Amrit Science Campus

Lainchur, Kathmandu

Lab Manual Of Artificial Intelligence

Course Code: (CSC-261)

Semester: Fourth

Lecturer:- Abhimanu Yadav

Lab Manual

Lab: "ARTIFICIAL INTELLIGENCE LAB USING PYTHON"

List of Experiments:

1. Write a Program to Implement Breadth First Search using Python.
2. Write a Program to Implement Depth First Search using Python.
3. Write a Program to Implement Tic-Tac-Toe game using Python.
4. Write a Program to Implement Water-Jug problem using Python.
5. Write a Program to Implement Travelling Salesman Problem using Python.
6. Write a Program to Implement Tower of Hanoi using Python.
7. Write a Program to Implement Monkey Banana Problem using Python.
8. Write a Program to Implement N-Queens Problem using Python.
9. Write a Program to Implement Naïve Bayes Algorithm using Python.
10. Write a Program to Implement Back propagation Algorithm using Python
11. Write a Program to Implement Genetics algorithm using Python.
12. Write a program to implement A* Search Algorithm.
13. Write a program to implement greedy search algorithm.
14. Write a program to implement the uniform cost search Algorithm.

Solution:

1. *Write a Program to Implement Breadth First Search using Python.*

Source Code:

```
# Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict
# This class represents a directed graph
# using adjacency list representation
class Graph:
# Constructor
def __init__(self):
# default dictionary to store graph
self.graph = defaultdict(list)
# function to add an edge to graph
def addEdge(self,u,v):
self.graph[u].append(v)
# Function to print a BFS of graph
def BFS(self, s):
# Mark all the vertices as not visited
visited = [False] * (max(self.graph) + 1)
# Create a queue for BFS
queue = []
# Mark the source node as
# visited and enqueue it
```

```
queue.append(s)
visited[s] = True
while queue:
# Dequeue a vertex from
# queue and print it
s = queue.pop(0)
print (s, end = " ")
# Get all adjacent vertices of the
# dequeued vertex s. If a adjacent
# has not been visited, then mark it
# visited and enqueue it
for i in self.graph[s]:
if visited[i] == False:
queue.append(i)
visited[i] = True
# Driver code
# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
```

```
print ("Following is Breadth First Traversal"
" (starting from vertex 2)")
g.BFS(2)
```

Output:

```
Following is Breadth First Traversal (starting from vertex 2)
> 3
2 0 3 1 3
>
```

2. *Write a Program to Implement Depth First Search using Python.*

Source Code:

```
# program to print DFS traversal
# from a given given graph
```

```
from collections import defaultdict
# This class represents a directed graph using
# adjacency list representation
class Graph:
# Constructor
def __init__(self):
# default dictionary to store graph
self.graph = defaultdict(list)
# function to add an edge to graph
def addEdge(self, u, v):
self.graph[u].append(v)
# A function used by DFS
def DFSUtil(self, v, visited):
# Mark the current node as visited
# and print it
visited.add(v)
print(v, end=' ')
# Recur for all the vertices
# adjacent to this vertex
for neighbour in self.graph[v]:
if neighbour not in visited:
self.DFSUtil(neighbour, visited)
# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self, v):
```

```
# Create a set to store visited vertices
visited = set()
# Call the recursive helper function
# to print DFS traversal
self.DFSUtil(v, visited)
# Driver code
# Create a graph given
# in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

Output:

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 9 3
```

3. Write a Program to Implement Tic-Tac-Toe game using Python.

Source Code:

```
# Tic-Tac-Toe Program using
# random number in Python
# importing all necessary libraries
import numpy as np
import random
from time import sleep
# Creates an empty board
def create_board():
return(np.array([[0, 0, 0],
[0, 0, 0],
[0, 0, 0]]))
# Check for empty places on board
def possibilities(board):
l = []
for i in range(len(board)):
for j in range(len(board)):
if board[i][j] == 0:
l.append((i, j))
return(l)
# Select a random place for the player
def random_place(board, player):
selection = possibilities(board)
current_loc = random.choice(selection)
board[current_loc] = player
return(board)
# Checks whether the player has three
# of their marks in a horizontal row
```

```
def row_win(board, player):
for x in range(len(board)):
win = True
for y in range(len(board)):
if board[x, y] != player:
win = False
continue
if win == True:
return(win)
return(win)
# Checks whether the player has three
# of their marks in a vertical row
def col_win(board, player):
for x in range(len(board)):
win = True
for y in range(len(board)):
if board[y][x] != player:
win = False
continue
if win == True:
return(win)
return(win)
# Checks whether the player has three
# of their marks in a diagonal row
def diag_win(board, player):
```

```
win = True
y = 0
for x in range(len(board)):
if board[x, x] != player:
win = False
if win:
return win
win = True
if win:
for x in range(len(board)):
y = len(board) - 1 - x
if board[x, y] != player:
win = False
return win
# Evaluates whether there is
# a winner or a tie
def evaluate(board):
winner = 0
for player in [1, 2]:
if (row_win(board, player) or
col_win(board,player) or
diag_win(board,player)):
winner = player
if np.all(board != 0) and winner == 0:
winner = -1
```

```
return winner
# Main function to start the game
def play_game():
board, winner, counter = create_board(), 0, 1
print(board)
sleep(2)
while winner == 0:
for player in [1, 2]:
board = random_place(board, player)
print("Board after " + str(counter) + " move")
print(board)
sleep(2)
counter += 1
winner = evaluate(board)
if winner != 0:
break
return(winner)
# Driver Code
print("Winner is: " + str(play_game()))
```

Output:

```
[[0 0 0]
[0 0 0]
[0 0 0]]
Board after 1 move
[[0 0 0]
```

```
[0 0 0]
[1 0 0]]
Board after 2 move
[[0 0 0]
[0 2 0]
[1 0 0]]
Board after 3 move
[[0 1 0]
[0 2 0]
[1 0 0]]
Board after 4 move
[[0 1 0]
[2 2 0]
[1 0 0]]
Board after 5 move
[[1 1 0]
[2 2 0]
[1 0 0]]
Board after 6 move
[[1 1 0]
[2 2 0]
[1 2 0]]
Board after 7 move
[[1 1 0]
[2 2 0]
```

```
[1 2 1]]
Board after 8 move
[[1 1 0]
[2 2 2]
[1 2 1]]
Winner is: 2
```

4. Write a Program to Implement Water-Jug problem using Python.

Source Code:

```
# This function is used to initialize the
# dictionary elements with a default value.
from collections import defaultdict
# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
```

```
jug1, jug2, aim = 4, 3, 2
# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)
# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):
# Checks for our goal and
# returns true if achieved.
if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
print(amt1, amt2)
return True
# Checks if we have already visited the
# combination or not. If not, then it proceeds further.
if visited[(amt1, amt2)] == False:
print(amt1, amt2)
# Changes the boolean value of
# the combination as it is visited.
visited[(amt1, amt2)] = True
# Check for all the 6 possibilities and
# see if a solution is found in any one of them.
```

```
return (waterJugSolver(0, amt2) or
waterJugSolver(amt1, 0) or
waterJugSolver(jug1, amt2) or
waterJugSolver(amt1, jug2) or
waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
amt2 - min(amt2, (jug1-amt1))) or
waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
amt2 + min(amt1, (jug2-amt2))))
# Return False if the combination is
# already visited to avoid repetition otherwise
# recursion will enter an infinite loop.
else:
return False
print("Steps: ")
# Call the function and pass the
# initial amount of water present in both jugs.
waterJugSolver(0, 0)
```

Output:

```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
```

```
4 2
0 2
```

5. Write a Program to Implement Travelling Salesman Problem using Python.

Source Code:

```
# program to implement traveling salesman
# problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4
# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):
# store all vertex apart from source vertex
vertex = []
for i in range(V):
```

```
if i != s:
vertex.append(i)
# store minimum weight
min_path = maxsize
next_permutation=permutations(vertex)
for i in next_permutation:
# store current Path weight(cost)
current_pathweight = 0
# compute current path weight
k = s
for j in i:
current_pathweight += graph[k][j]
k = j
current_pathweight += graph[k][s]
# update minimum
min_path = min(min_path, current_pathweight)
return min_path
# Driver Code
if __name__ == "__main__":
# matrix representation of graph
graph = [[0, 10, 15, 20], [10, 0, 35, 25],
[15, 35, 0, 30], [20, 25, 30, 0]]
s = 0
print(travellingSalesmanProblem(graph, s))
```

Output

6. Write a Program to Implement Tower of Hanoi using Python.

Source Code:

```
# Recursive Python function to solve tower of hanoi
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
if n == 1:
print("Move disk 1 from rod",from_rod,"to rod",to_rod)
return
TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
print("Move disk",n,"from rod",from_rod,"to rod",to_rod)
TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)
# Driver code
n = 4
TowerOfHanoi(n, 'A', 'C', 'B')
# A, C, B are the name of rods
```

Output

```
 Move disk 1 from rod A to rod B
```

```
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

Output:

```
Tower of Hanoi Solution for 4 disks:
A: [4, 3, 2, 1] B: [] C: []
Move disk from rod A to rod B
A: [4, 3, 2] B: [1] C: []
Move disk from rod A to rod C
A: [4, 3] B: [1] C: [2]
Move disk from rod B to rod C
A: [4, 3] B: [] C: [2, 1]
Move disk from rod A to rod B
A: [4] B: [3] C: [2, 1]
```

Move disk from rod C to rod A

A: [4, 1] B: [3] C: [2]

Move disk from rod C to rod B

A: [4, 1] B: [3, 2] C: []

Move disk from rod A to rod B

A: [4] B: [3, 2, 1] C: []

Move disk from rod A to rod C

A: [] B: [3, 2, 1] C: [4]

Move disk from rod B to rod C

A: [] B: [3, 2] C: [4, 1]

Move disk from rod B to rod A

A: [2] B: [3] C: [4, 1]

Move disk from rod C to rod A

A: [2, 1] B: [3] C: [4]

Move disk from rod B to rod C

A: [2, 1] B: [] C: [4, 3]

Move disk from rod A to rod B

A: [2] B: [1] C: [4, 3]

Move disk from rod A to rod C

A: [] B: [1] C: [4, 3, 2]

Move disk from rod B to rod C

A: [] B: [] C: [4, 3, 2, 1]

7. Write a Program to Implement the Monkey Banana Problem using Python.

Source Code:

```
from poodle import Object, schedule
from typing import Set
class Position(Object):
def __str__(self):
if not hasattr(self, "locname"): return "unknown"
return self.locname
class HasHeight(Object):
height: int
class HasPosition(Object):
at: Position
class Monkey(HasHeight, HasPosition): pass
class PalmTree(HasHeight, HasPosition):
def __init__(self, *args, **kwargs):
super().__init__(*args, **kwargs)
self.height = 2
class Box(HasHeight, HasPosition): pass
class Banana(HasHeight, HasPosition):
owner: Monkey
```

```
attached: PalmTree
class World(Object):
locations: Set[Position]
p1 = Position()
p1.locname = "Position A"
p2 = Position()
p2.locname = "Position B"
p3 = Position()
p3.locname = "Position C"
w = World()
w.locations.add(p1)
w.locations.add(p2)
w.locations.add(p3)
m = Monkey()
m.height = 0 # ground
m.at = p1
box = Box()
box.height = 2
box.at = p2
p = PalmTree()
p.at = p3
b = Banana()
b.attached = p
def go(monkey: Monkey, where: Position):
assert where in w.locations
assert monkey.height < 1, "Monkey can only move while on the ground"
monkey.at = where
return f"Monkey moved to {where}"
def push(monkey: Monkey, box: Box, where: Position):
assert monkey.at == box.at
assert where in w.locations
assert monkey.height < 1, "Monkey can only move the box while on the ground"
monkey.at = where
box.at = where
return f"Monkey moved box to {where}"
def climb_up(monkey: Monkey, box: Box):
assert monkey.at == box.at
monkey.height += box.height
return "Monkey climbs the box"
def grasp(monkey: Monkey, banana: Banana):
assert monkey.height == banana.height
assert monkey.at == banana.at
banana.owner = monkey
return "Monkey takes the banana"
def infer_owner_at(palmtree: PalmTree, banana: Banana):
assert banana.attached == palmtree
banana.at = palmtree.at
return "Remembered that if banana is on palm tree, its location is where palm tree is"
def infer_banana_height(palmtree: PalmTree, banana: Banana):
assert banana.attached == palmtree
```

```
banana.height = palmtree.height
return "Remembered that if banana is on the tree, its height equals tree's height"
print('\n'.join(x() for x in schedule(
[go, push, climb_up, grasp, infer_banana_height, infer_owner_at],
[w,p1,p2,p3,m,box,p,b],
goal=lambda: b.owner == m)))
```

Result:

```
$ pip install poodle
$ python ./monkey.py
Monkey moved to Position B
Remembered that if banana is on the tree, its height equals tree's height
Remembered that if banana is on palm tree, its location is where palm tree is
Monkey moved box to Position C
Monkey climbs the box
Monkey takes the banana
```

8. Write a Program to Implement the N-Queens Problem using Python.

Source Code:

```
# Python program to solve N Queen
# Problem using backtracking
global N
N = 4
def printSolution(board):
for i in range(N):
for j in range(N):
print board[i][j],
print
# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):
# Check this row on left side
for i in range(col):
```

```
if board[row][i] == 1:
return False
# Check upper diagonal on left side
for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
if board[i][j] == 1:
return False
# Check lower diagonal on left side
for i, j in zip(range(row, N, 1), range(col, -1, -1)):
if board[i][j] == 1:
return False
return True
def solveNQUtil(board, col):
# base case: If all queens are placed
# then return true
if col >= N:
return True
# Consider this column and try placing
# this queen in all rows one by one
for i in range(N):
if isSafe(board, i, col):
# Place this queen in board[i][col]
board[i][col] = 1
# recur to place rest of the queens
if solveNQUtil(board, col + 1) == True:
return True
```

```
# If placing queen in board[i][col
# doesn't lead to a solution, then
# queen from board[i][col]
board[i][col] = 0
# if the queen can not be placed in any row in
# this colum col then return false
return False
# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
board = [ [0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]
]
if solveNQUtil(board, 0) == False:
print "Solution does not exist"
return False
printSolution(board)
```

```
return True
# driver program to test above function
solveNQ()
```

Output:

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

THE END

Lab: “ARTIFICIAL INTELLIGENCE LAB USING Prolog