Parul® University

Parul Institute of Technology, Limda, Vadodara

Subject: Artificial Intelligence(303105308) **Course:** B.Tech

Year &Sem: B.Tech Sem-V **Branch:** Artificial Intelligence

LAB MANNUAL

ARTIFICIAL INTELLIGENCE

Course: B.TECH **Semester: 5**

Branch: AI

Practical-1: Develop an AI-based medical diagnosis system using expert systems architecture and knowledge representation techniques.

Here's an example Python program demonstrating facts and queries:

```
# Define diseases and their symptoms as a dictionary
diseases = {
    'Common Cold': ['fever', 'cough', 'sore throat', 'runny nose',
'headache'],
    'Flu': ['fever', 'cough', 'body ache', 'fatigue', 'headache',
'chills'],
    'COVID-19': ['fever', 'cough', 'fatigue', 'loss of taste', 'loss of
smell', 'difficulty breathing'],
    'Malaria': ['fever', 'chills', 'sweating', 'headache', 'nausea'],
    'Dengue': ['fever', 'headache', 'rash', 'muscle pain', 'joint
pain'],
}

# Function to ask user about symptoms and collect confirmed ones
def ask_symptoms(symptom_list):
    print("Please answer yes/no:")  # Inform user about input format
    user_symptoms = set()           # Create empty set to store user
symptoms
    for symptom in symptom_list:    # Loop over each symptom in list
        while True:                 # Loop until valid input received
            ans = input(f"Do you have {symptom}? ").strip().lower() #
Ask user symptom question #strip removes extra space from user input
            if ans in ['yes', 'y']: # If user says yes
                user_symptoms.add(symptom) # Add symptom to set
                break # Exit inner loop, ask next symptom
            elif ans in ['no', 'n']: # If user says no
                break # Exit inner loop, ask next symptom
            else:
                print("Answer yes or no only.") ## Ask again if invalid
input
    return user_symptoms ## Return set of symptoms user has

# Find diseases that match user symptoms
def diagnose():
    # Get all symptoms from diseases
    all_symptoms = set() # Empty set for all unique symptoms
    for symptoms in diseases.values(): # Loop over symptom lists of
each disease
        all_symptoms.update(symptoms) # Add symptoms to the set

    # Ask user about all symptoms
    user_symptoms = ask_symptoms(sorted(all_symptoms))

    # Calculate how well each disease matches
    disease_scores = {} ## Dictionary to store match score for each
disease
    for disease, symptoms in diseases.items(): ## Loop over diseases
and their symptoms
        matched = user_symptoms.intersection(symptoms) ## Symptoms user
has that disease also has
        score = len(matched) / len(symptoms) ## Calculate percentage of
symptoms matched
        disease_scores[disease] = score ## Store score for disease

    max_score = max(disease_scores.values()) ## Find highest score
among diseases
    threshold = 0.5  #  Minimum required match score (50%)

    # Select diseases that match best and meet threshold ## List
diseases with highest score and above threshold # Select disease d
along with its score if score==max score and score>=threshold
    possible_diseases = [d for d, score in disease_scores.items() if
score == max_score and score >= threshold]

    # Show results
    if possible_diseases: ## If there is at least one matching disease
        print("\nYou may have:") ## Print diagnosis message
        for d in possible_diseases: # # Loop over possible diseases
            print(f"- {d} ({disease_scores[d]*100:.0f}% match)") # #
Show disease and match %
    else:
        print("\nNo matching disease found.") ## No disease matched
enough symptoms

# Run diagnosis if script is executed directly (not imported)
if __name__ == "__main__":
    print("Welcome to Medical Diagnosis System\n")  # Welcome message
    diagnose()                                     # Start diagnosis
process
```

