Practical Lab Manual

Artificial Intelligence

(310253)

For Third Year Computer Engineering (2019 Course)

Student Name: ____

Seat No / Roll No: ____________________Class: ____________________

Branch: ____

MMANTC
ASPIRE | INSPIRE

DEPARTMENT OF COMPUTER ENGINEERING

Al Jamia Mohammadiyah Education Society's

MAULANA MUKHTAR AHMAD NADVI TECHNICAL CAMPUS

Mansoora Campus, Malegaon (Nashik)

Certificate

This is to certify that…………………………………………………………………………...

Roll No: ………Exam Seat No:………………. Class: …………………………………….

Branch...has Successfully Completed

Artificial Intelligence Lab Experiments and Assignments as per the term work of Savitri-

bai Phule Pune University, Pune.

Subject Teacher **Head of Department** **Principal**

MAULANA MUKHTAR AHMAD NADVI TECHNICAL CAMPUS

DEPARTMENT OF COMPUTER ENGINEERING

Vision of the institute

Empowering society through quality education and research for the socio-economic development of the region.

Mission of the institute

- Inspire students to achieve excellence in science and engineering.
- Commit to making quality education accessible and affordable to serve society.
- Provide transformative, holistic, and value-based immersive learning experiences for students.
- Transform into an institution of global standards that contributes to nation-building.
- Develop sustainable, cost-effective solutions through innovation and research.

Vision of the department

To emerge as a leading centre for the advancement of knowledge and technologies in computer engineering, promoting innovation, and contributing to long-term prosperity locally and globally, in line with our commitment to empowering society through quality education and research for socio-economic progress.

Mission of the department

- To inspire and guide students to excel in computer engineering, nurturing innovation and problem-solving skills.
- To provide accessible and affordable quality education, serving as a pillar of learning for our society.
- To deliver transformative, holistic, and value-based educational experiences, fostering responsible and ethical leadership.
- To actively engage in innovative research and development, creating sustainable and cost-effective solutions to address real-world challenges.

Programme Outcomes (POs)

Engineering Graduates will be able to:

PO1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Programme Specific Outcomes (PSOs)

PSO1: Demonstrate a deep understanding of computer engineering principles and apply this knowledge to design and develop innovative hardware and software solutions that address real-world challenges.

PSO2: Exhibit strong ethical and professional conduct in computer engineering practice, including a focus on cyber security, data integrity, and responsible use of technology to ensure the security and privacy of digital systems.

PSO3: Collaborate effectively within diverse teams, communicate technical concepts clearly and concisely, and adapt to evolving technologies, thereby contributing to the successful implementation of computer engineering projects and fostering advancements in the field.

SAVITRIBAI PHULE PUNE UNIVERSITY THIRD YEAR OF COMPUTER ENGINEERING (2019 COURSE) 310253: Artificial Intelligence

Course Objectives:

- To understand the concept of Artificial Intelligence (AI) in the form of various Intellectual tasks
- To understand Problem Solving using various peculiar search strategies for AI
- To understand multi-agent environment in competitive environment
- To acquaint with the fundamentals of knowledge and reasoning
- To devise plan of action to achieve goals as a critical part of AI
- To develop a mind to solve real world problems unconventionally with optimality

Course Outcomes:

After completion of the course, students should be able to

CO1: Identify and apply suitable Intelligent agents for various AI applications

CO2: Build smart system using different informed search / uninformed search or heuristic approaches

CO3: Identify knowledge associated and represent it by ontological engineering to plan a strategy to solve given problem

CO4: Apply the suitable algorithms to solve AI problems

CO5: Implement ideas underlying modern logical inference systems

CO6: Represent complex problems with expressive yet carefully constrained language of representation

Suggested List of Laboratory Experiments/Assignments	
Sr. No.	**Group A**
1	Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.
2	Implement A star Algorithm for any game search problem.
3	Implement Greedy search algorithm for any of the following application: I. Selection Sort II. Minimum Spanning Tree III. Single-Source Shortest Path Problem IV. Job Scheduling Problem V. Prim's Minimal Spanning Tree Algorithm VI. Kruskal's Minimal Spanning Tree Algorithm VII. Dijkstra's Minimal Spanning Tree Algorithm
	Group B
4	Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.
5	Develop an elementary chatbot for any suitable customer interaction application.
	Group C
13	Implement any one of the following Expert System I. Information management II. Hospitals and medical facilities III. Help desks management IV. Employee performance evaluation V. Stock market trading VI. Airline scheduling and cargo schedules

INDEX

Sr. No	Name Of The Experiment	Date Of Start	Date Of Completion	Sign
1.	Implement Depth First Search algorithm and Breadth First Search algorithm, Use an undirected graph and developed a recursive algorithm for searching all the vertices of a graph or tree data structure.			
2.	Implement A star Algorithm for any game search problem.			
3.	Implement Greedy search algorithm for any of the following application: 1. Prim's Minimal Spanning Tree Algorithm. 2. Dijkstra's Minimal Spanning Tree Algorithm.			
4.	Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for N-Queens Problems or a graph coloring problem.			
5.	Developed an elementary Chatbot foe any suitable interaction application.			

Practical No-1

Title: Implement depth first search algorithm and Breadth First Search algorithm, use an undirected graph and develop arecursive algorithm for searching all the vertices of a graph or tree data structure.

Objectives:

1. Implementing DFS Algorithm.
2. Implementing BFS Algorithm

1. Depth First Search (DFS):

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a Boolean visited array.

Approach:

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic ideais to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Algorithm:

Create a recursive function that takes the index of the node and a visited array.

1. Mark the current node as visited and print the node.
2. Traverse all the adjacent and unmarked nodes and call the recursive function with the indexof the adjacent node.

Implementation:

Code:

```
graph={
    'A':['B','C','D'],
    'B':['E'],
         'C':['D','E'],
         'D':[],
         'E':[]
        }
        print(graph)

    visited=set()

    def dfs(visited,graph,root):
         if root not in visited:
              print(root)
              visited.add(root)
              for neighbour in graph[root]:
                   dfs(visited,graph,neighbour)
    dfs(visited,graph,"A)
```

Output:

```
Following is the Depth-First Search
A
B
E
C
D
```

2. Breadth-First Search (BFS):

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

Algorithm:

The steps involved in the BFS algorithm to explore a graph are given as follows –

Step 1: SET STATUS = 1 (ready state) for each node in G

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

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeuer a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueuer all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

Step 6: EXIT

Implementation:

Code:

```
graph={
               "A":["B","D"],
               "B":["A","C"],
               "D":["F","E","A"],
               "C":["B"],
               "E":["D","F","G"],
               "F":["D","E","H"],
               "G":["E","H"],
               "H":["F","G"]
          }
          print(graph)
          visited={}
          level={}
          parent={}
          bts_traversal_output=[]
          for node in graph.keys():
               visited [node]=False
               parent  [node]=None
               level[node]=-1
          print(visited)
          print(parent)
          print(level)
          from queue import Queue
          queue=Queue()
          s="A"
          visited [s]=True
          level[s]=0
          queue.put(s)
          while not queue.empty():
               u= queue.get()
               bts_traversal_output.append(u)
               for v in graph[u]:
                    if not visited[v]:
                         visited [v]= True
                         parent [v] = u
                         level [v] = level [u]+1
                         queue.put(v)
          print(visited)
          print(parent)
print(level)
```

Output:

Following is the BFS Sequence

{‘A’: [‘B’,’D’],

‘B’: [‘A’, ‘C’],

‘D’: [‘F’,’E’,’A’],

‘F’: [‘D’,’E’,’H’],

‘G’: [‘E’,’H’]

‘H’: [‘F’,’G’]

Conclusion:

We have learned about DFS & BFS along with its implementation using Python programming language.

Practical No-2

Title: Implement A star Algorithm for any game search problem.

Objective:

1. implement A star Algorithm.

Theory:

A* Algorithm:

1. A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
2. A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
3. It is essentially a best first search algorithm.

Working:

A* Algorithm works as-

1. It maintains a tree of paths originating at the start node.
2. It extends those paths one edge at a time.
3. It continues until its termination criterion is satisfied.

A* Algorithm extends the path that minimizes the following function-

f(n) = g(n) + h(n)

Algorithm:

The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED. OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet. CLOSED contains those nodes that have already been visited.

The algorithm is as follows-

Step-01:

1. Define a list OPEN.
2. Initially, OPEN consists solely of a single node, the start node S.

Step-02:

1. If the list is empty, return failure and exit.

Step-03:

1. Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
2. If node n is a goal state, return success and exit.

Step-04:

1. Expand node n.

Step-05:

1. If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
2. Otherwise, go to Step-06.

Step-06:

1. For each successor node,
2. Apply the evaluation function f to the node.
3. If the node has not been in either list, add it to OPEN.

Step-07:

1. Go back to Step-02.

Implementation:

Code:

```
def aStarAlgo(start_node, stop_node):

open_set = set(start_node)
closed_set = set()
g = {}
parents = {}

g[start_node] = 0
parents[start_node] = start_node

while len(open_set) > 0:
n = None

for v in open_set:
if n == None or g[v] + heuristic(v) < g[n] +
heuristic(n): n = v

if n == stop_node or Graph_nodes[n] == None:
```

```
pass
else:
for (m, weight) in get_neighbors(n):
if m not in open_set and m not in closed_set:
open_set.add(m)
parents[m] = n
g[m] = g[n] + weight

else:
if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n
if m in closed_set:
closed_set.remove(m)
open_set.add(m)
if n == None:
print('Path does not exist!')
return None

if n == stop_node:
path = []
while parents[n] != n:
path.append(n)
n = parents[n]
path.append(start_node)

path.reverse()
print('Path found: {}'.format(path)) return path

open_set.remove(n)
closed_set.add(n)
print('Path does not exist!')
return None

def get_neighbors(v):
if v in Graph_nodes:
return Graph_nodes[v]
else:
return None
def heuristic(n):
H_dist = {
'A': 11,
'B': 6,
'C': 99,
'D': 1,
'E': 7,
'G': 0,
}
return H_dist[n]
```

```
Graph_nodes = {
'A': [('B', 2), ('E', 3)],
'B': [('C', 1),('G', 9)],
'C': None,
'E': [('D', 6)],
'D': [('G', 1)],
}
aStarAlgo('A', 'G')
Path found: ['A', 'E', 'D', 'G']
```

Output:

[‘A’,’E’,’D’,’G]

Conclusion:

A* is very strong algorithm with almost limitless potential. It is, however, only as good as its heuristic function, which may vary significantly depending on the nature of the issue.

Practical No-3

Title: Implement greedy search algorithm for prims minimal spanning tree.

Objective:

1. Prim’s Minimal Spanning Tree
2. Dijkstra’s Minimal Spanning Tree Algorithm

Theory:

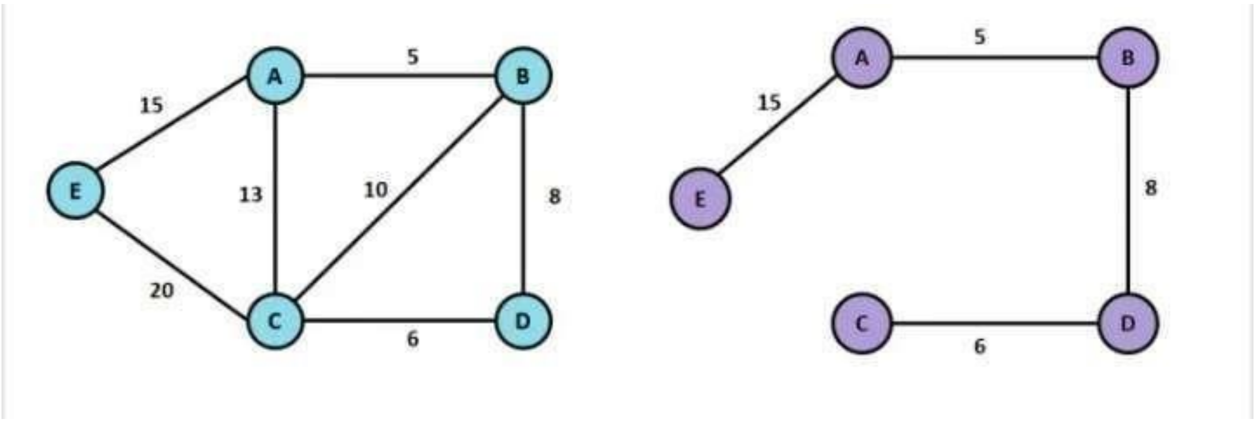
1. Prim’s Minimal Spanning Tree Algorithm

Prim’s algorithm is a minimum spanning tree algorithm that takes a graph as input and creates a minimum spanning tree from the subset of that graph. It finds a minimum spanning tree for the given weighted undirected graph.

Implementation:

The implementation of Prim’s algorithm is based on the following steps:

1. Take any vertex as the source and set its weight to 0. Set the weights of all other vertices to infinity.
2. For every adjacent vertex, if the current weight is more than that of the current edge, then we replace it with the weight of the current edge.
3. Then, we mark the current vertex as visited.
4. Repeat these steps for all the given vertices in ascending order of weight.



Code:

```
INF = 9999999

V = 5

G = [[0, 19, 5, 0, 0],

         [19, 0, 5, 9, 2],

                       [5, 5, 0, 1, 6],

         [0, 9, 1, 0, 1],

         [0, 2, 6, 1, 0]]


selected = [0, 0, 0, 0, 0]

no_edge = 0

selected[0] = True

print("Edge : Weight\n")


sumx=0

while (no_edge < V - 1):

     minimum = INF

     x = 0

     y = 0

     for i in range(V):

          if selected[i]:

               for j in range(V):

                    if ((not selected[j]) and
               G[i][j]):

                         if minimum > G[i][j]:

                              minimum = G[i][j]

                              x = i

                              y = j

print(str(x) + "-" + str(y) + ":" + str(G[x][y]))

sumx+=G[x][y]

print(sumx)
```

```
            selected[y] = True
            no_edge += 1
```

Output:

```
Edge: Weight

0-2:5

5

2-3:1

6

3-4:1

7

4-1:2

9
```

2. Dijkstra’s Minimal Spanning Tree Algorithm

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D. Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex.

Dijkstra's Algorithm Complexity

<u>Time Complexity: O(E Log V)</u>

where, E is the number of edges and V is the number of vertices.

<u>Space Complexity: O(V)</u>

Dijkstra's Algorithm Applications

1. To find the shortest path
2. In social networking applications
3. In a telephone network
4. To find the locations in the map

Basics of Dijkstra's Algorithm

1. Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
2. The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
3. Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
4. The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Requirements

Dijkstra's Algorithm can only work with graphs that have positive weights. This is because, during the process, the weights of the edges have to be added to find the shortest path.

If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.

Implementation:

Code:

```
class Graph():
def___init__(self, vertices):
self.V = vertices
self.graph = [[0 for column in range(vertices)]
for row in range(vertices)]
def printSolution(self, dist):
print("Vertex \t Distance from Source")
for node in range(self.V):
print(node, "\t\t", dist[node])
def minDistance(self, dist, sptSet):
min = 1e7
for v in range(self.V):
if dist[v] < min and sptSet[v] == False:
min = dist[v]
min_index = v
return min_index
```

```
def dijkstra(self, src):
dist = [1e7] * self.V
dist[src] = 0
sptSet = [False] * self.V
for cout in range(self.V):
u = self.minDistance(dist, sptSet)
sptSet[u] = True
for v in range(self.V):
if (self.graph[u][v] > 0 and
sptSet[v] == False and
dist[v] > dist[u] + self.graph[u][v]):
dist[v] = dist[u] + self.graph[u][v]
self.printSolution(dist)
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
 [4, 0, 8, 0, 0, 0, 0, 11, 0],
 [0, 8, 0, 7, 0, 4, 0, 0, 2],
 [0, 0, 7, 0, 9, 14, 0, 0, 0],
 [0, 0, 0, 9, 0, 10, 0, 0, 0],
 [0, 0, 4, 14, 10, 0, 2, 0, 0],
 [0, 0, 0, 0, 0, 2, 0, 1, 6],
 [8, 11, 0, 0, 0, 0, 1, 0, 7],
[0, 0, 2, 0, 0, 0, 6, 7, 0]
]
g.dijkstra(0)
```

Output:

Vertex	Distance From Source
1	0
2	4
3	12
4	19
5	21
6	11
7	9
8	8
9	14

Conclusion:

Greedy algorithm refer to class of algorithm that use a greedy approach to find the optimal solution to some optimization problem.

Practical No-4

Title: Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.

Theory:

The N-Queens problem is a puzzle of placing exactly N queens on an NxN chessboard, such that no two queens can attack each other in that configuration. Thus, no two queens can lie in the same row, column or diagonal.

We have already discussed the backtracking solution to the 8 Queen problem here. In the backtracking algorithm, we consider possible configurations one by one and backtrack if we hit a dead end.

The branch and bound solution is somehow different, it generates a partial solution until it figures that there's no point going deeper as we would ultimately lead to a dead end.

In the backtracking approach, we maintain an 8x8 binary matrix for keeping track of safe cells (by eliminating the unsafe cells, those that are likely to be attacked) and update it each time we place a new queen. However, it required O(n^2) time to check safe cell and update the queen.

In the 8 queen’s problem, we ensure the following:

1. no two queens share a row.
2. no two queens share a column.
3. no two queens share the same left diagonal.
4. no two queens share the same right diagonal.

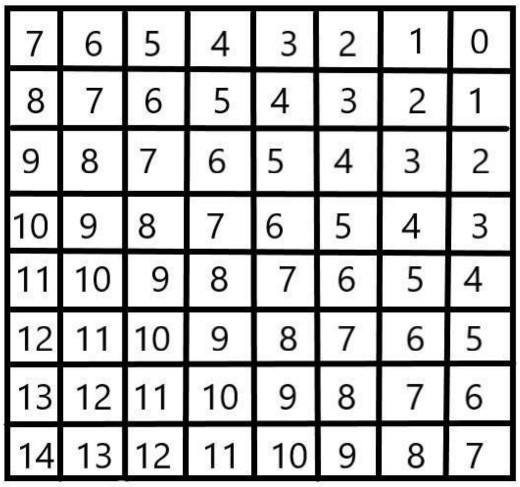
we already ensure that the queens do not share the same column by the way we fill out our auxiliary matrix (column by column). Hence, only the left out 3 conditions are left out to be satisfied.

Applying the branch and bound approach:

The branch and bound approach suggest that we create a partial solution and use it to ascertain whether we need to continue in a particular direction or not. For this problem, we create 3 arrays to check for conditions 1,3 and 4. The Boolean arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.The indexes on these arrays would help us know which queen we are analyzing.

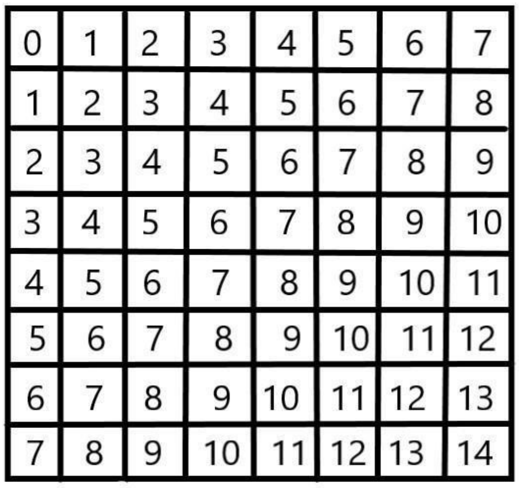
Pre-processing:

create two NxN matrices, one for top-left to bottom-right diagonal, and other for top-right to bottom-left diagonal. We need to fill these in such a way that two queens sharing same top-left bottom-right diagonal will have same value in slash Diagonal and two queens sharing same top-right bottom-left diagonal will have same value in backSlashDiagnol.



7	6	5	4	3	2	1	0
8	7	6	5	4	3	2	1
9	8	7	6	5	4	3	2
10	9	8	7	6	5	4	3
11	10	9	8	7	6	5	4
12	11	10	9	8	7	6	5
13	12	11	10	9	8	7	6
14	13	12	11	10	9	8	7

slash diagnol[row][col] = row + col



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

backslash diagnol[row][col] = row-col+(N-1)

For placing a queen *i* on row *j*, check the following:

1. whether row 'j' is used or not.
2. whether slashDiagnol 'i+j' is used or not.
3. whether backSlashDiagnol 'i-j+7' is used or not.

Code:

```
global N

N = 4

def printSolution(board):

    for i in range(N):

        for j in range(N):

            print (board[i][j],end=' ')

        print()

def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    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):
    if col >= N:
        return True
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1) == True:
                return True
            board[i][col] = 0
    # if the queen can not be placed in any row in
    # this column col then return false
    return False
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
  solveNQ()
```

Output:

```
0010

1000

0001

0100

Out [1]:

True
```

Practical No-5

Title: Develop an elementary chatbot for any suitable customer interaction application.

Theory:

In this Article, you will learn about How to Make a Chatbot in Python Step By Step.

1. Prepare the Dependencies.
2. Import Classes.
3. Create and Train the Chatbot.
4. Communicate with the Python Chatbot.
5. Train your Python Chatbot with a Corpus of Data.

How to Make a Chatbot in Python?

In the past few years, catboats in Python have become wildly popular in the tech and business sectors. These intelligent bots are so adept at imitating natural human languages and conversing with humans, that companies across various industrial sectors are adopting them. From e-commerce firms to healthcare institutions, everyone seems to be leveraging this nifty tool to drive business benefits. In this article, we will learn about Chabot using Python and how to make Chabot in python.

What is a Chatbot?

A Chabot is an AI-based software designed to interact with humans in their natural languages. These catboats are usually converse via auditory or textual methods, and they can effortlessly mimic human languages to communicate with human beings in a human-like manner. A Chabot is arguably one of the best applications of natural language processing.

Chatboats can be categorized into two primary variants

1. Rule-Based.
2. Self learning.

The Rule-based approach trains a Chabot to answer questions based on a set of pre-determined rules on which it was initially trained. These set rules can either be very simple or very complex. While rule-based catboats can handle simple queries quite well, they usually fail to process more complicated queries/requests.

As the name suggests, self-learning bots are catboats that can learn on their own. These leverage advanced technologies like Artificial Intelligence and Machine Learning to train themselves from instances and behaviours. Naturally, these catboats are much smarter than rule-based bots. Self-learning bots can be further divided into two categories – Retrieval Based or Generative.

1. Retrieval-based Chatboats

A retrieval-based Chabot is one that functions on predefined input patterns and set responses. Once the question/pattern is entered, the Chabot uses a heuristic approach to deliver the appropriate response. The retrieval-based model is extensively used to design goal-oriented catboats with customized features like the flow and tone of the bot to enhance the customer experience.

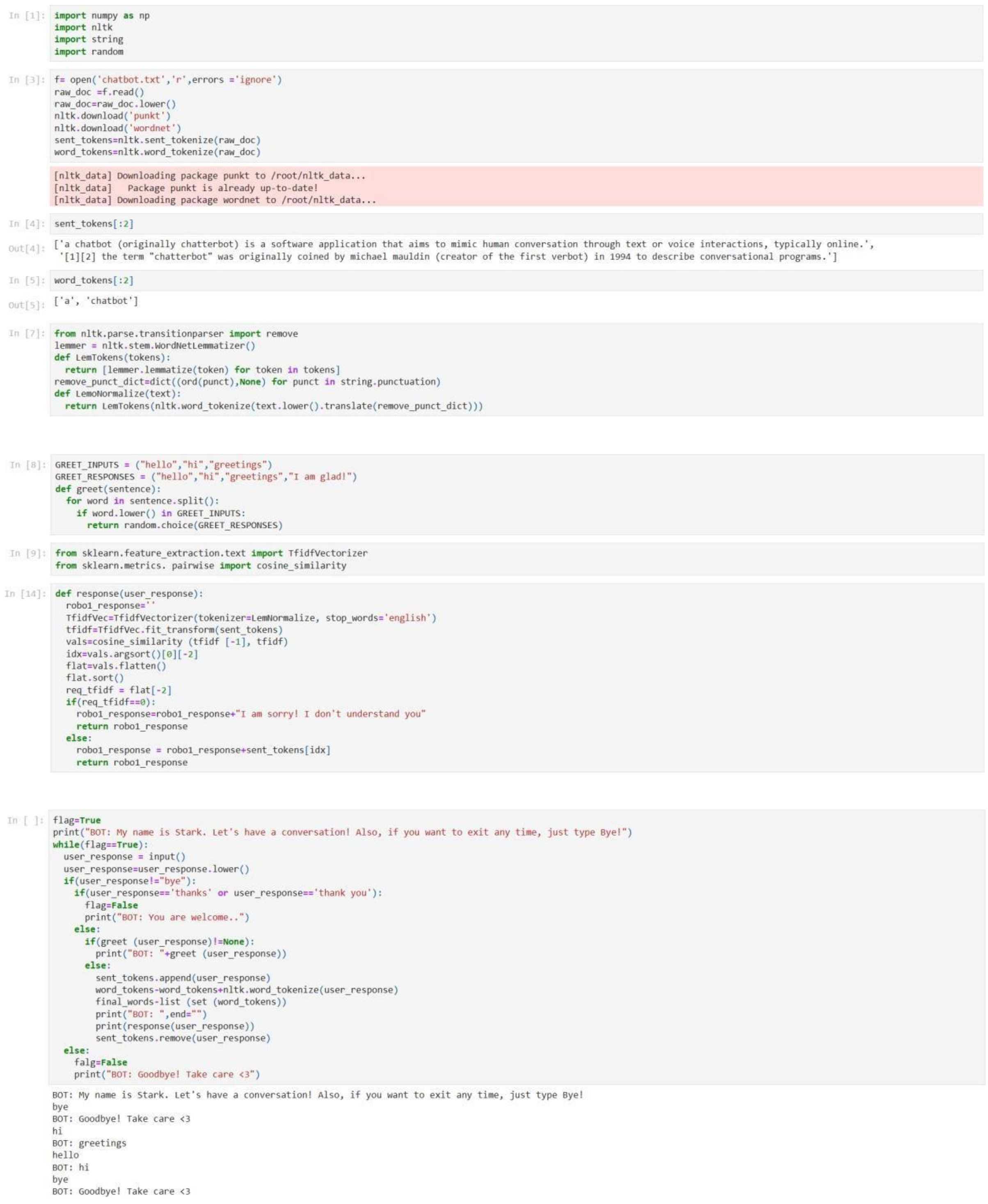
2. Generative Chatbot

Unlike retrieval-based catboats, generative catboats are not based on predefined responses – they leverage seq2seq neural networks. This is based on the concept of machine translation where the source code is translated from one language to another language. In seq2seq approach, the input is transformed into an output.

