Pabna University of Science and Technology

Faculty of Engineering and Technology

Department of

Information and Communication Engineering

Lab Report

Course Code: ICE-3102

Course Title: Artificial Intelligence and Robotics Sessional

Submitted By,

Md. Ibne Shihab Shad

Roll: **200618**

Registration: **1065383**

Session: 2019-2020

3rd Year 1st Semester

Dept. of Information and Communications Engineering, PUST

Submitted To,

Dr. Md. Omar Faruk

Associate Professor,

and

Tarun Debnath

Assistant Professor,

Department of Information and Communication Engineering, Pabna University of Science and Technology.

Date of Submission: 30/ 10/ 2023

Index

Serial No.	Experiment Name
01	Write a Program to Implement Tower of Hanoi for n Disk.
02	Write a Program to Implement Breadth First Search Algorithm.
03	Write a Program to Implement Depth First Search Algorithm.
04	Write a Program to Implement Travelling Salesman Problem.
05	(i) Create and Load Different Datasets in Python. (ii) Write a Python Program to Compute Mean, Median, Mode, Variance and Standard Deviation using Datasets.
06	Write a Python Program to Implement Simple Linear Regression and Plot the Graph.
07	Write a Python Program to Implement Find S Algorithm.
08	Write a Python Program to Implement Support Vector Machine (SVM) Algorithm.

Experiment Number: 01

Name of the Experiment: To write a program to implement Tower of Hanoi for n disk

Objectives

1) To learn about the mathematical puzzel of Tower of Hanoi.
2) To learn how a recursive function works.

Theory:

The tower of Hanoi is a classic puzzle that involve three rods and stack of disks of varying sizes The objective is to move the entire stack from one rod to another where following the specific rules:

i) Only one disk can be moved at a time.
ii) Each move consists of taking the upper disk from one of the stacks and placing it on the top of an other stack or an empty rod.
iii) No disk may be placed on top of a smaller stack.

It typically solved using recursive algorithms, while the number of moves required being $2^N - 1$ for N disks. The puzzel serves as an educational tool to teach recursion and problem solving in computer science and mathematics.

Results and Discussions:

From the code and theory we have seen that at each stage only one disk is moved from one tower to another towers. And as the second condition we haven't put a smaller disk under a larger disk and that's how we have generate our code in the code there are total six steps for 3 disks in the three used towers. and that's how we have solved the experiment and the experiment was successfull because of maintaining the given steps properly.

CODE:

```
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n-1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n-1, auxiliary, source, target)
num_disks = int(input("Enter the number of disks: "))

tower_of_hanoi(num_disks, 'A', 'B', 'C')
```

INPUT:

Enter the number of disks: 3

OUTPUT:

Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Experiment Number: 02

Name of the Experiment: To write a program to implement Breadth-first search Algorithm

Objectives:

1) To know about Breadth-first Search (BFS) algorithm
2) Apply and implement Breadth-First Search (BFS) algorithm.
3) Knowing the complexity (Time and Space) about the BFS algorithm.

Theory:

Breadth First Search is a graph traversal algorithm that systematically explores the structure of a graph or tree in a level wise manner. It starts at a specific node, known as the source or root and proceeds to visit all nodes at the current depth level before moving on to nodes at the next level. key aspects of BFS includes its use of Queue data structure, the ability to find the shortest path in unweighted graph.

This graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then it selects the nearest node and explore all the unexplored nodes. While using BFS algorithm any node in the graph can be considered as the root node.

Algorithm of BFS algorithm

Step 01: SET STATUS = 1 (ready state) for each node in the graph.

Step 02: Enqueue the starting node A and set its STATUS = 2 (waiting state)

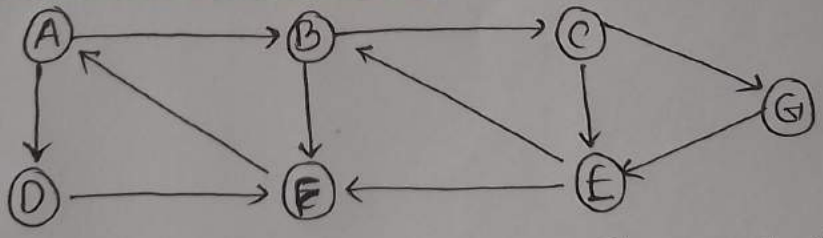
Step 03: Repeat 4 and 5 untill Queue is empty

Step 04: Dequeue a node N. Process it and set it STATUS = 3 (processed state)

Step 05: Enqueue all the neighbours of N that are in the ready state and set their STATUS = 2 [End of Loop]

Step 06: Exit.

