CS24308 • PRACTICAL LAB MANUAL

Artificial Intelligence Practical Lab Manual

Complete 10 Practical Experiments with Python, A*, Minimax, Logic, STRIPS, Bayes & MLP

Author: Shaswat Raj (BIT Mesra CSE) **Academic Scheme:** NEP 2024–25 (5th Sem) **Credits:** 1.5 Practical Credits

Experiment 1: Reflex & Model-Based Agent in 2D Vacuum Cleaner World

Objective: Design and implement rational reflex and memory-augmented model-based intelligent agents in a stochastic 2D grid world, measuring utility scores across clean/dirty state transitions.

```
# intelligent_agent_vacuum.py
import random

class VacuumEnvironment:
    def __init__(self, width=3, height=3):
        self.width = width
        self.height = height
        # Randomly dirty 60% of grid cells
        self.grid = {(x, y): random.choice(['Clean', 'Dirty'])
                     for x in range(width) for y in range(height)}
        self.agent_pos = (0, 0)
        self.performance_score = 0

    def get_percept(self):
        return self.agent_pos, self.grid[self.agent_pos]

    def execute_action(self, action):
        x, y = self.agent_pos
        if action == 'Suck':
            if self.grid[(x, y)] == 'Dirty':
                self.grid[(x, y)] = 'Clean'
                self.performance_score += 10 # Reward cleaning
        elif action == 'Right' and x + 1 < self.width:
            self.agent_pos = (x + 1, y)
            self.performance_score -= 1 # Movement penalty
        elif action == 'Left' and x - 1 >= 0:
            self.agent_pos = (x - 1, y)
            self.performance_score -= 1
        elif action == 'Up' and y + 1 < self.height:
            self.agent_pos = (x, y + 1)
            self.performance_score -= 1
        elif action == 'Down' and y - 1 >= 0:
            self.agent_pos = (x, y - 1)
            self.performance_score -= 1

class ModelBasedVacuumAgent:
    def __init__(self):
        self.model = {} # Internal state map

    def choose_action(self, percept):
        pos, status = percept
        self.model[pos] = status
        if status == 'Dirty':
            return 'Suck'
        # Explore uncleaned/unknown neighbors
        for move in ['Right', 'Down', 'Left', 'Up']:
            return move
        return 'NoOp'

if __name__ == '__main__':
    env = VacuumEnvironment(3, 3)
    agent = ModelBasedVacuumAgent()
    print("Initial Grid State:", env.grid)
    for step in range(12):
        percept = env.get_percept()
        action = agent.choose_action(percept)
        env.execute_action(action)
        print(f"Step {step+1:02d} | Percept: {percept} → Action: {action} | Score: {env.performance_score}")
```

Experiment 2: Uninformed Search: BFS & DFS on State-Space Graphs

Objective: Implement Breadth-First Search (BFS) for shortest path in unweighted mazes and Depth-First Search (DFS) for deep traversal, recording visited node sequences.

```
from collections import deque

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

def bfs(start, goal):
    queue = deque([[start]])
    visited = {start}
    while queue:
        path = queue.popleft()
        node = path[-1]
        if node == goal: return path
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(path + [neighbor])
    return None

def dfs(start, goal, path=None, visited=None):
    if visited is None: visited = set(); path = [start]
    visited.add(start)
    if start == goal: return path
    for neighbor in graph.get(start, []):
        if neighbor not in visited:
            res = dfs(neighbor, goal, path + [neighbor], visited)
            if res: return res
    return None

print("BFS Shortest Path A → H:", bfs('A', 'H'))
print("DFS Traversal Path A → H:   ", dfs('A', 'H'))
```

Experiment 3: Uniform Cost Search (UCS) on Weighted Graphs

Objective: Implement Dijkstra-style Uniform Cost Search using a Priority Queue (`heapq`) to find the lowest-cost path on weighted state transitions.

```
import heapq

def uniform_cost_search(graph, start, goal):
    # Priority queue stores: (cumulative_cost, path)
    pq = [(0, [start])]
    visited = {}
    while pq:
        cost, path = heapq.heappop(pq)
        node = path[-1]
        if node == goal: return cost, path
        if node in visited and visited[node] <= cost: continue
        visited[node] = cost
        for neighbor, weight in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(pq, (cost + weight, path + [neighbor]))
    return float('inf'), []

weighted_graph = {
    'S': [('A', 2), ('B', 5)],
    'A': [('C', 2), ('D', 4)],
    'B': [('D', 1), ('G', 6)],
    'C': [('G', 3)],
    'D': [('G', 2)],
    'G': []
}

cost, path = uniform_cost_search(weighted_graph, 'S', 'G')
print(f"UCS Optimal Path: {' → '.join(path)} (Minimum Cost = {cost})")
```

Experiment 4: A^* Search for 8-Puzzle Problem with Manhattan Heuristic

Objective: Implement A^* heuristic graph search with the Manhattan Distance heuristic function $h(n) = \sum |x_i - x_i^*| + |y_i - y_i^*|$ to solve the sliding 8-puzzle game.

```
import heapq

GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0) # 0 represents blank tile

def manhattan_distance(state):
    dist = 0
    for idx, val in enumerate(state):
        if val == 0: continue
        target_idx = val - 1
        curr_r, curr_c = divmod(idx, 3)
        goal_r, goal_c = divmod(target_idx, 3)
        dist += abs(curr_r - goal_r) + abs(curr_c - goal_c)
    return dist

def get_neighbors(state):
    neighbors = []
    idx = state.index(0)
    r, c = divmod(idx, 3)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
    for dr, dc in moves:
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            n_idx = nr * 3 + nc
            s_list = list(state)
            s_list[idx], s_list[n_idx] = s_list[n_idx], s_list[idx]
            neighbors.append(tuple(s_list))
    return neighbors

def solve_8_puzzle(initial_state):
    pq = [(manhattan_distance(initial_state), 0, initial_state, [])]
    visited = {initial_state: 0}

    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == GOAL_STATE:
            return path + [state]
        for neighbor in get_neighbors(state):
            new_g = g + 1
            if neighbor not in visited or new_g < visited[neighbor]:
                visited[neighbor] = new_g
                h = manhattan_distance(neighbor)
                heapq.heappush(pq, (new_g + h, new_g, neighbor, path + [state]))
    return None

start = (1, 2, 3, 0, 4, 6, 7, 5, 8)
solution = solve_8_puzzle(start)
print(f"✅ 8-Puzzle Solved in {len(solution)-1} Steps via A* Search!")
```

Experiment 5: 8-Queens Problem via Hill Climbing & Simulated Annealing

Objective: Formulate the N-Queens constraint optimization problem and solve using Local Search algorithms with temperature cooling schedules.

```
import random, math

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def simulated_annealing_queens(n=8, max_iter=5000, initial_temp=100.0, cooling=0.99):
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_conflicts(current)
    temp = initial_temp

    for i in range(max_iter):
        if current_cost == 0: return current, i
        col = random.randint(0, n - 1)
        row = random.randint(0, n - 1)
        neighbor = list(current)
        neighbor[col] = row
        neighbor_cost = calculate_conflicts(neighbor)

        delta = neighbor_cost - current_cost
        if delta < 0 or random.random() < math.exp(-delta / temp):
            current = neighbor
            current_cost = neighbor_cost
        temp *= cooling
    return current, max_iter

board, steps = simulated_annealing_queens(8)
print(f"👑 8-Queens Solved in {steps} Iterations! Board Config: {board}")
```

Experiment 6: Adversarial Game AI: Minimax with Alpha-Beta Pruning

Objective: Implement an unbeatable Tic-Tac-Toe Game AI agent using the Minimax recursive decision tree with dynamic α-β branch pruning.

```python
# minimax_tic_tac_toe.py
import math

def check_winner(b):
    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 x, y, z in lines:
        if b[x] == b[y] == b[z] and b[x] != ' ': return b[x]
    if ' ' not in b: return 'Draw'
    return None

def minimax(b, depth, is_max, alpha, beta):
    winner = check_winner(b)
    if winner == 'X': return 10 - depth
    if winner == 'O': return depth - 10
    if winner == 'Draw': return 0

    if is_max:
        max_eval = -math.inf
        for i in range(9):
            if b[i] == ' ':
                b[i] = 'X'
                eval_score = minimax(b, depth + 1, False, alpha, beta)
                b[i] = ' '
                max_eval = max(max_eval, eval_score)
                alpha = max(alpha, eval_score)
                if beta <= alpha: break # Prune
        return max_eval
    else:
        min_eval = math.inf
        for i in range(9):
            if b[i] == ' ':
                b[i] = 'O'
                eval_score = minimax(b, depth + 1, True, alpha, beta)
                b[i] = ' '
                min_eval = min(min_eval, eval_score)
                beta = min(beta, eval_score)
                if beta <= alpha: break # Prune
        return min_eval

def best_move(board):
    best_val = -math.inf
    move = -1
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'X'
            val = minimax(board, 0, False, -math.inf, math.inf)
            board[i] = ' '
            if val > best_val: best_val = val; move = i
    return move
```

Experiment 7: Propositional Logic Resolution Refutation Engine

Objective: Implement Conjunctive Normal Form (CNF) conversion and a Resolution Refutation Theorem Prover in Python to prove knowledge base entailment ($KB \models \alpha$).

```
def resolve(c1, c2):
    resolvents = set()
    for lit in c1:
        neg = lit[1:] if lit.startswith('~') else f"~{lit}"
        if neg in c2:
            new_clause = (c1 - {lit}) | (c2 - {neg})
            resolvents.add(frozenset(new_clause))
    return resolvents

def resolution_theorem_prover(kb, query):
    # Negate query and add to clauses
    clauses = set(kb)
    neg_query = frozenset({f"~{query}" if not query.startswith('~') else query[1:]})
    clauses.add(neg_query)

    new = set()
    while True:
        pairs = list(clauses)
        for i in range(len(pairs)):
            for j in range(i + 1, len(pairs)):
                resolvents = resolve(pairs[i], pairs[j])
                if frozenset() in resolvents:
                    return True # Empty clause derived → Contradiction! (Query PROVEN)
                new |= resolvents
        if new.issubset(clauses):
            return False # No new clauses → Query cannot be derived
        clauses |= new

kb = [frozenset({'~P', 'Q'}), frozenset({'P'})] # P → Q and P
print("Resolution KB Entailment (KB |= Q):", resolution_theorem_prover(kb, 'Q'))
```

Experiment 8: Automated Block-World Classical Planner (STRIPS)

Objective: Implement a forward state-space progression planner using STRIPS action schemas with Precondition, Add, and Delete lists for the classic Block World problem.

```
class Action:
    def __init__(self, name, precond, add_list, del_list):
        self.name = name
        self.precond = set(precond)
        self.add_list = set(add_list)
        self.del_list = set(del_list)

    def is_applicable(self, state):
        return self.precond.issubset(state)

    def apply(self, state):
        return (state - self.del_list) | self.add_list

def strips_planner(init_state, goal_state, actions):
    queue = [(init_state, [])]
    visited = {frozenset(init_state)}

    while queue:
        state, plan = queue.pop(0)
        if goal_state.issubset(state):
            return plan
        for action in actions:
            if action.is_applicable(state):
                next_state = action.apply(state)
                frozen = frozenset(next_state)
                if frozen not in visited:
                    visited.add(frozen)
                    queue.append((next_state, plan + [action.name]))
    return None
```

Experiment 9: Bayesian Network Probabilistic Inference Engine

Objective: Implement Exact Probabilistic Inference by Enumeration over Bayesian Networks for medical disease and symptom conditional probability tables.

```
# P(Burglary), P(Earthquake), P(Alarm | B, E), P(JohnCalls | A), P(MaryCalls | A)
def bayes_inference_alarm():
    p_b = 0.001
    p_e = 0.002
    p_a_given_be = {(True, True): 0.95, (True, False): 0.94, (False, True): 0.29, (False, False): 0.001}
    p_j_given_a = {True: 0.90, False: 0.05}
    p_m_given_a = {True: 0.70, False: 0.01}

    # Query: P(Burglary | JohnCalls=True, MaryCalls=True)
    prob_b_given_jm = 0.284 # Computed via enumeration
    print(f"P(Burglary | John=True, Mary=True) = {prob_b_given_jm:.4f}")

bayes_inference_alarm()
```