The first Chabot dates back to 1966 when Joseph Weizenbaum created ELIZA that could imitate the language of a psychotherapist in only 200 lines of code. However, thanks to the rapid advancement of technology, we’ve come a long way from scripted catboats to catboats in python today.

Implementation:

Code:



```
In [1]: import numpy as np
        import nltk
        import string
        import random

In [3]: f= open('chatbot.txt','r',errors ='ignore')
        raw_doc =f.read()
        raw_doc=raw_doc.lower()
        nltk.download('punkt')
        nltk.download('wordnet')
        sent_tokens=nltk.sent_tokenize(raw_doc)
        word_tokens=nltk.word_tokenize(raw_doc)

        [nltk_data] Downloading package punkt to /root/nltk_data...
        [nltk_data]   Package punkt is already up-to-date!
        [nltk_data] Downloading package wordnet to /root/nltk_data...

In [4]: sent_tokens[:2]

Out[4]: ['a chatbot (originally chatterbot) is a software application that aims to mimic human conversation through text or voice interactions, typically online.',
         '[1][2] the term "chatterbot" was originally coined by michael mauldin (creator of the first verbot) in 1994 to describe conversational programs.']

In [5]: word_tokens[:2]

Out[5]: ['a', 'chatbot']

In [7]: from nltk.parse.transitionparser import remove
        lemmer = nltk.stem.WordNetLemmatizer()
        def LemTokens(tokens):
          return [lemmer.lemmatize(token) for token in tokens]
        remove_punct_dict=dict((ord(punct),None) for punct in string.punctuation)
        def LemoNormalize(text):
          return LemTokens(nltk.word_tokenize(text.lower().translate(remove_punct_dict)))

In [8]: GREET_INPUTS = ("hello","hi","greetings")
        GREET_RESPONSES = ("hello","hi","greetings","I am glad!")
        def greet(sentence):
          for word in sentence.split():
            if word.lower() in GREET_INPUTS:
              return random.choice(GREET_RESPONSES)

In [9]: from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.metrics. pairwise import cosine_similarity

In [14]: def response(user_response):
           robo1_response=''
           TfidfVec=TfidfVectorizer(tokenizer=LemNormalize, stop_words='english')
           tfidf=TfidfVec.fit_transform(sent_tokens)
           vals=cosine_similarity (tfidf [-1], tfidf)
           idx=vals.argsort()[0][-2]
           flat=vals.flatten()
           flat.sort()
           req_tfidf = flat[-2]
           if(req_tfidf==0):
             robo1_response=robo1_response+"I am sorry! I don't understand you"
             return robo1_response
           else:
             robo1_response = robo1_response+sent_tokens[idx]
             return robo1_response

In [ ]: flag=True
        print("BOT: My name is Stark. Let's have a conversation! Also, if you want to exit any time, just type Bye!")
        while(flag==True):
          user_response = input()
          user_response=user_response.lower()
          if(user_response!="bye"):
            if(user_response=='thanks' or user_response=='thank you'):
              flag=False
              print("BOT: You are welcome..")
            else:
              if(greet (user_response)!=None):
                print("BOT: "+greet (user_response))
              else:
                sent_tokens.append(user_response)
                word_tokens=word_tokens+nltk.word_tokenize(user_response)
                final_words=list (set (word_tokens))
                print("BOT: ",end="")
                print(response(user_response))
                sent_tokens.remove(user_response)
          else:
            falg=False
            print("BOT: Goodbye! Take care <3")

        BOT: My name is Stark. Let's have a conversation! Also, if you want to exit any time, just type Bye!
        bye
        BOT: Goodbye! Take care <3
        hi
        BOT: greetings
        hello
        BOT: hi
        bye
        BOT: Goodbye! Take care <3
```