Example of BFS Algorithm



Step 01: Queue 1 = {A}
Queue 2 = {Null}

Step 02: Queue 1 = {B, D}
Queue 2 = {A}

Step 03: Queue 1 = {D, C, F}
Queue 2 = {A, B}

Step 04: Queue 1 = {C, F}
Queue 2 = {A, B, D}

Step 05: Queue 1 = {F, E, G}
Queue 2 = {AB, D, C}

Step 06: Queue 1 = {E, G}
Queue 2 = {A, B, D, C, F}

Step 07: Queue 1 = {G}
Queue 2 = {A, B, C, D, E, F}

Code:

```
from collections import deque
def bfs(graph, start_node):
    visited = set()
    queue = deque()

    visited.add(start_node)
    queue.append(start_node)

    while queue:
        current_node = queue.popleft()
        print(current_node, end=' ')

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

graph = {
    0: [1, 3],
    1: [2, 5],
    2: [4, 6],
    3: [5],
    4: [1,5],
    5: [0],
    6: [4]
}
print("Breadth First Traversal (starting from node 6):", end=' ')
bfs(graph, 0)
```

Output:

Breadth First Traversal (starting from node 6): 6 4 1 5 2 0 3

Results and Disscussion:

In the Breadth first Algorithm we have to Traverse an unweighted graph. We can start with any node of the graph and by traversing the adjacent nodes we complete graph traversal. The breadth first method/algorithm uses the Queue conditions. By following the algorithm we have generated a code and the code satisfy the Breadth First Search conditions and traverses all the nodes properly so we can say that we have done our experiment properly as it generates appropriate result.

Experiment Number: 03

Name of The Experiment: To write a program to implement Depth-First Search

Objectives:

1) To get proper knowledge about Depth-Fist Search
2) Finding solution using Depth-First Search algorithm
3) knowing Time and space complexity about the depth first Search.

Theory:

Depth first Search (DFS) is an algorithm for traversing or searching tree or graph data structures It starts at a specified node and explores as far as possible along each branch before backtracking. systematically following paths to their deepest levels. The step by step process to implement the DFS traversal is given as follows

① First we have to create a stack with total number of vertices in the graph.

② Now have to choose any vertex as the starting point of traversal and push that vertex into the stack

③ After that push an adjacent vertex of the first vertex into the stack.

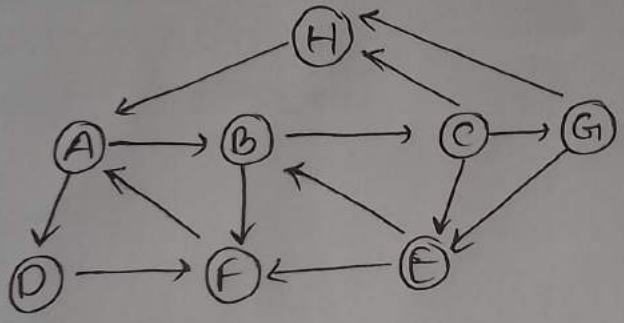
④ Now repeat same thing until no vertices are left to visit.

⑤ If no vertex left then go and pop a vertex from the stack

Algorithm

step 01: Start
step 02: Set STATUS=1 for each node in Graph
step 03: Repeat 4 and 5 until stack is emptly
step 04: Pop the top node N. Process it and set STATUS=3
step 05: Push on the stack all the neighbors of N that are in the ready state and set their STATUS=2
step 06: Exit

Example



Adjacency List

A: B, D
B: C, F
C: E, G, H
G: E, H
D: F
E: B, F
F: A
H: A

step 01: stack: H

step 02: print: H
stack: A

step 03: print: A
stack: B, D

step 04: print: B
stack: D, F, C

step 05: print: D
stack: F, C

step 06: print: F
stack: C

step 07: print: C
stack: H, G, E

step 08: print: H
stack: G, E

step 09: print: G
stack: E

step 10: print: E

CODE:

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, node, visited):
        visited[node] = True
        print(node, end=' ')

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

graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)
visited = [False] * len(graph.graph)
print("Depth-First Search:")
graph.dfs(2, visited)
```

OUTPUT:

```
Depth-First Search:
2 0 1 3
```

Results and Discussions:

The Depth-first Search algorithm first visits the adjacent node of the previous nodes then after completing traversing adjacent nodes it uses the back tracking formula to check that if any node remains untraversed. Using the algorithm we have generated a code that satisfies the DFS algorithm and produces an appropriate result and the result matches with the example and no nodes remains untraversed so we can say that the experiment went successfull.

Experiment Number: 04

Name of the Experiment: To write a program to implement Travelling Selsman problem.

Objective:

① To find the shortest possible route
② To understand travelling salesman problem
③ To minimize the total cost of traveling

Theory:

Travelling salesman Problem is like a puzzle where a sales person wants to visit a bunch of cities and returns home while travelling the shortest distance possible. The goal is to figure out the best order to visit these cities to minimizes the total travel distance. It seems to be similar like hamiltonian cycle but there is some difference between hamiltonian cycle and travelling salesman problem. The hamiltonian cycle is to find if there exists a tour that visit every city exactly once but travelling salesman is like visiting every city with minimum cost. The time complexity of Travelling salesman problem is $O(n^2 * 2^n)$ and the space complexity is $O(n * 2^n)$

CODE:

```
import itertools
def calculate_total_distance(tour, distances):
    total_distance = 0
    for i in range(len(tour) - 1):
        total_distance += distances[tour[i]][tour[i + 1]]
    total_distance += distances[tour[-1]][tour[0]]  # Return to the starting city
    return total_distance
num_cities = int(input("Enter the number of cities: "))
distances = []
print("Enter distances between cities (separate values by space):")
for i in range(num_cities):
    row = [int(x) for x in input().split()]
    distances.append(row)

def traveling_salesman_bruteforce(distances):
    num_cities = len(distances)
    cities = list(range(num_cities))
    shortest_tour = None
    shortest_distance = float('inf')

    for tour in itertools.permutations(cities):
        tour_distance = calculate_total_distance(tour, distances)
        if tour_distance < shortest_distance:
            shortest_distance = tour_distance
            shortest_tour = tour

    return shortest_tour, shortest_distance
shortest_tour, shortest_distance = traveling_salesman_bruteforce(distances)
print("Shortest Tour:", shortest_tour)
print("Shortest Distance:", shortest_distance)
```

OUTPUT:

Enter the number of cities: 3

Enter distances between cities (separate values by space):

1 2 3

4 5 6

7 8 9

Shortest Tour: (0, 1, 2)

Shortest Distance: 15

Results and Discussions

In travelling salesman problem the salesman visits all the cities without visiting a city twice and travelling minimum distance. The code follows the travelling salesman algorithm and generates appropriate results. In the code there are 3 cities and there is a distance matrix which indicates the distance between each city. Here the shortest distance is 15.

Experiment Number: 05

Name of the Experiment: i) To create and load different datasets in Python ii) To write a python program to compute Mean, Median, Mode, Variance and Standard Deviation using Data sets.

Objectives:

① Creating a data set a load that dataset using python

② Finding the mean, median, mode, variance and standard deviation using python

Theory:

A dataset is a structured collection of data, typically organized in a way that makes it easy to manage, analyze and retrieve information. It consists of individual data points, which are organized into rows and columns. Data set are used for various perpose, such as research, analysis, machine learning and data driven-decision making.

Mean: The mean is the average of the set of numbers.

Median: Median is the middle ~~n~~number when the numbers are listed in order. If there is an even number of numbers it is the average of the two middle number.

Mode: The mode is the number that appears most frequently in a set of numbers. It is the one that shows up the most.

Variance: Variance measures how spread out or different the numbers are in a dataset. High variance means the numbers are far from the mean, while low variance means they are closed to the mean.

Standard Deviation: Standard Deviation is an other way to measure how spread out the numbers are. It is the square root of variance. A high standard deviation means the numbers are spread out and how a low standard deviation means they are closed to the mean.

CODE:

```
import statistics
dataset = [10, 20, 30, 40, 50]
mean = statistics.mean(dataset)
median = statistics.median(dataset)
mode_value = statistics.mode(dataset)
variance = statistics.variance(dataset)
std_dev = statistics.stdev(dataset)
print("Dataset Statistics:")
print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode_value}")
print(f"Variance: {variance}")
print(f"Standard Deviation: {std_dev}")
```

OUTPUT:

```
Dataset Statistics:
Mean: 30
Median: 30
Mode: 10
Variance: 250
Standard Deviation: 15.811388300841896
```

Experiment Number: 06

Name of the Experiment: To write a program to implement Simple linear Regression and Plot the graph.

Objectives:

① To know about the negration line and its concept.

② Ploting point on graph and analysis negration line

Theory:

Simple Linear Regression is a statistical method used to find straight line that best represents the relationship between two variables.

1. Dependent Variable (Y): This is the variable that we want to predict on understand. It is the output on response.

2. Independent Variable (X): This is the variable we use to make predictions on understand its impact on y. It is the input on predicton

The goal of simple linear negression is to find the equation of a line ($Y = b_0 + b_1 * X$) that best fits the data. This line has

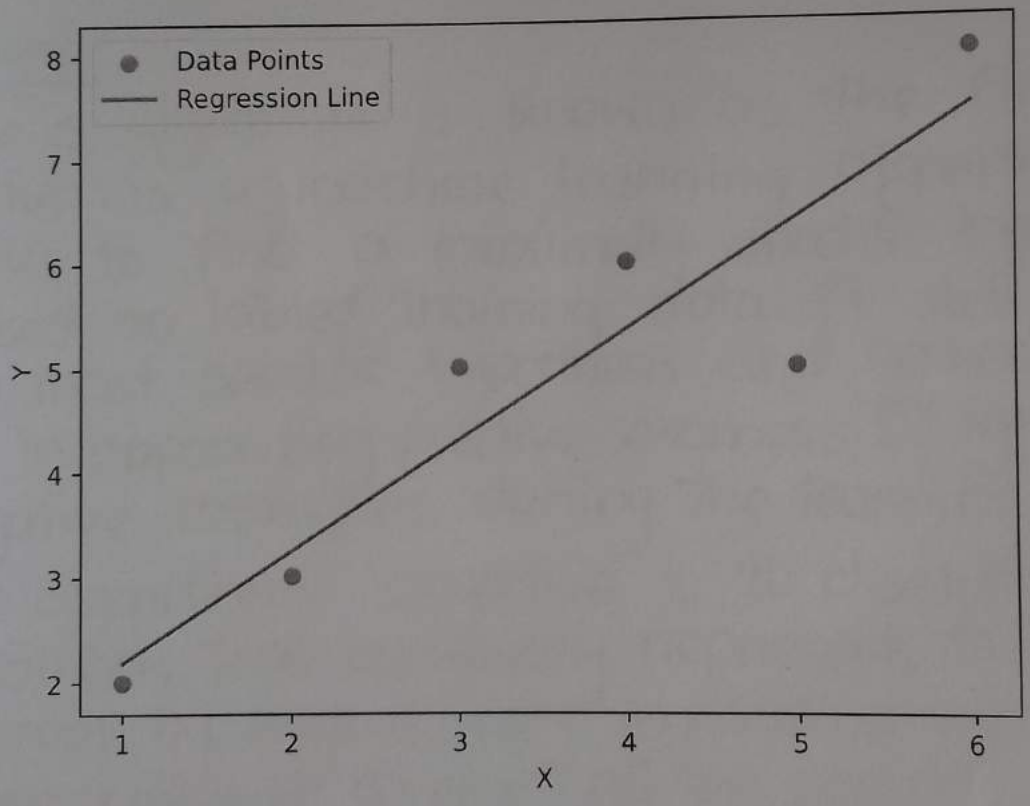
b_0 = The y-intercept, where the line crosses the x-axis

b_1 = The slope of the line, showing how y charges when x charges by one unit.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([1, 2, 3, 4, 5, 6])
y = np.array([2, 3, 5, 6, 5, 8])
slope, intercept = np.polyfit(x, y, 1)
y_pred = slope * x + intercept
plt.scatter(x, y, label='Data Points')
plt.plot(x, y_pred, color='red', label='Regression Line')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

OUTPUT:



Experiment Number: 07

Name of the Experiment: To write a program to implement Find S angorithm.

Objectives:

① Getting knowledge about the Find S algorithm

② Apply and implement S algorithm in the field of Machine learning

Theory:

The S algorithm is known as the Find S algorithm, is a machine learning algorithm that seeks to find a maximally specific hypothesis based on labled training data. It stants with the most specific hypothesis and generalizes it by incorporating positive examples. It ignores negative examples during the learning process. The algorithm's objective is to discover a hypothesis that accurately represents the target concept by possessively expanding the hypothesis space until it covers all the positive instances. There are some symbols used in the Find S Algorithm

Negative Examples (-)

Hypothesis

$\emptyset$ (Empty Set)

? (Don't care)

positive Examples (+)

CODE:

```
# the hypothesis with the most specific values
hypothesis = None

def is_consistent(example, hypothesis):
    for i in range(len(hypothesis)):
        if hypothesis[i] != '?' and hypothesis[i] != example[i]:
            return False
    return True

def find_s(training_data):
    global hypothesis
    for example in training_data:
        x, y = example[:-1], example[-1]
        # If it's a positive example, update the hypothesis
        if y == 'Yes':
            if hypothesis is None:
                hypothesis = list(x)
            else:
                for i in range(len(hypothesis)):
                    if x[i] != hypothesis[i]:
                        hypothesis[i] = '?'