Practical -2

Build an intelligent agent for optimizing e-commerce inventory management using search algorithms like hill climbing and best-first search.

```
import heapq  # Importing heapq for priority queue used in Best-First
Search
import random  # Importing random to start hill climbing from a random
product

# Products dictionary: product name → (cost, profit)
products = {
    'A': (5, 10),
    'B': (4, 9),
    'C': (6, 11),
    'D': (3, 5),
    'E': (7, 14)
}

budget = 15  # Maximum allowed total cost

# --- Helper Functions ---
def total_cost(selection):
    # Returns the total cost of selected products #It adds up the costs
of all products in the given selection.
    return sum(products[p][0] for p in selection)

def total_profit(selection):
    # Returns the total profit of selected products #It adds up the
profits of all products in the given selection
    return sum(products[p][1] for p in selection)

def get_neighbors(state):# Defines a function named get_neighbors that
takes state (a list of selected products) as input.
    #A neighbor is created by either adding a new product or removing
an existing one.
    """Generate neighbors by adding or removing one product."""
    neighbors = []  # Initializes an empty list to store all valid
neighbor selections.
    for p in products:#Loop over every product (A, B, C, D, E).
        #For each product p, the function will try:#Removing it (if
it's already selected) or Adding it (if it's not selected)
        new_state = state.copy()  # Create a copy of current state
        #We make a copy so we don't modify the original state directly.
        if p in state:
            new_state.remove(p)  # Remove product if already present
        else:
            new_state.append(p)  # Add product if not present
        if total_cost(new_state) <= budget:#Check if the total cost of
new_state is within budget
            # Only keep valid neighbors within the budget
            neighbors.append(new_state) #If yes, add it to the list of
valid neighbors
    return neighbors #After looping through all products, returns the
list of all valid neighbor selections #These are helpful in both Hill
Climbing and BFS
```

```
# --- Hill Climbing Algorithm ---
def hill_climbing(): #This defines the function named hill_climbing()
    # Start with one random product from the list ('A', 'B', etc.).
#list(products.keys()) → gets all product names.
    #random.choice(...) → picks one product randomly. #current_state is
a list holding the currently selected products.
    current_state = [random.choice(list(products.keys()))]
    while True:
        neighbors = get_neighbors(current_state)  #Calling of the
get_neighbors() function to generate all valid neighbors (add/remove
one product) of the current state.
        if not neighbors: #If there are no valid neighbors, break the
loop.
            break  # No neighbors to explore

        best_neighbor = max(neighbors, key=total_profit) #Among all
neighbors, pick the one with the maximum profit.
        #key=total_profit tells Python to use total_profit() as a basis
for comparison.
        if total_profit(best_neighbor) > total_profit(current_state):
#Check if the best_neighbor has a higher profit than the current
selection.
            # Move to better neighbor if profit increases
            current_state = best_neighbor
        else:
            break  # Stop if no better neighbor found (local optimum
reached).
    return current_state, total_profit(current_state)  # Return final
state and its profit

# --- Best-First Search Algorithm ---
def best_first_search():
    heap = [(-0, [])]  # Priority queue with (-profit, state); start
with empty state
    #A min-heap (priority queue) to store states, sorted by negative
profit so the highest profit comes out first.
    #Python's heapq is a min-heap, so we use negative values to
simulate max profit behavior.
    visited = set()  # Keeps track of the states we’ve already explored
(to avoid revisiting same combinations).
    best = ([], 0)  # A tuple that keeps the best solution (product
list and its profit) found so far. #Till now No product and profit zero

    while heap: #Loop until the heap is empty, means we still have
states to explore. #जब तक heap में कोई state बची है, हम search जारी रखेंगे।
        neg_profit, state = heapq.heappop(heap)  # Remove the state
with maximum profit (minimum negative profit) from heap.
        state_key = tuple(sorted(state))  # Convert to tuple to store
in set #Convert the state (list of products) into a sorted tuple #
        if state_key in visited:#Skip the current state if it's already
visited.
            continue  # Skip already visited states
        visited.add(state_key)  # Mark current state as visited #Add
this state to visited so we don’t process it again.
        profit = total_profit(state)  # Calculate current state's
profit #Compute the total profit of the current selection.
        if profit > best[1]:
            best = (state, profit)  # If current state's profit is
better than the best so far, update the best.

        for neighbor in get_neighbors(state): #Generate all valid
neighbors (within budget) of the current state.
            n_key = tuple(sorted(neighbor))  # Create a unique sorted
tuple version of the neighbor.
            if n_key not in visited:
                # If the neighbor hasn't been visited, add it to the
heap with negative profit as priority.
                heapq.heappush(heap, (-total_profit(neighbor),
neighbor))

    return best  # Return best solution and its profit

# --- Run and Compare ---
print("=== Hill Climbing ===")
hc_solution, hc_profit = hill_climbing()  # Run hill climbing algorithm
print("Selected Products:", hc_solution)  # Print selected products
print("Total Profit:", hc_profit)  # Print total profit

print("\n=== Best-First Search ===")
bfs_solution, bfs_profit = best_first_search()  # Run best-first search
algorithm
print("Selected Products:", bfs_solution)  # Print selected products
print("Total Profit:", bfs_profit)  # Print total profit
```

Practical-3: Implement a constraint satisfaction algorithm to solve scheduling problems in healthcare facilities.

```
from constraint import Problem, AllDifferentConstraint,
SomeInSetConstraint, NotInSetConstraint

def solve_nurse_scheduling():
    # 1. Create a Problem instance
    problem = Problem()

    # 2. Define Variables and their Domains
    # Variables are the shifts, and their domains are the nurses who
can work them
    nurses = ['Nurse A', 'Nurse B', 'Nurse C']
    shifts = ['Morning', 'Afternoon', 'Night']

    for shift in shifts:
        problem.addVariable(shift, nurses)

    # 3. Add Constraints

    # Constraint 1 & 2: Each shift must have exactly one nurse, and
each nurse can work at most one shift.
    # The AllDifferentConstraint ensures that all assigned values
(nurses) are unique.
```

```
    problem.addConstraint(AllDifferentConstraint(), shifts)

    # Constraint 3: Nurse A cannot work the Night shift.
    problem.addConstraint(lambda nurse: nurse != 'Nurse A',
['Night'])

    # Constraint 4: Nurse B prefers not to work the Morning shift
(but can if necessary, so we won't make it a hard constraint here
    # for simplicity, but in a real system, you might model this
with preferences/costs).
    # For a hard constraint, it would be:
    # problem.addConstraint(lambda nurse: nurse != 'Nurse B',
['Morning'])
    # Let's add it as a soft constraint later or handle it with cost
functions in a more advanced CSP.
    # For now, if we want to enforce it as a hard constraint:
    problem.addConstraint(lambda nurse: nurse != 'Nurse B',
['Morning'])


    # 4. Find Solutions
    solutions = problem.getSolutions()

    # 5. Display Solutions
    if solutions:
        print(f"Found {len(solutions)} solution(s):\n")
        for i, solution in enumerate(solutions):
            print(f"Solution {i+1}:")
            for shift in shifts:
                print(f"  {shift}: {solution[shift]}")
            print("\n”)
    else:
        print("No solutions found for the given constraints.")

if __name__ == "__main__":
    solve_nurse_scheduling()
```

4. Practical-4: Create a recommendation system for personalized learning using means-end analysis and heuristic search techniques.

```
from heapq import heappush, heappop

class LearningResource:
    def __init__(self, name, teaches):
        self.name = name
        self.teaches = set(teaches)  # concepts this resource teaches

    def __repr__(self):
        return f"{self.name}"

def heuristic(current_state, goal_state):
    # Number of concepts in goal not yet known
    return len(goal_state - current_state)

def means_end_search(resources, start_state, goal_state):
    # Priority queue for A* search: (cost + heuristic, cost, current_state,
path)
```

```
    frontier = []
    heappush(frontier, (heuristic(start_state, goal_state), 0, start_state,
[]))
    explored = set()

    while frontier:
        est_total_cost, cost, current_state, path = heappop(frontier)

        # If goal reached
        if goal_state.issubset(current_state):
            return path  # Return sequence of resources applied

        # Avoid re-exploring states
        state_key = frozenset(current_state)
        if state_key in explored:
            continue
        explored.add(state_key)

        # Means-End Analysis: choose resources that reduce difference
        difference = goal_state - current_state

        for res in resources:
            # Only consider resources that teach at least one missing
concept
            if res.teaches & difference:
                new_state = current_state | res.teaches
                new_cost = cost + 1  # uniform cost for each resource
applied
                est = new_cost + heuristic(new_state, goal_state)
                heappush(frontier, (est, new_cost, new_state, path +
[res]))

    # If no path found
    return None

# Example usage:

# Define resources
resources = [
    LearningResource("Intro to Python", ["variables", "loops",
"functions"]),
    LearningResource("Advanced Python", ["decorators", "generators",
"context managers"]),
    LearningResource("Data Structures", ["lists", "dicts", "trees"]),
    LearningResource("Algorithms", ["sorting", "searching", "recursion"]),
    LearningResource("Machine Learning Basics", ["regression",
"classification", "clustering"]),
]

# User current knowledge
start_state = set(["variables"])

# User learning goals
goal_state = set(["variables", "loops", "functions", "decorators",
"generators"])

path = means_end_search(resources, start_state, goal_state)

if path:
    print("Recommended Learning Path:")
    for step, res in enumerate(path, 1):
        print(f"{step}. {res.name}")
else:
    print("No learning path found to achieve the goal.")
```

