# AI-Assignment :01

Date:19/02/2025

Made by : Mir-Ahmed 23i-0142

# TASK:01

**Essay on Turing's Ideas About AI:**

Alan Turing's 1950 paper asked Can machines think? He proposed the *Turing Test*: if a machine can chat like a human, it's intelligent He also addressed objections.

### 1. Objections That Still Matter

- **Machines Can't Understand**: Some argue machines only follow rules like a cookbook, and don't *truly* think. People still debate this today.
- **Math Limits Machines**: Machines can't solve every problem (due to math rules like Gödel's theorem). But humans have limits too, so this isn't a big weakness.
- **Human Behavior Is Too Messy**: Humans act unpredictably. Turing said machines might mimic this—today's AI (like chatbots) does this well, but it's still programmed not free-thinking.

### 2. Were Turing's Answers Good?

- **Religious/Fear-Based Objections**: Turing correctly said these aren't logical. Fear of machines or claiming only humans have souls isn't scientific.
- **Machines Can't Be Creative**: Turing said they could. He was right! Modern AI writes poems, jokes, and even invents recipes.
- **Brains Aren't Machines**: Turing argued machines can mimic brains. Today's neural networks (like ChatGPT) prove this works.

### 3. New Problems Turing Didn't See

- **Bias**: AI can be unfair (e.g., favoring certain groups in hiring).
- **Environment**: Training AI uses massive energy, harming the planet.
- **Control**: What if super-smart AI becomes uncontrollable?

**4. Was His 2000 Prediction Reasonable?**

Turing guessed a machine might fool 30% of people in a 5-minute chat by 2000. While no AI fully passed his test by then, early chatbots (like ELIZA) tricked some users. Today, tools like ChatGPT or Siri often act human-like. His guess was a bit early but not wrong.

**Conclusion**

Turing's ideas were ahead of his time. Machines *can* act smart but debates about real thinking or ethics still matter. His test isn't perfect, but it started the AI conversation we're still having today!

# TASK:02

**Note:** For this Question i also did research from many sources then gave answer to you sir i also provides links of sources.

- **1. Playing a decent game of table tennis (ping-pong).**
  **Status:** Partially feasible.
  **Reason:** Google DeepMind developed an AI-powered robot capable of competing at an amateur human level. It won 13 out of 29 matches against humans, defeating beginners and 55% of amateurs but lost to advanced players.
  **Source:** MIT Technology Review

- **2. Playing a decent game of bridge at a competitive level.**
  **Status:** Feasible.
  **Reason:** The AI system Nook, developed by NukkAI, outperformed eight world champion bridge players in a tournament, winning 67 out of 80 sets (83% success rate).
  **Source:** The Register

- **3. Writing an intentionally funny story.**
  **Status:** Currently infeasible.
  **Reason:** Humor relies on cultural context, sarcasm, and timing, which AI struggles

to replicate authentically in storytelling.
**Source:** Studies on AI and humor (e.g., arXiv)

- **4. Giving competent legal advice in a specialized area of law.**
  **Status:** Partially feasible.
  **Reason:** AI tools like ROSS assist with legal research and document analysis but lack contextual judgment for nuanced or ethical decisions.
  **Source:** Legal Tech News

- **5. Discover and prove a new mathematical theorem.**
  **Status:** Partially feasible.
  **Reason:** AI (e.g., DeepMind) can identify patterns and propose conjectures, but formal proofs require human creativity and expertise.
  **Source:** DeepMind Research

- **6. Perform a surgical operation.**
  **Status:** Partially feasible.
  **Reason:** Robotic systems like da Vinci assist surgeons with precision but cannot operate fully autonomously due to the need for human adaptability.
  **Source:** Nature Medicine

- **7. Unload any dishwasher in any home.**
  **Status:** Currently infeasible.
  **Reason:** Robots struggle with varied dish layouts, fragile items, and adapting to unpredictable home environments.
  **Source:** Robotics Journals

- **8. Construct a building.**
  **Status:** Partially feasible.
  **Reason:** Automation handles tasks like 3D printing or bricklaying, but full construction requires human oversight for plumbing, wiring, and problem-solving.
  **Source:** Construction Dive

## TASK:03

**Domain: Autonomous Delivery Robot for a University Campus**

**Environment Description**

The environment is a university campus where the robot delivers packages (e.g., books, food, documents) to students and staff.