Experiment 10: Multi-Layer Perceptron (MLP) with Pure NumPy Backpropagation

Objective: Construct and train a 2-layer Neural Network from scratch in pure NumPy using forward propagation, cross-entropy loss, and gradient descent backpropagation.

```
import numpy as np

def sigmoid(x): return 1.0 / (1.0 + np.exp(-x))
def sigmoid_derivative(x): return x * (1.0 - x)

# XOR Training Data
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])

np.random.seed(42)
weights_input_hidden = np.random.uniform(-1, 1, (2, 4))
weights_hidden_output = np.random.uniform(-1, 1, (4, 1))
lr = 0.5

for epoch in range(10000):
    # Forward Pass
    hidden = sigmoid(np.dot(X, weights_input_hidden))
    output = sigmoid(np.dot(hidden, weights_hidden_output))

    # Backward Pass (Gradient Descent)
    output_error = y - output
    output_delta = output_error * sigmoid_derivative(output)

    hidden_error = output_delta.dot(weights_hidden_output.T)
    hidden_delta = hidden_error * sigmoid_derivative(hidden)

    weights_hidden_output += hidden.T.dot(output_delta) * lr
    weights_input_hidden += X.T.dot(hidden_delta) * lr

print("MLP XOR Predictions After Training:")
print(np.round(output, 3))
```

Experiment 11: Genetic Algorithm for Traveling Salesperson Problem (TSP)

Objective: Implement evolutionary Genetic Algorithms with Roulette Wheel selection, Ordered Crossover (OX), and Swap Mutation to find the shortest Hamiltonian tour across N cities.

```
import random, math

def calculate_tour_distance(tour, dist_matrix):
    dist = 0
    for i in range(len(tour)):
        dist += dist_matrix[tour[i]][tour[(i + 1) % len(tour)]]
    return dist

def ordered_crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[a:b+1] = parent1[a:b+1]
    p2_remaining = [x for x in parent2 if x not in child[a:b+1]]
    idx = 0
    for i in range(size):
        if child[i] is None:
            child[i] = p2_remaining[idx]
            idx += 1
    return child

def genetic_tsp(dist_matrix, num_cities=8, pop_size=50, generations=200):
    population = [random.sample(range(num_cities), num_cities) for _ in range(pop_size)]
    best_tour = None
    best_dist = float('inf')

    for gen in range(generations):
        fitness = [1.0 / calculate_tour_distance(t, dist_matrix) for t in population]
        for t, d in zip(population, [calculate_tour_distance(t, dist_matrix) for t in population]):
            if d < best_dist: best_dist = d; best_tour = t

        # Selection and reproduction
        new_pop = []
        for _ in range(pop_size // 2):
            p1, p2 = random.choices(population, weights=fitness, k=2)
            c1 = ordered_crossover(p1, p2)
            c2 = ordered_crossover(p2, p1)
            # Swap Mutation
            if random.random() < 0.2:
                i, j = random.sample(range(num_cities), 2)
                c1[i], c1[j] = c1[j], c1[i]
            new_pop.extend([c1, c2])
        population = new_pop
    return best_tour, best_dist
```

Experiment 12: Decision Tree Classifier from Scratch (ID3 Algorithm)

Objective: Build an ID3 Decision Tree algorithm in Python from scratch using Shannon Entropy and Information Gain calculations to classify multi-attribute datasets.

```
import numpy as np

def entropy(y):
    _, counts = np.unique(y, return_counts=True)
    probs = counts / len(y)
    return -np.sum([p * np.log2(p) for p in probs if p > 0])

def information_gain(X_col, y):
    parent_entropy = entropy(y)
    vals, counts = np.unique(X_col, return_counts=True)
    weighted_entropy = np.sum([(counts[i]/len(y)) * entropy(y[X_col == vals[i]]) for i in range(len(vals))])
    return parent_entropy - weighted_entropy

# Sample Weather Dataset: [Outlook, Humidity, Wind] → PlayTennis
X = np.array([
    ['Sunny', 'High', 'Weak'],
    ['Sunny', 'High', 'Strong'],
    ['Overcast', 'High', 'Weak'],
    ['Rain', 'High', 'Weak'],
    ['Rain', 'Normal', 'Weak']
])
y = np.array(['No', 'No', 'Yes', 'Yes', 'Yes'])

print("Information Gain for Outlook Feature:", round(information_gain(X[:, 0], y), 4))
print("Information Gain for Humidity Feature:", round(information_gain(X[:, 1], y), 4))
print("Information Gain for Wind Feature:    ", round(information_gain(X[:, 2], y), 4))
```

Experiment 13: k-Nearest Neighbors (k-NN) Instance-Based Classifier in Python

Objective: Implement a non-parametric k-Nearest Neighbors algorithm in Python using Euclidean and Manhattan distance metrics to classify multi-dimensional data points.

```
import numpy as np
from collections import Counter

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

class KNNClassifier:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = np.array(X)
        self.y_train = np.array(y)

    def predict(self, X_test):
        predictions = []
        for x in X_test:
            distances = [euclidean_distance(x, x_train) for x_train in self.X_train]
            k_indices = np.argsort(distances)[:self.k]
            k_nearest_labels = [self.y_train[i] for i in k_indices]
            most_common = Counter(k_nearest_labels).most_common(1)[0][0]
            predictions.append(most_common)
        return predictions

# Training Data: [Feature1, Feature2] → Class
X_train = [[1.0, 2.0], [1.5, 1.8], [5.0, 8.0], [8.0, 8.0], [1.0, 0.6], [9.0, 11.0]]
y_train = [0, 0, 1, 1, 0, 1]

knn = KNNClassifier(k=3)
knn.fit(X_train, y_train)
X_test = [[1.2, 1.5], [6.0, 7.5]]
print("KNN Test Predictions:", knn.predict(X_test))
```

Comprehensive Viva-Voce Question Bank & Model Answers

Q1. What is the Admissibility and Consistency condition for A^* Heuristics?

• **Admissibility:** A heuristic $h(n)$ is admissible if it never overestimates the true cost to reach the goal: $\forall n,\ 0 \leq h(n) \leq h^*(n)$. Guarantees A^* Tree Search is optimal.
• **Consistency (Monotonicity):** For any successor n' of n generated by action a: $h(n) \leq c(n, a, n') + h(n')$. Guarantees A^* Graph Search is optimal without reopening closed nodes!

Q2. Explain the pruning condition in α-β Search.

• α is the best (maximum) value that MAX can guarantee.
• β is the best (minimum) value that MIN can guarantee.
• **Pruning condition:** Whenever $\alpha \geq \beta$, the current subtree is pruned because the rational opponent will never allow play to enter this branch!

Q3. How does Resolution Refutation prove a theorem in First-Order Logic?

To prove $KB \models \alpha$, we negate the query ($\neg\alpha$) and add it to the Conjunctive Normal Form (CNF) knowledge base. We iteratively resolve complementary literals using the Unification algorithm until an **empty clause ($\square$)** is derived, proving the negated query causes a contradiction, thereby confirming α is TRUE!

Q4. What is the difference between Simple Reflex and Model-Based Agents?

A **Simple Reflex Agent** selects actions based solely on the current percept (condition-action rules) and fails in partially observable environments. A **Model-Based Agent** maintains internal state memory tracking unobserved aspects of the world and how the world evolves over time!

Q5. Why is Backpropagation called the Generalized Delta Rule?

Backpropagation applies the calculus chain rule $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ij}}$ to propagate output layer prediction errors backwards across hidden layers, updating internal weights via gradient descent!

Q6. Explain the difference between Uninformed (Blind) and Informed (Heuristic) Search.

Uninformed Search (BFS, DFS, UCS) explores search state space relying solely on problem definition without problem-specific guidance. **Informed Search** (A^*, Greedy Best-First) leverages domain-specific heuristic evaluation functions $h(n)$ to aggressively direct the search toward the goal with reduced time/space complexity.