5. Practical-5: • Write a python program to Illustrate Different Set Operations?

• Write a python program to generate Calendar for the given month and year?

• Write a python program to implement Simple Calculator program?

a) Set Operations:

Python

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}

# Union (combines elements from both sets)
union_set = set1 | set2
print("Union:", union_set)

# Intersection (elements present in both sets)
intersection_set = set1 & set2
print("Intersection:", intersection_set)

# Difference (elements in set1 but not set2)
difference_set = set1 - set2
print("Difference:", difference_set)

# Symmetric Difference (elements in either set but not both)
symmetric_difference = set1 ^ set2
print("Symmetric Difference:", symmetric_difference)

# Check if a set is a subset of another (all elements of set1 are in set2)
is_subset = set1.issubset(set2)
print("Is set1 a subset of set2?", is_subset)

# Check if a set is a superset of another (all elements of set2 are in
set1)
is_superset = set1.issuperset(set2)
print("Is set1 a superset of set2?", is_superset)

# Check if sets are disjoint (no common elements)
is_disjoint = set1.isdisjoint(set2)
print("Are set1 and set2 disjoint?", is_disjoint)
```

b) Calendar Generation:

Python

```
import calendar

defgenerate_calendar(month, year):
  """Generates a calendar for the given month and year."""
  print(calendar.monthcalendar(year, month))

month = int(input("Enter month (1-12): "))
year = int(input("Enter year: "))
generate_calendar(month, year)
```

Use code with caution.
content_copy

c) Simple Calculator:

Python

```
def calculate(num1, operator, num2):
  """Performs basic arithmetic operations."""
  if operator == "+":
    return num1 + num2
elif operator == "-":
    return num1 - num2
elif operator == "*":
    return num1 * num2
elif operator == "/":
    if num2 == 0:
      print("Error: Division by zero")
      return None
    else:
      return num1 / num2
  else:
    print("Invalid operator. Please use +, -, *, or /")
    return None

while True:
  num1 = float(input("Enter first number: "))
  operator = input("Enter operator (+, -, *, /): ")
  num2 = float(input("Enter second number: "))
  result = calculate(num1, operator, num2)

  if result is not None:
    print(f"Result: {result}")
  else:
    print("Calculation failed.")

  choice = input("Do you want to continue? (y/n): ").lower()
  if choice != 'y':
    break
```

Use code with caution.
content_copy

6. Practical-6:• Write a python program to Add Two Matrices.

• Write a python program to Transpose a Matrix.

a) Matrix Addition:

Python

```
defadd_matrices(matrix1, matrix2):
  """Adds two matrices with the same dimensions."""
  if len(matrix1) != len(matrix2) or len(matrix1[0]) != len(matrix2[0]):
    print("Error: Matrices must have the same dimensions.")
    return None

  rows, cols = len(matrix1), len(matrix1[0])
  result = [[0 for _ in range(cols)] for _ in range(rows)]
  for i in range(rows):
    for j in range(cols):
      result[i][j] = matrix1[i][j] + matrix2[i][j]
  return result

matrix1 = [[1, 2], [3, 4]]
matrix2 = [[5, 6], [7, 8]]

result = add_matrices(matrix1, matrix2)
if result is not None:
  print("Resultant Matrix:")
  for row in result:
    print(row)
```

Use code with caution.

content_copy

b) Matrix Transpose:

Python

