Sure! Here's the revised version with **slightly more elaborate theory** and all answers formatted with **question, aim, theory, and code**.

---

1. Write a program to implement DFS and BFS

Aim:
To traverse a graph using Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms.

Theory:
DFS explores a path as deep as possible before backtracking. It uses recursion or a stack. BFS explores all neighbors at the current level before moving to the next, using a queue. These algorithms are commonly used in pathfinding, network analysis, and AI search.

Code:

```
from collections import deque


graph = {
  'A': ['B', 'C'],
  'B': ['D', 'E'],
  'C': ['F'],
  'D': [], 'E': [], 'F': []
}


def dfs(start, visited=set()):
  if start not in visited:
    print(start, end=' ')
    visited.add(start)
    for neighbor in graph[start]:
      dfs(neighbor, visited)


def bfs(start):
  visited = set()
  queue = deque([start])
  while queue:
    node = queue.popleft()
    if node not in visited:
```

```
        print(node, end=' ')
        visited.add(node)
        queue.extend(graph[node])

print("DFS:")
dfs('A')
print("\nBFS:")
bfs('A')
```

---

2. Write a Program to find the solution for Travelling Salesman Problem

Aim:
To determine the shortest path that visits all cities once and returns to the origin.

Theory:
The Travelling Salesman Problem (TSP) is a classic NP-hard optimization problem. It models routing, logistics, and manufacturing. Brute-force generates all possible tours and selects the one with the minimum cost.

Code:

```
from itertools import permutations

dist = {
   ('A', 'B'): 10, ('A', 'C'): 15, ('A', 'D'): 20,
   ('B', 'A'): 10, ('B', 'C'): 35, ('B', 'D'): 25,
   ('C', 'A'): 15, ('C', 'B'): 35, ('C', 'D'): 30,
   ('D', 'A'): 20, ('D', 'B'): 25, ('D', 'C'): 30
}
cities = ['A', 'B', 'C', 'D']
min_path = float('inf')

for perm in permutations(cities[1:]):
   path = ['A'] + list(perm) + ['A']
   cost = sum(dist[(path[i], path[i+1])] for i in range(len(path)-1))
   if cost < min_path:
```

```
        min_path = cost
        best_path = path

print("Best Path:", best_path, "Cost:", min_path)
```

---

3. Write a program to implement Simulated Annealing Algorithm

Aim:
To solve optimization problems by probabilistically accepting worse solutions.

Theory:
Simulated Annealing is inspired by the process of metal annealing. It allows random moves to avoid local minima. Over time, the algorithm "cools" and becomes more selective. It's used in scheduling, routing, and neural net training.

Code:

```
import math, random

def cost(x): return x**2 + 4*x + 4

def simulated_annealing():
    x = random.uniform(-10, 10)
    T = 1000
    cooling = 0.95
    while T > 1e-3:
        new_x = x + random.uniform(-1, 1)
        delta = cost(new_x) - cost(x)
        if delta < 0 or random.random() < math.exp(-delta / T):
            x = new_x
        T *= cooling
    return x, cost(x)

solution, val = simulated_annealing()
print("Minimum at x =", round(solution, 2), "Value =", round(val, 2))
```

---

4. Write a program to find the solution for Wumpus World Problem

Aim:
To simulate an AI agent safely navigating Wumpus World to find gold.

Theory:
Wumpus World is a logic-based AI problem with hidden pits and a monster. The agent uses percepts (like breeze or stench) and inference to explore safely. It models reasoning under uncertainty, key in AI planning.

Code:

```
world = [['', 'W', '', 'G'],
        ['P', '', 'P', ''],
        ['', 'B', '', ''],
        ['A', '', '', '']]


agent_pos = (3, 0)
safe = set([agent_pos])
moves = [(0,1),(1,0),(-1,0),(0,-1)]


def get_safe_moves(pos):
    x, y = pos
    possible = []
    for dx, dy in moves:
        nx, ny = x+dx, y+dy
        if 0<=nx<4 and 0<=ny<4 and world[nx][ny] not in ['W', 'P']:
            possible.append((nx, ny))
    return possible


path = []
def dfs(pos):
    path.append(pos)
    if world[pos[0]][pos[1]] == 'G':
        return True
    for move in get_safe_moves(pos):
```

```
        if move not in path:
            if dfs(move):
                return True
    path.pop()
    return False

dfs(agent_pos)
print("Path to gold:", path)
```

---

5. Write a program to implement 8 Puzzle Problem

Aim:
To solve the 8-puzzle using BFS for shortest move sequence.

Theory:
The 8-puzzle involves moving tiles to match a goal state by sliding the blank space. It is a classic AI search problem. BFS is used to ensure the shortest path solution.

Code:

```
from collections import deque

goal = '12345678_'
start = '2831647_5'

def get_neighbors(state):
    i = state.index('_')
    row, col = divmod(i, 3)
    moves = []
    for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
        r, c = row + dx, col + dy
        if 0 <= r < 3 and 0 <= c < 3:
            j = r*3 + c
            lst = list(state)
            lst[i], lst[j] = lst[j], lst[i]
            moves.append(''.join(lst))
```

```
    return moves


def bfs(start):
    queue = deque([(start, [])])
    visited = set()
    while queue:
        state, path = queue.popleft()
        if state == goal:
            return path + [state]
        if state not in visited:
            visited.add(state)
            for neighbor in get_neighbors(state):
                queue.append((neighbor, path + [state]))


solution = bfs(start)
print("Steps to solve:", len(solution)-1)
for s in solution:
    print(s[:3], "\n"+s[3:6], "\n"+s[6:], "\n")
```

---

6. Write a Program to Implement 8-Queens Problem

Aim:
To place 8 queens on a chessboard so that no two queens attack each other.

Theory:
The 8-Queens problem is a classic constraint satisfaction problem (CSP). A solution requires that no two queens share the same row, column, or diagonal.

Code:

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or \
           abs(board[i] - col) == abs(i - row):
            return False
    return True
```

```
def solve(row=0, board=[]):
  if row == 8:
    print(board)
    return
  for col in range(8):
    if is_safe(board, row, col):
      solve(row + 1, board + [col])

solve()
```

---

7. Write a Program to Implement Towers of Hanoi Problem

Aim:
To move all disks from source to destination using a helper rod.

Theory:
Towers of Hanoi is a recursive problem where you move n-1 disks to an auxiliary peg, move the largest disk, then move n-1 disks onto it.

Code:

```
def hanoi(n, src, aux, dest):
  if n == 1:
    print(f"Move disk 1 from {src} to {dest}")
    return
  hanoi(n-1, src, dest, aux)
  print(f"Move disk {n} from {src} to {dest}")
  hanoi(n-1, aux, src, dest)

hanoi(3, 'A', 'B', 'C')
```

---

8. Developing Best First Search and A Algorithm for real world problems*

Aim:
To find the optimal path in a graph using heuristics (Best-First and A*).

Theory:
Best-First Search uses only the heuristic function (greedy). A* combines cost so far and heuristic: f(n) = g(n) + h(n). Both are used in pathfinding like GPS and games.

*Code (A Search):**

```
import heapq


graph = {
  'A': [('B', 1), ('C', 3)],
  'B': [('D', 3)],
  'C': [('D', 1)],
  'D': []
}
heuristic = {'A': 4, 'B': 2, 'C': 2, 'D': 0}


def astar(start, goal):
  pq = [(heuristic[start], 0, start, [])]
  while pq:
    f, g, node, path = heapq.heappop(pq)
    path = path + [node]
    if node == goal:
      return path
    for neighbor, cost in graph[node]:
      heapq.heappush(pq, (g+cost+heuristic[neighbor], g+cost, neighbor, path))


print("A* Path:", astar('A', 'D'))
```

---

9. Write a program to implement Hill Climbing Algorithm

Aim:
To find a local optimum by iteratively improving the current solution.

Theory:
Hill Climbing is a local search algorithm. It selects the neighbor with the best score and stops when no improvements are possible. Used in scheduling, planning, etc.

Code:

```
import random

def objective(x): return -(x**2) + 10

def hill_climb():
  x = random.uniform(-10, 10)
  step = 0.1
  while True:
    neighbors = [x + step, x - step]
    next_x = max(neighbors, key=objective)
    if objective(next_x) <= objective(x):
      break
    x = next_x
  return x, objective(x)

x, val = hill_climb()
print("Best x:", round(x, 2), "Value:", round(val, 2))
```

---

10. Implementation of Minimax Algorithm for an Application

Aim:
To build a decision-making agent for two-player games like Tic-Tac-Toe using Minimax.

Theory:
Minimax is used in adversarial games. The player maximizes their score while minimizing the opponent's. It's recursive and explores all possible moves.

Code (Tic-Tac-Toe Example):

```
def minimax(board, depth, is_max):
  scores = {'X': -1, 'O': 1, 'Tie': 0}
  result = check_winner(board)
  if result:
    return scores[result]

  if is_max:
```

```
        best = -float('inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                best = max(best, minimax(board, depth+1, False))
                board[i] = ' '
        return best
    else:
        best = float('inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                best = min(best, minimax(board, depth+1, True))
                board[i] = ' '
        return best

def check_winner(board):
    combos = [(0,1,2),(3,4,5),(6,7,8),
              (0,3,6),(1,4,7),(2,5,8),
              (0,4,8),(2,4,6)]
    for a,b,c in combos:
        if board[a] == board[b] == board[c] != ' ':
            return board[a]
    if ' ' not in board:
        return 'Tie'
    return None

# Usage Example
board = ['X', 'O', 'X',
         ' ', 'O', ' ',
         ' ', ' ', 'X']
```

```
print("Minimax Score:", minimax(board, 0, True))
```

---

11. Write a Program to Implement Alpha-Beta Pruning

Aim:
To optimize Minimax by pruning unnecessary branches in game trees.

Theory:
Alpha-Beta Pruning avoids exploring branches that won't affect the final decision. It keeps track of alpha (best max score) and beta (best min score).

Code:

```
def alphabeta(depth, nodeIndex, isMax, values, alpha, beta):
  if depth == 3:
    return values[nodeIndex]


  if isMax:
    best = -float('inf')
    for i in range(2):
      val = alphabeta(depth+1, nodeIndex*2+i, False, values, alpha, beta)
      best = max(best, val)
      alpha = max(alpha, best)
      if beta <= alpha:
        break
    return best
  else:
    best = float('inf')
    for i in range(2):
      val = alphabeta(depth+1, nodeIndex*2+i, True, values, alpha, beta)
      best = min(best, val)
      beta = min(beta, best)
      if beta <= alpha:
        break
    return best
```

```
values = [3, 5, 6, 9, 1, 2, 0, -1]

print("Optimal value:", alphabeta(0, 0, True, values, -float('inf'), float('inf')))
```

---

12. Write a program to implement Tic-Tac-Toe game

Aim:
To build a two-player Tic-Tac-Toe game with a simple interface.

Theory:
Tic-Tac-Toe is a turn-based 3x3 grid game. Players take turns marking cells. The game ends when a player wins or the board is full.

Code:

```
def print_board(board):
    for i in range(0, 9, 3):
        print(board[i:i+3])


def check(board):
    lines = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]
    for a,b,c in lines:
        if board[a] == board[b] == board[c] != ' ':
            return board[a]
    if ' ' not in board:
        return 'Tie'
    return None


board = [' '] * 9
turn = 'X'


while True:
    print_board(board)
    move = int(input(f"{turn}'s turn (0-8): "))
    if board[move] == ' ':
        board[move] = turn
        winner = check(board)
```

```
    if winner:
        print_board(board)
        print("Winner:", winner)
        break
    turn = 'O' if turn == 'X' else 'X'
```

---

13. Write a Program to Implement Water-Jug problem

Aim:
To find a sequence of steps to measure a target amount using two jugs.

Theory:
This is a classic AI problem. States are represented as jug capacities. We use BFS to find the shortest solution path.

Code:

```
from collections import deque

def water_jug(x, y, target):
    visited = set()
    queue = deque()
    queue.append((0, 0))

    while queue:
        a, b = queue.popleft()
        if (a, b) in visited:
            continue
        visited.add((a, b))
        print(f"Jug1: {a}, Jug2: {b}")
        if a == target or b == target:
            break
        queue.extend([
            (x, b), (a, y), (0, b), (a, 0),
            (min(a + b, x), b - (min(a + b, x) - a)) if b > 0 else (a, b),
            (a - (min(a + b, y) - b), min(a + b, y)) if a > 0 else (a, b)
```

```
    ])


water_jug(4, 3, 2)
```

---

14. Write a Program to Implement Monkey Banana Problem

Aim:
To simulate a monkey getting a banana using actions like move, push, climb.

Theory:
Monkey Banana problem is a planning problem. The goal is to reach a state where the monkey has the banana by using action sequences in a logic-based environment.

Code:

```
def monkey_banana():
  state = {
    'monkey': 'floor',
    'box': 'center',
    'monkey_pos': 'A',
    'banana': 'center',
    'has_banana': False
  }

  actions = []
  if state['monkey_pos'] != state['box']:
    actions.append("Monkey moves to box")
    state['monkey_pos'] = state['box']
  if state['monkey'] == 'floor':
    actions.append("Monkey pushes box to banana")
    state['box'] = state['banana']
    state['monkey_pos'] = 'center'
    actions.append("Monkey climbs box")
    state['monkey'] = 'box'
  if state['box'] == state['banana']:
    actions.append("Monkey grabs banana")
```

```
    state['has_banana'] = True

  for act in actions:
    print(act)
  print("Banana acquired:", state['has_banana'])

monkey_banana()
```

---

15. Implementation of Block World Problem

Aim:
To simulate block stacking with rules and a goal configuration.

Theory:
Block world is a classic AI planning problem. States consist of stacks of blocks. The goal is to transform one configuration into another using valid moves.

Code:

```
def block_world(start, goal):
  moves = []
  while start != goal:
    for i, stack in enumerate(start):
      if stack and stack[-1] != goal[i][-1]:
        block = stack.pop()
        for j in range(len(start)):
          if i != j:
            start[j].append(block)
            moves.append(f"Move {block} from Stack {i+1} to Stack {j+1}")
            break
        break
  for m in moves:
    print(m)

start_state = [['A', 'B'], ['C'], []]
goal_state = [[], [], ['A', 'B', 'C']]
```

block_world(start_state, goal_state)

---

16. Implementation of Constraint Satisfaction Problems (CSP)

Aim:
To solve CSPs by assigning values to variables under given constraints.

Theory:
CSP involves variables, domains, and constraints. Common examples include map coloring, Sudoku, and scheduling. We use backtracking and constraint propagation.

Code (Map Coloring Example):

```
colors = ['Red', 'Green', 'Blue']

neighbors = {
  'WA': ['NT', 'SA'],
  'NT': ['WA', 'SA', 'Q'],
  'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],
  'Q': ['NT', 'SA', 'NSW'],
  'NSW': ['Q', 'SA', 'V'],
  'V': ['SA', 'NSW'],
  'T': []
}
assignment = {}


def is_valid(state, var, value):
  for neighbor in neighbors[var]:
    if neighbor in state and state[neighbor] == value:
      return False
  return True


def backtrack(state):
  if len(state) == len(neighbors):
    return state
  var = [v for v in neighbors if v not in state][0]
  for value in colors:
```

```
    if is_valid(state, var, value):
        new_state = state.copy()
        new_state[var] = value
        result = backtrack(new_state)
        if result:
            return result
  return None

print(backtrack({}))
```

---

17. Implementation of Knowledge Representation Schemes – Use Case

Aim:
To infer knowledge from a given knowledge base (KB) using logical reasoning.

Theory:
Knowledge representation involves formalizing facts and inference rules in FOL (First-Order Logic). Forward or backward chaining is used for inference.

Given KB:

- Marcus was a man
- Marcus was a Pompeian
- All Pompeians were Romans
- Caesar was a ruler
- All Romans were either loyal to Caesar or hated him
- Everyone is loyal to someone
- People only try to assassinate rulers they are not loyal to
- Marcus tried to assassinate Caesar
- All men are people

Task: Did Marcus hate Caesar?

Reasoning Chain (Logic):

1. Marcus is a man ⇒ Marcus is a person
2. Marcus is a Pompeian ⇒ Marcus is a Roman
3. Romans ⇒ loyal or hate Caesar
4. People only try to assassinate rulers they are **not loyal to**

5. Marcus tried to assassinate Caesar ⇒ Marcus is **not loyal** to Caesar
6. ⇒ By (3), Marcus **must hate Caesar**

Conclusion:
Yes, Marcus hated Caesar.

---

18. Expert System Case Study

Aim:
To build a simple rule-based expert system for medical diagnosis.

Theory:
Expert systems use if-then rules and an inference engine. Knowledge is encoded as facts and rules. They mimic human expert reasoning.

Code (Flu Diagnosis):

```
def flu_diagnosis(symptoms):
  rules = [
    ({"fever", "cough", "body ache"}, "You may have flu."),
    ({"fever", "rash"}, "You may have measles."),
    ({"cough"}, "You might have a common cold.")
  ]
  for cond, diag in rules:
    if cond.issubset(symptoms):
      return diag
  return "No clear diagnosis."

# Example
symptoms = {"fever", "cough", "body ache"}
print(flu_diagnosis(symptoms))
```

---

19. Implementation of Unification and Resolution for Real World Problems

Aim:
To implement FOL unification and resolution for AI reasoning.

Theory:
Unification finds substitutions to make predicates identical. Resolution is a rule of inference for automated theorem proving.

Code (Unification Example):

```
def unify(x, y, subs={}):
    if x == y:
        return subs
    if isinstance(x, str) and x.islower():
        return unify_var(x, y, subs)
    if isinstance(y, str) and y.islower():
        return unify_var(y, x, subs)
    if isinstance(x, tuple) and isinstance(y, tuple) and x[0] == y[0]:
        for a, b in zip(x[1:], y[1:]):
            subs = unify(a, b, subs)
            if subs is None:
                return None
        return subs
    return None

def unify_var(var, x, subs):
    if var in subs:
        return unify(subs[var], x, subs)
    elif x in subs:
        return unify(var, subs[x], subs)
    else:
        subs[var] = x
        return subs

# Example
x = ('knows', 'john', 'X')
y = ('knows', 'john', 'mary')
print("Unification result:", unify(x, y))
```