training_data = [
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Yes'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Yes'],
    ['Cloudy', 'Cold', 'High', 'Weak', 'No'],
    ['Rainy', 'Cold', 'High', 'Weak', 'No']
]

find_s(training_data)

print("Final Hypothesis:", hypothesis)
```

OUTPUT:

```
Final Hypothesis: ['Sunny', 'Warm', '?', 'Strong']
```

Experiment Number: 08

Name of the Experiment: To write a python program to implement Support Vector Machine (SVM) Algorithm

objectives:

① To apply and implement support vector machine (SVM) algorithm

② To implement machine learning using SVM

Theory:

The Support Vector Machine is a powerful machine learning algorithm used for classification and regressions tasks. SVM finds hyperplane that best separates data point into distinct class maximizes classes between classes. It is efective in high dimensional space and can handle both linear and non-linear data through kernel functions. SVM is known for its ability to handle outliers and generalize well. It's widely used in various applications, such as text classification. SVM aims to find the optimal decision boundary that maximizes classification errors.

CODE and OUTPUT:

jupyter SMV AL Last Checkpoint: 8 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Help

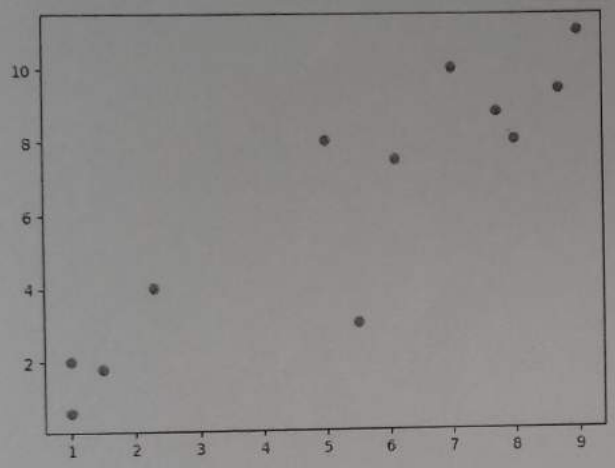
Trusted | Python 3 (ipykernel)

Run Code

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import svm
```

In [3]:
```
# linear data
X = np.array([1, 5, 1.5, 8, 1, 9, 7, 8.7, 2.3, 5.5, 7.7, 6.1])
y = np.array([2, 8, 1.8, 8, 0.6, 11, 10, 9.4, 4, 3, 8.8, 7.5])
```

In [4]:
```
# show unclassified data
plt.scatter(X, y)
plt.show()
```



In [5]:
```
# shaping data for training the model
training_X = np.vstack((X, y)).T
training_y = [0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1]
```

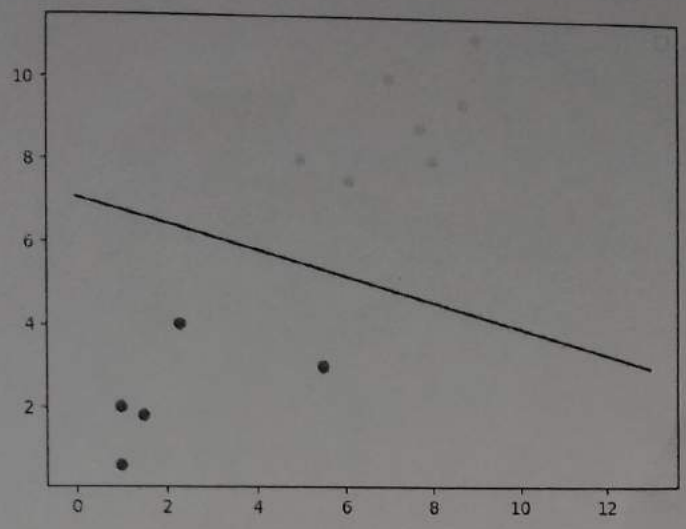
In [6]:
```
# define the model
clf = svm.SVC(kernel='linear', C=1.0)
```

In [7]:
```
# train the model
clf.fit(training_X, training_y)
```

Out[7]:
```
SVC(kernel='linear')
```

In [8]:
```
# get the weight values for the linear equation from the trained SVM model
w = clf.coef_[0]

# get the y-offset for the linear equation
a = -w[0] / w[1]

# make the x-axis space for the data points
XX = np.linspace(0, 13)

# get the y-values to plot the decision boundary
yy = a * XX - clf.intercept_[0] / w[1]

# plot the decision boundary
plt.plot(XX, yy, 'k-')

# show the plot visually
plt.scatter(training_X[:, 0], training_X[:, 1], c=training_y)
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend
() is called with no argument.
```



In [ ]: