection: The mathematical objection suggests that computers have limitations on the amount of computation they can perform, making it impossible for them to possess intelligence. In response, Turing argues that intellect is not always linked to mathematical prowess. Nonetheless, some researchers contend that computers have inherent limitations, such as the halting problem, which may hinder their ability to act intelligently in certain situations.

Consciousness Argument: The consciousness argument contends that machines are unable to possess intelligence because they lack subjective experience. Turing responds by stating that it is impossible to prove or disprove the existence of computer consciousness. However, this argument still holds weight as true intelligence may depend on consciousness, which is a complex and poorly understood phenomenon.

Question 02:

AI systems have made some progress in this area, as demonstrated by the Forpheus robot created by Omron Corporation, which can play a decent game of table tennis. However, current AI systems are still unable to compete at a high level due to difficulties in real-time decision-making and fine motor control.

Although self-driving cars have advanced significantly in recent years, driving in chaotic and congested areas such as Karachi remains a significant challenge for AI systems. Real-time decision-making in such situations is difficult, and AI systems may struggle to handle new or unexpected circumstances.

Writing an intentionally funny story is challenging because humor is a complex and individualized phenomenon that is difficult to computationally model. While AI systems can produce grammatically correct and coherent text, the challenge of writing intentionally humorous stories remains.

Providing competent legal advice in a specialized area of law still requires human expertise and judgement, despite AI systems being developed to assist lawyers with various tasks such as document review and legal research. AI systems still struggle with legal reasoning and interpreting complex legal precedents and codes.

Real-time translation of spoken English into spoken Urdu is still difficult despite significant advancements in machine translation. Managing idiomatic expressions, nuanced meanings, and cultural contexts can be challenging for AI systems, particularly in languages with significant variations in syntax or grammar.

Question 03:

The environment in which self-driving cars operate is constantly changing, accessible, and only partly observable. To navigate safely and effectively, the agent must receive inputs from multiple sensors, such as LIDAR, RADAR, cameras, GPS, and other onboard systems, and make quick decisions while following traffic regulations and avoiding hazards.

A hybrid agent architecture that combines both reactive and deliberative methods is ideal for this domain. The deliberative component of the architecture helps the agent plan high-level decisions, such as route planning, while the reactive component responds promptly to immediate situations, such as avoiding obstacles, based on long-term objectives.

The agent must have the ability to learn from experience and adapt to changing environmental conditions. Reinforcement learning techniques can be used to train the agent to respond to different situations. For example, the agent could be rewarded for making safe decisions while driving and punished for breaking traffic rules or causing accidents.

In conclusion, the self-driving vehicle environment requires an agent that can quickly react to complex and dynamic situations while also planning for long-term goals. The optimal architecture for this domain combines reactive and deliberative methods with reinforcement learning strategies.

Question 04:

1. **Playing soccer:**

- Performance measure: Score goals and prevent the opposing team from scoring.
- Environment: Dynamic, multi-agent, accessible, and continuous.
- Actuators: Legs, feet, torso, and arms for running, kicking, and throwing.
- Sensors: Eyes, ears, skin for sensing the location of the ball, other players, and the field.

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

- Performance measure: Collect data on the subsurface features of the Arabian Sea.
- Environment: Partially observable, continuous, hazardous, and inaccessible.
- Actuators: Submarine, robotic arms, and sampling equipment.
- Sensors: Sonar, cameras, temperature sensors, pressure sensors.

3. **Shopping for used AI books on the Internet:**

- Performance measure: Find and purchase used AI books.
- Environment: Accessible, static, and discrete.
- Actuators: Mouse and keyboard for browsing, selecting, and purchasing.
- Sensors: Screen for viewing options and prices.

4. **Playing a tennis match:**

- Performance measure: Win the match by scoring more points than the opponent.
- Environment: Dynamic, multi-agent, accessible, and continuous.
- Actuators: Arms and legs for running, swinging the racket, and hitting the ball.
- Sensors: Eyes, ears, and skin for sensing the location of the ball, the opponent, and the court.

5. **Practicing tennis against a wall:**

- Performance measure: Improve tennis skills.
- Environment: Accessible, static, and continuous.
- Actuators: Arms and legs for hitting the ball against the wall.
- Sensors: Eyes and skin for sensing the location of the ball and the wall

6. **Performing a high jump:**

- Performance measure: Jump the highest possible.
- Environment: Static, continuous, and accessible.
- Actuators: Legs for jumping.
- Sensors: Eyes and skin for sensing the height of the bar and the landing area.

7. **Knitting a sweater:**

- Performance measure: Create a knitted sweater.

- Environment: Static, discrete, and accessible.
- Actuators: Hands and needles for knitting.
- Sensors: Eyes and skin for sensing the pattern and the texture of the yarn.

8. **Bidding on an item at an auction:**

- Performance measure: Win the auction and purchase the item at the lowest possible price.
- Environment: Multi-agent, dynamic, and accessible.
- Actuators: Mouse and keyboard for bidding.
- Sensors: Screen for viewing the item and tracking the bids.

Question 05:

False. If the agent chooses the best course of action based on the information at hand, even when it only has partial knowledge about the state, it can still be considered perfectly rational. For instance, even if a chess-playing agent can't see the complete board, it can still decide which move to make based on the pieces that are visible.

True. Pure reflex agents only take actions based on their current perception, so they are unable to act sensibly in situations where their current perception is insufficient to enable them to make an informed decision. For instance, a pure reflex agent cannot act rationally in a task environment where an agent must make a choice based on the history of prior perceptions.

