Tribhuvan University
Faculty of Humanities and Social Sciences

Artificial Intelligence

A LAB REPORT

Submitted to

Department of Computer Application

Shahid Smarak College

In partial fulfillment of the requirements for the Bachelors in Computer Application

Submitted by: -

Amir Maharjan

Internal supervisor	**External Supervisor**
Rajesh Shahi Thakuri	

Table of Contents

Question – 1

Write a python program for the implementation of vacuum cleaner.

Code:

```
import random
class VacuumCleaner:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.environment = [[random.choice(['Clean', 'Dirty']) for _ in range(cols)] for _ in range(rows)]

    def initial_environment( self ):
        print("Current Environment:")
        print("=========================")
        for row in self.environment:
            print( " | ".join( row ) )
        print("=========================")
        print()

    def clean_environment( self ):
        print( "Cleaned Environment:" )
        print("=========================")
        for row in self.get_clean_environment():
            print( " | ".join( row ) )
        print("=========================")
        print()

    def get_clean_environment( self ) :
        row_count = 0
        clean_environment = []
        for row in self.environment:
            column_count = 0
            clean_environment.append( [] )
            for column in row :

                if( column == 'Dirty' ) :
                    column = 'Clean'
                    ( clean_environment[ row_count ] ).append( column )
                else:
                    ( clean_environment[ row_count ] ).append( column )
                column_count = column_count + 1
            row_count = row_count + 1
        return clean_environment

# Initialize and run the vacuum cleaner simulation
if __name__ == "__main__":
    rows = 3
    cols = 3
    vacuum_cleaner = VacuumCleaner(rows, cols)
    vacuum_cleaner.initial_environment()
    vacuum_cleaner.clean_environment()
```

Output:

```
Current Environment:
=========================
Clean | Dirty | Clean
Dirty | Dirty | Dirty
Clean | Clean | Dirty
=========================

Cleaned Environment:
=========================
Clean | Clean | Clean
Clean | Clean | Clean
Clean | Clean | Clean
=========================
```

Question – 2

Write a python program for the implementation of depth first search (DFS).

Code:

```
class Graph:
    def __init__(self):
        self.graph = {}  # Adjacency list representation

    def add_edge(self, vertex, neighbor):
        """Add an edge to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
        self.graph[vertex].append(neighbor)

    def dfs(self, start, visited=None):
        """Recursive implementation of Depth First Search."""
        if visited is None:
            visited = set()

        visited.add(start)
        print(start, end=" ")  # Print the current vertex

        for neighbor in self.graph.get(start, []):
            if neighbor not in visited:
                self.dfs(neighbor, visited)

    def dfs_iterative(self, start):
        """Iterative implementation of Depth First Search."""
        visited = set()
        stack = [start]

        while stack:
            vertex = stack.pop()
            if vertex not in visited:
                print(vertex, end=" ")
                visited.add(vertex)
                # Add neighbors to the stack (in reverse order for correct traversal)
                stack.extend(reversed(self.graph.get(vertex, [])))

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)
    g.add_edge(2, 6)

    print("DFS (Recursive):")
    g.dfs(0)
    print("\nDFS (Iterative):")
    g.dfs_iterative(0)
```

Output:

```
DFS (Recursive):
0 1 3 4 2 5 6
DFS (Iterative):
0 1 3 4 2 5 6
PS D:\College\7th Semester\Artificial Intelligence\Lab Codes>
```

Question – 3

Write a python program for the implementation of breadth first search (BFS).

Code:

```
from collections import deque
class Graph:
    def __init__(self):
        self.graph = {}  # Adjacency list representation

    def add_edge(self, vertex, neighbor):
        """Add an edge to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
        self.graph[vertex].append(neighbor)

    def bfs(self, start):
        """Implementation of Breadth-First Search."""
        visited = set()  # To track visited nodes
        queue = deque([start])  # Queue for BFS (FIFO)
        print("BFS Traversal:", end=" ")

        while queue:
            vertex = queue.popleft()  # Dequeue the front of the queue

            if vertex not in visited:
                print(vertex, end=" ")  # Process the current node
                visited.add(vertex)

                # Enqueue all unvisited neighbors
                for neighbor in self.graph.get(vertex, []):
                    if neighbor not in visited:
                        queue.append(neighbor)

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)
    g.add_edge(2, 6)
    print( "Graph: ", g.graph )
    g.bfs(0)
```

Output:

```
Graph:  {0: [1, 2], 1: [3, 4], 2: [5, 6]}
BFS Traversal: 0 1 2 3 4 5 6
PS D:\College\7th Semester\Artificial Intelligence\Lab Codes>
```

Question – 4

Write a python program for the implementation of uniform cost search.

Code:

```
import heapq
class Graph:
    def __init__(self):
        self.graph = {}  # Adjacency list: {node: [(neighbor, cost), ...]}

    def add_edge(self, vertex, neighbor, cost):
        """Add an edge with a cost to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
        self.graph[vertex].append((neighbor, cost))

        # For undirected graphs, add the reverse edge
        if neighbor not in self.graph:
            self.graph[neighbor] = []
        self.graph[neighbor].append((vertex, cost))

    def uniform_cost_search(self, start, goal):
        """Uniform Cost Search (UCS) implementation."""
        # Priority queue to hold nodes and their cumulative costs
        priority_queue = [(0, start)]  # (cumulative_cost, vertex)
        visited = set()
        while priority_queue:
            cost, current = heapq.heappop(priority_queue)  # Pop the node with the lowest cost

            if current in visited:
                continue

            visited.add(current)
            print(f"Visited Node: {current}, Cost: {cost}")

            # Goal test
            if current == goal:
                print(f"Goal Node {goal} reached with Cost: {cost}")
                return

            # Expand neighbors
            for neighbor, edge_cost in self.graph.get(current, []):
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (cost + edge_cost, neighbor))

        print("Goal not reachable!")
        return

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge("A", "B", 1)
    g.add_edge("A", "C", 4)
    g.add_edge("B", "C", 2)
    g.add_edge("B", "D", 6)
    g.add_edge("C", "D", 3)
    g.add_edge("C", "E", 5)
    g.add_edge("D", "E", 1)
    print("Uniform Cost Search from A to E:")
    g.uniform_cost_search("A", "E")
```

Output:

```
Uniform Cost Search from A to E:
Visited Node: A, Cost: 0
Visited Node: B, Cost: 1
Visited Node: C, Cost: 3
Visited Node: D, Cost: 6
Visited Node: E, Cost: 7
Goal Node E reached with Cost: 7
PS D:\College\7th Semester\Artificial Intelligence\Lab Codes>
```

Question – 5

Write a python program for the implementation of greedy best first search.

Code:

```
import heapq
class Graph:
    def __init__(self):
        self.graph = {}  # Adjacency list: {node: [(neighbor, cost), ...]}
        self.heuristics = {}  # Heuristic values: {node: heuristic_value}

    def add_edge(self, vertex, neighbor, cost):
        """Add an edge with a cost to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
        self.graph[vertex].append((neighbor, cost))

    def set_heuristic(self, node, value):
        """Set heuristic value for a node."""
        self.heuristics[node] = value

    def greedy_best_first_search(self, start, goal):
        """Greedy Best-First Search implementation."""
        priority_queue = [(self.heuristics[start], start)]  # (heuristic_value, vertex)
        visited = set()

        print("Path Traversed:", end=" ")

        while priority_queue:
            h_value, current = heapq.heappop(priority_queue)  # Node with smallest heuristic value

            if current in visited:
                continue

            print(current, end=" -> ")
            visited.add(current)

            # Goal test
            if current == goal:
                print("Goal Reached!")
                return

            # Explore neighbors
            for neighbor, _ in self.graph.get(current, []):
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (self.heuristics[neighbor], neighbor))

        print("Goal not reachable!")
        return
```

```python
# Example usage
if __name__ == "__main__":
    g = Graph()
    # Adding edges to the graph
    g.add_edge("A", "B", 1)
    g.add_edge("A", "C", 4)
    g.add_edge("B", "D", 6)
    g.add_edge("B", "E", 2)
    g.add_edge("C", "F", 5)
    g.add_edge("E", "G", 1)
    g.add_edge("F", "G", 2)

    # Setting heuristic values (lower = better)
    g.set_heuristic("A", 7)
    g.set_heuristic("B", 4)
    g.set_heuristic("C", 6)
    g.set_heuristic("D", 8)
    g.set_heuristic("E", 2)
    g.set_heuristic("F", 3)
    g.set_heuristic("G", 0)

    print("Greedy Best-First Search from A to G:")
    g.greedy_best_first_search("A", "G")
```

Output:

```
Greedy Best-First Search from A to G:
Path Traversed: A -> B -> E -> G -> Goal Reached!
PS D:\College\7th Semester\Artificial Intelligence\Lab Codes>
```

Question – 6

Write a python program for the implementation of A* search.

Code:

```
import heapq
class Graph:
    def __init__(self):
        self.graph = {}  # Adjacency list: {node: [(neighbor, cost), ...]}
        self.heuristics = {}  # Heuristic values: {node: heuristic_value}

    def add_edge(self, vertex, neighbor, cost):
        """Add an edge with a cost to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
        self.graph[vertex].append((neighbor, cost))

    def set_heuristic(self, node, value):
        """Set heuristic value for a node."""
        self.heuristics[node] = value

    def a_star_search(self, start, goal):
        """A* Search implementation."""
        # Priority queue to hold nodes and their f(n) = g(n) + h(n)
        open_list = [(0 + self.heuristics[start], 0, start)]  # (f(n), g(n), node)
        g_costs = {start: 0}  # Dictionary to track g(n) values
        came_from = {}  # To reconstruct the path
        visited = set()

        print("Path Traversed:", end=" ")

        while open_list:
            _, g, current = heapq.heappop(open_list)  # Get the node with lowest f(n)

            if current in visited:
                continue

            print(current, end=" -> ")
            visited.add(current)

            # Goal test
            if current == goal:
                print("Goal Reached!")
                self.reconstruct_path(came_from, goal)
                return

            # Explore neighbors
            for neighbor, cost in self.graph.get(current, []):
                tentative_g = g + cost
                if neighbor not in g_costs or tentative_g < g_costs[neighbor]:
                    g_costs[neighbor] = tentative_g
                    f = tentative_g + self.heuristics.get(neighbor, float('inf'))
                    heapq.heappush(open_list, (f, tentative_g, neighbor))
                    came_from[neighbor] = current

        print("Goal not reachable!")
        return
```

```python
    def reconstruct_path(self, came_from, goal):
        """Reconstruct and print the path from start to goal."""
        path = []
        current = goal
        while current in came_from:
            path.append(current)
            current = came_from[current]
        path.append(current)
        path.reverse()
        print(" -> ".join(path))

# Example usage
if __name__ == "__main__":
    g = Graph()

    # Adding edges (graph structure)
    g.add_edge("A", "B", 1)
    g.add_edge("A", "C", 4)
    g.add_edge("B", "D", 6)
    g.add_edge("B", "E", 2)
    g.add_edge("C", "F", 5)
    g.add_edge("E", "G", 1)
    g.add_edge("F", "G", 2)

    # Setting heuristic values (lower = better)
    g.set_heuristic("A", 7)
    g.set_heuristic("B", 4)
    g.set_heuristic("C", 6)
    g.set_heuristic("D", 8)
    g.set_heuristic("E", 2)
    g.set_heuristic("F", 3)
    g.set_heuristic("G", 0)

    print("A* Search from A to G:")
    g.a_star_search("A", "G")
```

Output:

```
A* Search from A to G:
Path Traversed: A -> B -> E -> G -> Goal Reached!
A -> B -> E -> G
PS D:\College\7th Semester\Artificial Intelligence\Lab Codes>
```

Question – 7

Write a program to implement logic programming using prolog.

Code:

```
dog(rover).
dog(felix).
dog(benny).

animal(A):-dog(A).
```

Output:

```
1 ?- consult("question-1.pl").
true.

2 ?- animal(A).
A = rover ;
A = felix ;
A = benny.

3 ?- dog(rover).
true.

4 ?- dog(cat)
.
false.
```

Question – 8

Write a prolog program to represent few basic facts and perform queries (Elephant is an animal. Elephant is bigger than horse) etc.

Code:

```
animal(elephant).
animal(horse).
animal(dog).
animal(cat).

bigger_than(elephant, horse).
bigger_than(horse, dog).
bigger_than(dog, cat).

is_bigger(X, Y) :- bigger_than(X, Y).
is_bigger(X, Y) :- bigger_than(X, Z), is_bigger(Z, Y).
```

Output:

```
1 ?- consult("question-2.pl").
true.

2 ?- animal(elephant).
true.

3 ?- is_bigger(elephant, dog).
true .

4 ?- is_bigger(dog, elephant).
false.

5 ?- bigger_than(A, B).
A = elephant,
B = horse ;
A = horse,
B = dog ;
A = dog,
B = cat.
```

Question – 9

Write a program in prolog to implement simple arithmetic.

Code:

```
add(X, Y, Result) :- Result is X + Y.
subtract(X, Y, Result) :- Result is X - Y.
multiply(X, Y, Result) :- Result is X * Y.
divide(X, Y, Result) :- Y \= 0, Result is X / Y.
```

Output:

```
1 ?- consult("question-3.pl").
true.

2 ?- add(10, 1, 11).
true.

3 ?- add(10, 2, 20).
false.

4 ?- subtract(1,1, 0).
true.

5 ?- subtract(1,1, 2).
false.

6 ?- multiply(1, 1, 1).
true.

7 ?- multiply(1, 1, 2).
false.

8 ?- divide(1, 1, 1).
true.

9 ?- divide(1, 1, 10).
false.
```

Question – 10

Write a prolog program to convert the sentences into FOPL and to execute queries

Code:

```
loves(likesh, lumanti).
loves(lumanti, likesh).
man(likesh).
woman(lumanti).
likes(om, yomari).
likes(puza, samebaji).
parent(om, sangita).
parent(puza, sangita).
parent(anish, om).

father(X, Y) :- parent(X, Y), man(X).
mother(X, Y) :- parent(X, Y), woman(X).
lovers(X, Y) :- loves(X, Y), loves(Y, X).
```

Output:

```
1 ?- consult("question-4.pl").
true.

2 ?- lovers(likesh, lumanti).
true.

3 ?- mother(Mother, sangita).
false.

4 ?- parent(A, B).
A = om,
B = sangita ;
A = puza,
B = sangita ;
A = anish,
B = om.

5 ?- likes(A, B).
A = om,
B = yomari ;
A = puza,
B = samebaji.
```

Question – 11

Write a prolog program to create a knowledge base of sentences and to execute queries.

Code:

```
male(rajesh).
male(sunil).
male(prakash).
male(ramesh).

female(anita).
female(sita).
female(gita).
female(laxmi).

parent(rajesh, sita).
parent(rajesh, sunil).
parent(gita, sita).
parent(gita, sunil).
parent(sunil, prakash).
parent(laxmi, prakash).

% Rules
father(X, Y) :- parent(X, Y), male(X).
mother(X, Y) :- parent(X, Y), female(X).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Output:

```
1 ?- consult("question-5.pl").
true.

2 ?- male(A).
A = rajesh ;
A = sunil ;
A = prakash ;
A = ramesh.

3 ?- female(A).
A = anita ;
A = sita ;
A = gita ;
A = laxmi.

7 ?- mother(gita, prakash).
false.

8 ?- mother(gita, sunil).
true.
```

```
4 ?- parent(A, B).
A = rajesh,
B = sita ;
A = rajesh,
B = sunil ;
A = gita,
B = sita ;
A = gita,
B = sunil ;
A = sunil,
B = prakash ;
A = laxmi,
B = prakash.

10 ?- sibling(gita, sunil).
false.
```

```
5 ?- father(rajesh, sita).
true.

6 ?- father(rajesh, gita).
false.

11 ?- sibling(X, Y).
X = sita,
Y = sunil ;
X = sunil,
Y = sita ;
X = sita,
Y = sunil ;
X = sunil,
Y = sita ;
false.
```

Question – 12

Write a python program to implement AND, OR gate using perceptron algorithm.

Code:

```
import numpy as np
class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.zeros(input_size + 1)  # Including bias term

    # Step 1: Define the activation function (Step function)
    def activation(self, x):
        return 1 if x >= 0 else 0

    # Step 2: Train the Perceptron
    def train(self, X, y):
        for _ in range(self.epochs):
            for i in range(len(X)):
                # Input with bias term
                inputs = np.append(X[i], 1)
                prediction = self.activation(np.dot(inputs, self.weights))
                # Update weights using the perceptron learning rule
                self.weights += self.learning_rate * (y[i] - prediction) * inputs

    # Step 3: Make predictions
    def predict(self, X):
        predictions = []
        for i in range(len(X)):
            inputs = np.append(X[i], 1)
            prediction = self.activation(np.dot(inputs, self.weights))
            predictions.append(prediction)
        return predictions

# Step 4: Define input and output for AND and OR gates
# AND Gate
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input
y_and = np.array([0, 0, 0, 1])  # Output

# OR Gate
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input
y_or = np.array([0, 1, 1, 1])  # Output

# Step 5: Create perceptron models for AND and OR gates
perceptron_and = Perceptron(input_size=2)
perceptron_or = Perceptron(input_size=2)

# Train the models
perceptron_and.train(X_and, y_and)
perceptron_or.train(X_or, y_or)
```

```python
# Step 6: Test the models
print("Testing AND Gate:")
predictions_and = perceptron_and.predict(X_and)
print(f"Predictions: {predictions_and}")
print(f"Expected: {y_and}\n")

print("Testing OR Gate:")
predictions_or = perceptron_or.predict(X_or)
print(f"Predictions: {predictions_or}")
print(f"Expected: {y_or}")
```

Output:

```
Testing AND Gate:
Predictions: [0, 0, 0, 1]
Expected: [0 0 0 1]

Testing OR Gate:
Predictions: [0, 1, 1, 1]
Expected: [0 1 1 1]
PS D:\College\7th Semester\Artificial Intelligence\Lab Codes>
```

Question – 13

Write a python program to illustrate working of backpropagation algorithm.

Code:

```
import numpy as np
# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Sigmoid derivative function
def sigmoid_derivative(x):
    return x * (1 - x)

# Neural Network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights with random values
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Random weights and biases
        self.weights_input_hidden = np.random.rand(self.input_size, self.hidden_size)
        self.weights_hidden_output = np.random.rand(self.hidden_size, self.output_size)

        self.bias_hidden = np.random.rand(self.hidden_size)
        self.bias_output = np.random.rand(self.output_size)

    def forward(self, X):
        # Forward propagation through the network
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)

        self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        self.final_output = sigmoid(self.final_input)

        return self.final_output

    def backward(self, X, y, output):
        # Backpropagation
        output_error = y - output
        output_delta = output_error * sigmoid_derivative(output)

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

        # Update weights and biases
        self.weights_hidden_output += self.hidden_output.T.dot(output_delta)
        self.weights_input_hidden += X.T.dot(hidden_delta)

        self.bias_output += np.sum(output_delta, axis=0)
        self.bias_hidden += np.sum(hidden_delta, axis=0)
```

```python
def train(self, X, y, epochs):
    for epoch in range(epochs):
      # Perform forward pass
      output = self.forward(X)

      # Perform backward pass (backpropagation)
      self.backward(X, y, output)

      # Optionally, print error at intervals (for tracking learning)
      if epoch % 1000 == 0:
        error = np.mean(np.square(y - output))  # Mean Squared Error
        print(f'Epoch {epoch}, Error: {error}')

  def predict(self, X):
    return self.forward(X)

# Define the XOR problem (training data)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input
y = np.array([[0], [1], [1], [0]])  # Output

# Create a neural network with 2 input neurons, 4 hidden neurons, and 1 output neuron
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)

# Train the neural network
nn.train(X, y, epochs=10000)

# Test the neural network after training
print("\nTesting after training:")
print(nn.predict(X))
```

Output:

```
Epoch 0, Error: 0.39541121161369497
Epoch 1000, Error: 0.004090642495651932
Epoch 2000, Error: 0.0009213306018762382
Epoch 3000, Error: 0.0005042071744035696
Epoch 4000, Error: 0.0003442509305999132
Epoch 5000, Error: 0.0002603486273090009
Epoch 6000, Error: 0.0002088730450990481
Epoch 7000, Error: 0.00017414682216605005
Epoch 8000, Error: 0.0001491747427014817
Epoch 9000, Error: 0.00013037205228140204

Testing after training:
[[0.00626674]
 [0.98947872]
 [0.9887542 ]
 [0.01365362]]
PS D:\College\7th Semester\Artificial Intelligence\Lab Codes>
```