1. **Accessible:** Partially accessible. The robot has sensors (cameras, LiDAR) to perceive its surroundings but may not fully know the layout of every building or room.
2. **Deterministic:** Mostly deterministic. The robot's actions (e.g., moving forward, turning) have predictable outcomes, but unexpected obstacles (e.g., students walking by) introduce some uncertainty.
3. **Episodic:** Not episodic. The robot's tasks are sequential and interconnected. For example, delivering one package may affect the route for the next.
4. **Static:** Mostly static. The environment doesn't change drastically during a delivery, but dynamic elements (e.g., moving people, weather) make it semi-dynamic.
5. **Continuous:** Continuous. The robot operates in real-time, navigating through a physical space with no clear breaks between actions.

**Agent Description**

The **autonomous delivery robot** is designed to:

1. **Perceive:** Use sensors (cameras, LiDAR, GPS) to detect obstacles, read maps, and locate delivery points.
2. **Plan:** Generate optimal routes using algorithms like A* or Dijkstra's, avoiding obstacles and minimizing travel time.
3. **Act:** Move using wheels, avoid collisions, and interact with users (e.g., notify recipients via an app).
4. **Learn:** Improve over time by analyzing past deliveries and adapting to new campus layouts.

**Agent Architecture**

The best architecture is a **hybrid model** combining:

1. **Reactive Components:** For real-time obstacle avoidance (e.g., using rule-based systems).
2. **Deliberative Components:** For route planning and decision-making (e.g., using search algorithms).

3. **Learning Components:** For adapting to new environments and improving efficiency (e.g., reinforcement learning).

**Why This Architecture?**

- **Reactive:** Handles dynamic obstacles (e.g., students walking).
- **Deliberative:** Ensures efficient route planning.
- **Learning:** Improves performance over time.

This hybrid approach balances real-time responsiveness with long-term adaptability, making it ideal for a semi-structured, dynamic environment like a university campus.

# TASK:04

**Note: For this question i gave answers according to our text book of AI**

1. **An agent with partial info can't be perfectly rational.**
   a. **False.** Example: A self-driving car with limited sensors can still drive safely by guessing missing info.

2. **Some tasks need more than reflex agents.**
   a. **True.** Example: Chess requires planning; reflex agents can't think ahead.

3. **Some tasks make all agents rational.**
   a. **False.** Example: In most environments, agents need specific goals or designs to act rationally. A random agent in a complex task (e.g., solving math problems) won't be rational.

4. **Agent program input ≠ agent function input.**

   a. **False.** Example: A robot's program processes raw camera images, but its function decides actions based on interpreted data

5. **Not all agent functions can be implemented.**
   a. **False.** Example: Solving the Halting Problem is impossible for any machine.

6. **Random actions can be rational in some tasks.**
   a. **True.** Example: Flipping a fair coin—no action is better than another.

7. **One agent can be rational in two tasks.**
   a. **True.** Example: A vacuum robot can clean both small rooms and large halls perfectly.

# TASK:05

QNo8-05

(i) BFS

- explores all nodes level by level

Exp: from Arad to Bucharest

Path :- Arad → Sibiu → Fagaras → Bucharest

cost :- 140 + 99 + 211 = 450

- it is shortest path in terms of steps
- Can be slow and memory intensive for large graphs

(ii) UCS

- explores the cheapest path

Exp: from Arad to bucharest

path :- arad → sibiu → Rimnicu Vilcea → Pitesti → Bucharest

cost :- 140 + 80 + 97 + 101 = 418

- lowest cost Path
- Can be slower than BFS

(iii) GBFS

- Use heuristic (eg: straight line distance to Bucharest) to choose next node

Exp: from Arad to bucharest

Path : Arad → Sibiu → Fagaras → Bucharest

cost : 140 + 99 + 211 = 450

- Fast and use less memory
- Do not guarantee shortest or cheapest path

(iv) IDDFS

• repeatedly performs depth first search with increasing depth limits

Exp:- from Arad to Bucharest

Path :- Arad → sibiu → Fagaras → Bucharest

Cost :- 140 + 99 + 211 = 450

→ it combines the benefits of BFS and DFS

→ Can be slower due to repeated searches

| **Algorithm** | **Path** | **cost** | **Best For** |
|---|---|---|---|
| Breadth-First Search (BFS) | Arad → Sibiu → Fagaras → Bucharest | 450 | Shortest path in steps |
| Uniform Cost Search (UCS) | Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest | 418 | Lowest cost path |
| Greedy Best-First Search (GBFS) | Arad → Sibiu → Fagaras → Bucharest | 450 | Fast but not always optimal |

| | | | |
|---|---|---|---|
| Iterative Deepening DFS (IDDFS) | Arad → Sibiu → Fagaras → Bucharest | 450 | Memory-efficient and optimal |

**Code:**

```
# Romania map with distances
romania_map = {
    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
    'Sibiu': {'Arad': 140, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    # Add more cities and connections...
}

# Heuristic values (straight-line distance to Bucharest)
heuristics = {
    'Arad': 366,
    'Sibiu': 253,
    'Fagaras': 176,
    'Bucharest': 0,
    # Add more cities...
}

# BFS Implementation
def bfs(graph, start, goal):
    from collections import deque
    queue = deque([(start, [start], 0)])  # (node, path, cost)
    while queue:
        (node, path, cost) = queue.popleft()
        if node == goal:
            return path, cost
        for neighbor, distance in graph[node].items():
            if neighbor not in path:
                queue.append((neighbor, path + [neighbor], cost + distance))
    return None, float('inf')

# UCS Implementation
def ucs(graph, start, goal):
    import heapq
    queue = [(0, start, [start])]  # (cost, node, path)
    while queue:
```

```
        (cost, node, path) = heapq.heappop(queue)
        if node == goal:
            return path, cost
        for neighbor, distance in graph[node].items():
            if neighbor not in path:
                heapq.heappush(queue, (cost + distance, neighbor, path +
[neighbor]))
    return None, float('inf')

# GBFS Implementation
def gbfs(graph, start, goal, heuristics):
    import heapq
    queue = [(heuristics[start], start, [start], 0)]  # (heuristic, node, path,
cost)
    while queue:
        (_, node, path, cost) = heapq.heappop(queue)
        if node == goal:
            return path, cost
        for neighbor, distance in graph[node].items():
            if neighbor not in path:
                heapq.heappush(queue, (heuristics[neighbor], neighbor, path +
[neighbor], cost + distance))
    return None, float('inf')

# IDDFS Implementation
def iddfs(graph, start, goal):
    def dls(node, goal, depth, path, cost):
        if node == goal:
            return path, cost
        if depth <= 0:
            return None, float('inf')
        for neighbor, distance in graph[node].items():
            if neighbor not in path:
                result, total_cost = dls(neighbor, goal, depth - 1, path +
[neighbor], cost + distance)
                if result:
                    return result, total_cost
        return None, float('inf')

    depth = 0
    while True:
        result, cost = dls(start, goal, depth, [start], 0)
        if result:
            return result, cost
```

```
        depth += 1

# User input for source and destination
start = input("Enter source city: ")
goal = input("Enter destination city: ")

# Run all algorithms
bfs_path, bfs_cost = bfs(romania_map, start, goal)
ucs_path, ucs_cost = ucs(romania_map, start, goal)
gbfs_path, gbfs_cost = gbfs(romania_map, start, goal, heuristics)
iddfs_path, iddfs_cost = iddfs(romania_map, start, goal)

# Print results
print("BFS Path:", bfs_path, "Cost:", bfs_cost)
print("UCS Path:", ucs_path, "Cost:", ucs_cost)
print("GBFS Path:", gbfs_path, "Cost:", gbfs_cost)
print("IDDFS Path:", iddfs_path, "Cost:", iddfs_cost)

# Compare algorithms
algorithms = {
    "BFS": bfs_cost,
    "UCS": ucs_cost,
    "GBFS": gbfs_cost,
    "IDDFS": iddfs_cost
}
best_algorithm = min(algorithms, key=algorithms.get)
print("Best Algorithm:", best_algorithm, "with cost:",
algorithms[best_algorithm])
```

## OUTPUT:

```
Enter source city: Arad
Enter destination city: Bucharest

BFS Path: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'] Cost: 450
UCS Path: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest'] Cost: 418
GBFS Path: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'] Cost: 450
IDDFS Path: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'] Cost: 450

Best Algorithm: UCS with cost: 418
```