False. The work environment affects rationality, and not all task environments promote rational behaviour. No agent can act rationally, for instance, in a task setting where results of actions are utterly random.

False. The present percept is the input to an agent programme, whereas the percept history is the input to an agent function. The agent function maps the percept history to an action, whereas the agent programme maps the present percept to an action.

False. A program/machine combination cannot perform every agent function. There may be agent tasks that cannot be performed by a program/machine combination because they require infinite memory or infinite processing time. Correct grammar mistake

Question 06:

```
from queue import PriorityQueue
from collections import deque

graph = {
   "Oradea" : {"Sibiu" : 151 , "Zerind" : 75 },
   "Zerind" : {"Arad" : 75 , "Oradea" : 71},
   "Arad" : {"Zerind" : 75 , "Timisoara" : 118 , "Sibiu" : 140},
   "Sibiu" : {"Fagarus" : 99 , "Arad" : 140 , "Oradea" : 151, "Rimnicu Vilcea" : 80},
   "Fagarus" : {"Sibiu" : 99 , "Bucharest" : 211},
   "Rimnicu Vilcea" : {"Sibiu" : 80 , "Pitesti" : 97 , "Craiova" : 146 },
   "Pitesti" : {"Rimnicu Vilcea" : 97, "Craiova" :138 , "Bucharest" : 101},
   "Bucharest" : {"Fagarus" : 211 , "Pitesti" : 101 , "Guirgui" : 90 , "Urziceni" : 85},
   "Guirgui" : {"Bucharest" : 90},
   "Urziceni" : {"Bucharest" : 85 , "Hirsova" : 98, "Vaslul" : 142},
   "Hirsova" : {"Eforie" : 86},
   "Eforie" : {"Hirsova" : 86},
   "Vaslul" : {"Urziceni" : 142 , "Iasi" : 92},
   "Iasi" : {"Vaslul" : 92, "Nearnt" : 87},
   "Nearnt" : {"Iasi" : 87},
   "Craiova" : { "Pitesti" : 138 , "Rimnicu Vilcea" : 146 , "Drobeta" : 120},
   "Drobeta" : {"Craiova" : 120 , "Mehadia" : 75},
   "Mehadia"  : {"Drobeta" : 75 , "Lugoj" : 70},
   "Lugoj" : {"Mehadia" : 70 , "Timisoara": 111},
   "Timisoara": {"Lugoj" : 111 , "Arad" : 118},
}

def Bfs(graph, a, b): #Breadth First Search
   a = graph.get(start)
   b = graph.get(last)
   visited = set()
   queue = [a]

   while queue:
      node = queue.pop()
      if node not in visited:
         visited.add(node)

         if node == b:
            return
         for neighbor in graph[node]:
            if neighbor not in visited:
               queue.appendleft(neighbor)
```

```
def Ucs(graph, a, b): #Uniform Cost Search
    a = graph.get(start)
    b = graph.get(last)
    visited = set()
    queue = PriorityQueue()
    queue.put((0, a))

    while queue:
        cost, node = queue.get()
        if node not in visited:
            visited.add(node)

            if node == b:
                return
            for i in graph.neighbors(node):
                if i not in visited:
                    t_cost = cost + graph.get_cost(node, i)
                    queue.put((t_cost, i))

print("SEARCHING")
print("1. Breadth First Search")
print("2. Uniform Cost Search")
print("3. Greedy Best First Search")
print("4. Iterative Deepening Depth First Search")
opt = input("Choose Searching Option From the Following: ")

if opt == '1':
    start = input("Enter Start: ")
    last = input("Enter Destination: ")
    a = graph.get(start)
    b = graph.get(last)
    {Bfs(graph, a, b)}
elif opt == '2':
    start = input("Enter Start: ")
    last = input("Enter Destination: ")
    a = graph.get(start)
    b = graph.get(last)
    {Ucs(graph, a, b)}
elif opt == '3':
    Gbfs()
elif opt == '4':
    Iddfs()
else:
    print("Invalid Option!")
```

Question 07:

```
import numpy as np
global n
n = int(input("Enter value of n b/w -> 4 & 8 :  "))

def is_safe(B, row, col): # function to check whether it is safe or not according to chess rule
    # Check if there is a queen in the same row
    for i in range(col):
        if B[row][i] == 1:
            return False

    # Check if there is a queen in upper diagonal on left side
    i = row
    j = col
    while i >= 0 and j >= 0:
        if B[i][j] == 1:
            return False
        i -= 1
        j -= 1

    # Check if there is a queen in lower diagonal on left side
    i = row
    j = col
    while j >= 0 and i < n:
        if B[i][j] == 1:
            return False
        i += 1
        j -= 1

    return True

# it backtracks if queen found not safe
def solve_n_queens_util(B, col):
    if col == n:
        return True

    for i in range(n):
        if is_safe(B, i, col):
            B[i][col] = 1
            if solve_n_queens_util(B, col+1):
                return True
            B[i][col] = 0
```

```
    # Backtrack
    return False

def solve_n_queens():
    # Creating 2D array and intializing it with 0
    B = [[0 for _ in range(n)] for _ in range(n)]

    # Calling of recursive function
    if solve_n_queens_util(B, 0) == False:
        print("No solution exists")
        return

    # Print the board
    for i in range(n):
        for j in range(n):
            if B[i][j] == 1:
                print("Q ", end="")
            else:
                print("- ", end="")
        print()

while True:
    if n < 4 or n > 8:
        continue
    else:
        solve_n_queens()
        break
```