```
deftranspose_matrix(matrix):
  """Transposes a matrix."""
  rows, cols = len(matrix), len(matrix[0])
  transposed = [[0 for _ in range(rows)] for _ in range(cols)]
  for i in range(rows):
    for j in range(cols):
      transposed[j][i] = matrix[i][j]
  return transposed

matrix = [[1, 2, 3], [4, 5, 6]]

transposed = transpose_matrix(matrix)
print("Transposed Matrix:")
for row in transposed:
  print(row)
```

Use code with caution.

content_copy

7. Practical-7: Write a python program to implement Breadth First Search Traversal?

Breadth-First Search (BFS)

Python

```
from collections import deque

defbfs_traversal(graph, start_node):
  """Performs Breadth-First Search traversal on a graph."""
  visited = set()
  queue = deque([start_node])

  while queue:
current_node = queue.popleft()
visited.add(current_node)
    print(current_node, end=" ")

    for neighbor in graph[current_node]:
      if neighbor not in visited:
queue.append(neighbor)

# Example usage (replace with your actual graph structure)
graph = {
  'A': ['B', 'C'],
  'B': ['D', 'E'],
  'C': ['F'],
  'D': [],
  'E': ['F'],
  'F': []
}

bfs_traversal(graph, 'A')
```

Practical-8

Write a python program to implement Water Jug Problem?

Python

```
defwater_jug_problem(jug1_capacity, jug2_capacity, target):
  """Solves the Water Jug Problem using backtracking."""

def backtrack(current_state, visited):
    """Recursive function to explore possible states."""
    if current_state == (target, 0):  # Goal state reached
      return True

    if current_state in visited:  # Avoid revisiting states
      return False

visited.add(current_state)

    # Explore all possible actions from the current state
    for action in ["fill jug1", "fill jug2", "empty jug1", "empty jug2",
"pour jug1 to jug2", "pour jug2 to jug1"]:
new_state = perform_action(current_state, jug1_capacity, jug2_capacity,
action)
      if new_state is not None and backtrack(new_state, visited):
        return True  # Solution found

    return False  # No solution found from this state

  # Function to perform actions on the current state and return the new
state
defperform_action(state, jug1_capacity, jug2_capacity, action):
    jug1, jug2 = state

    if action == "fill jug1":
      return jug1_capacity, jug2
elif action == "fill jug2":
      return jug1, jug2_capacity
elif action == "empty jug1":
      return 0, jug2
elif action == "empty jug2":
      return jug1, 0
elif action == "pour jug1 to jug2":
      amount = min(jug1, jug2_capacity - jug2)
      return jug1 - amount, jug2 + amount
elif action == "pour jug2 to jug1":
      amount = min(jug2, jug1_capacity - jug1)
      return jug1 + amount, jug2 - amount
    else:
      return None  # Invalid action

  # Start backtracking from the initial state (0, 0)
  visited = set()
  return backtrack((0, 0), visited)

# Example usage
jug1_capacity = 5
jug2_capacity = 3
target = 4

if water_jug_problem(jug1_capacity, jug2_capacity, target):
  print("Solution found!")
else:
  print("No solution exists for the given parameters.")
```

Practical-9: Write a program to implement Tic-Tac-Toe game using python.

Tic-Tac-Toe Game

Python

```
defdisplay_board(board):
  """Displays the current state of the Tic-Tac-Toe board."""
  for row in board:
    print("|", end="")
    for cell in row:
      print(cell, end="|")
    print()

defis_winner(board, player):
  """Checks if a player has won."""
win_conditions = ((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 condition in win_conditions:
    if all(board[i] == player for i in condition):
      return True
  return False

defis_board_full(board):
  """Checks if all cells are occupied."""
  return all(cell != ' ' for cell in board)

defget_player_move(board):
  """Gets a valid move from the current player."""
  while True:
    move = int(input("Enter your move (1-9): ")) - 1
    if 0 <= move <= 8 and board[move] == ' ':
      return move
    else:
      print("Invalid move. Please try again.")

defplay_tic_tac_toe():
  """Main function to play the Tic-Tac-Toe game."""
  board = [' '] * 9
current_player = 'X'

  while True:
display_board(board)
    move = get_player_move(board)
    board[move] = current_player

    if is_winner(board, current_player):
display_board(board)
      print(f"Player {current_player} wins!")
      break

    if is_board_full(board):
display_board(board)
      print("It's a tie!")
      break

current_player = 'O' if current_player == 'X' else 'X'

# Start the game
play_tic_tac_toe()
```

Use code with caution.

content_copy

Practical-10: • Write a python program to remove stop words for a given passage from a text file using NLTK?

• Write a python program to implement stemming for a given sentence using NLTK?

• Write a python program to POS (Parts of Speech) tagging for the give sentence using NLTK?

NLTK Text Processing

a) Remove Stop Words:

Python

```
import nltk
from nltk.corpus import stopwords

nltk.download('stopwords')  # Download stopwords corpus (first time only)

defremove_stopwords(text):
  """Removes stop words from a given text."""
stop_words = stopwords.words('english')
  words = [word for word in text.lower().split() if word not in stop_words]
  return ' '.join(words)

# Example usage
text = "This is a sample sentence for stop word removal."
clean_text = remove_stopwords(text)
```

```
print(f"Original text: {text}")
print(f"Clean text (stop words removed): {clean_text}")
```

b) Stemming:

Python

```
import nltk
from nltk.stem import PorterStemmer

nltk.download('punkt')  # Download punkt corpus for sentence tokenization
(first time only)

defstemming_sentence(text):
  """Stems words in a given sentence."""
  stemmer = PorterStemmer()
  words = [stemmer.stem(word) for word in text.lower().split()]
  return ' '.join(words)

# Example usage
sentence = "The cars were running very fast on the highway."
stemmed_sentence = stemming_sentence(sentence)
print(f"Original sentence: {sentence}")
print(f"Stemmed sentence: {stemmed_sentence}")
```

c) POS Tagging:

Python

```
import nltk
from nltk import word_tokenize

nltk.download('punkt')  # Download punkt corpus (first time only)

defpos_tagging(text):
  """Performs Part-of-Speech tagging on a sentence."""
  words = word_tokenize(text)
tagged_words = nltk.pos_tag(words)
  return tagged_words

# Example usage
sentence = "The quick brown fox jumps over the lazy dog."
tagged_words = pos_tagging(sentence)
print(f"Sentence: {sentence}")
print(f"POS Tags: {tagged_words}")
```

Practical-11:• Write a python program to implement Lemmatization using NLTK?

• Write a python program to for Text Classification for the give sentence using NLTK?

NLTK Text Processing

a) Lemmatization:

Python

```
import nltk
```

```
from nltk.stem import WordNetLemmatizer

nltk.download('wordnet')  # Download WordNet corpus (first time only)

deflemmatization_sentence(text):
  """Lemmatizes words in a given sentence."""
lemmatizer = WordNetLemmatizer()
  words = [lemmatizer.lemmatize(word) for word in text.lower().split()]
  return ' '.join(words)

# Example usage
sentence = "The cars were running very fast on the highway."
lemmatized_sentence = lemmatization_sentence(sentence)
print(f"Original sentence: {sentence}")
print(f"Lemmatized sentence: {lemmatized_sentence}")
```

b) Text Classification (Simple Example):

Python

```
import nltk
from nltk.classify import NaiveBayesClassifier

nltk.download('punkt')  # Download punkt corpus (first time only)

# Sample training data with sentiment labels
documents = [
    ("The movie was awesome", "positive"),
    ("The food was awful", "negative"),
    ("This is a great book!", "positive"),
    ("I don't like this movie", "negative"),
    ("This class is very interesting", "positive")
]

# Feature extraction function: bag-of-words
deffeature_extraction(document):
  words = set(document.lower().split())
  features = {}
  for word in words:
    features[word] = True
  return features

# Train the Naive Bayes classifier
features = [feature_extraction(d) for d, _ in documents]
labels = [l for _, l in documents]
classifier = NaiveBayesClassifier.train(features)

# Test the classifier with a new sentence
sentence = "This is a bad movie."
features = feature_extraction(sentence)
predicted_label = classifier.classify(features)
print(f"Sentence: {sentence}")
print(f"Predicted sentiment: {predicted_label}")
```