Bharati Vidyapeeth's College of Engineering for Women, Pune

Department of Information Technology

Lab Practice- II (Artificial Intelligence)

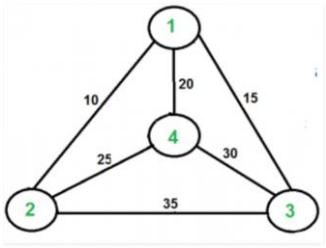
Lab Manual

1) Problem Statement: Identify and Implement heuristic and search strategy for Travelling Salesperson Problem.

Theory: Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities as an adjacency matrix, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

The Nearest Neighbour Method

This is perhaps the simplest TSP heuristic. The key to this method is to always visit the nearest destination and then go back to the first city when all other cities are visited. To solve the TSP using this method, choose a random city and then look for the closest unvisited city and go there. Once you have visited all cities, you must return to the first city.



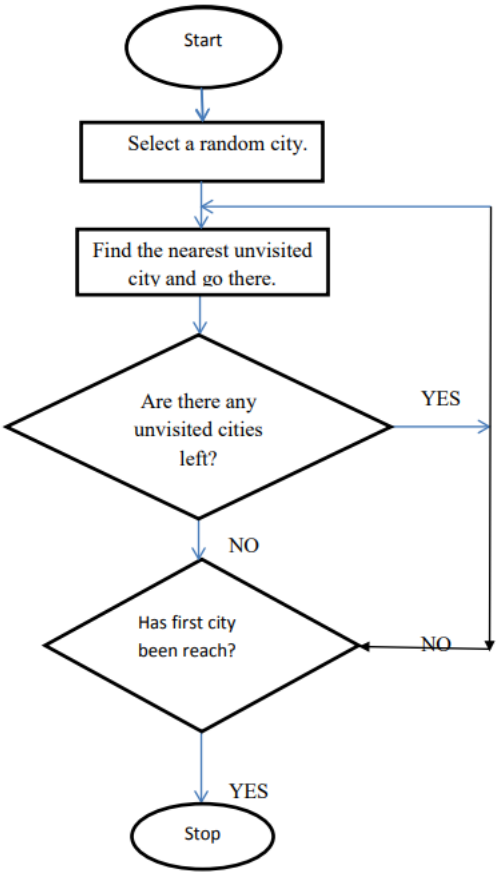
Consider the graph shown in the figure above. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hard problem.

Algorithmic Steps:

The algorithm generates the optimal path to visit all the cities exactly once and return to the starting city.

Do for all the cities:

1. select a city as current city.
2. find out the shortest edge connecting the current city and an unvisited city.
3. set the new city as current city.
4. mark the previous current city as visited.
5. if all the cities are visited, then terminate.
6. Go to step 2.



Flowchart of Nearest Neighbour

Attachment:

1) Implementation
2) Output

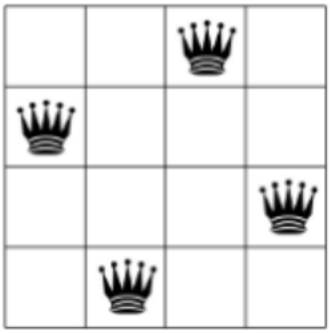
Implementation

```
import copy
inf = float('inf')

class TSP_AI:

# Travelling Salesman Problem Using Nearest Neighbor
    def __init__(self, city_matrix = None, source = 0):
        self.city_matrix = [[0]*4]*4 if city_matrix is None else city_m
atrix
        self.n : int = len(self.city_matrix)
        self.source : int = source


    def Input(self):
        self.n = int(input('Enter city count : '))
 # Get the distances between cities
        for i in range(self.n):
            self.city_matrix.append([
                inf if i == j else int(input(f'Cost to travel from city
 {i+1} to {j+1} : '))
                for j in range( self.n)
            ])

# Get the source city
        self.source = int(input('Source: ')) % self.n

# Initially minCost is infinity
    def solve(self):
        minCost = inf
        for i in range(self.n):
            print("Path", end='')

# Calling solver for each as source city
            cost = self._solve(copy.deepcopy(city_matrix), i, i)

            print(f" -> {i+1}    :    Cost = {cost}")

# If this cost is optimal, save it
            if cost and cost < minCost: minCost = cost

        return minCost


    def _solve(self, city_matrix, currCity = 0, source = 0):
        if self.n < 2: return 0
        print(f" -> {currCity+1}", end='')
```

```
 # Set all values in the currCity column as infinity
 #(once visited, shouldn't be visited anymore)
        for i in range(self.n):
            city_matrix[i][currCity] = inf
        currMin, currMinPos = inf, 0
        for j in range(self.n):

# Get the nearest city to the current city
            if currMin > city_matrix[currCity][j]:

                currMin, currMinPos = city_matrix[currCity][j], j

# If currMin is infinity(i.e. all cities have been visited,
#return cost of moving from this last city to start city to complete th
e path-loop)
        if currMin == inf: return self.city_matrix[currCity][source]

# Set distance from currCity to next city and vice versa to infinity
        city_matrix[currCity][currMinPos] = city_matrix[currMinPos][cur
rCity] = inf

# Calling the next recursion for selected city
        return currMin + self._solve(city_matrix, currMinPos, source)



if __name__ == '__main__':
    city_matrix = [
        [inf, 20,  15,  10],
        [20,  inf, 45,  25],
        [15,  45,  inf, 40],
        [10,  25,  40,  inf]
    ]

    source_city = 0
    tsp = TSP_AI(city_matrix, source_city)
    print(f"Optimal Cost : {tsp.solve()}")
```

2) Problem Statement: Implement N-Queens Problem as Constraints Satisfaction Problem.

Theory: The **goal** of the 4-queens problem is to **place four queens on a 4 x 4 chessboard such that no queen attacks any other by being in the same row, column, or diagonal.**



4-Queens Solution

1. **States:** Any arrangement of 0 to 4 queens on the board is a state.
2. **Initial state:** No queens on the board.
3. **Actions:** Add a queen to any empty square.
4. **Transition model:** Returns the board with a queen added to the specified square.
5. **Goal test:** 4 queens are on the board, none attacked.

1. The solution starts by placing the **first queen** in the **3rd column** of the and the **1st row** (You can place it in any column initially. I have placed in the third column).
2. Now, we must place the **second queen**. We know that the condition for this problem is: We cannot place a queen in the same row, same column, & diagonal to it. Thus. we cannot put the second queen in the 1st row, 3rd column & 2nd/4th column of the 2nd row, 1st column of 3rd row (as it is diagonal to first queen). Thus, the **second queen** is placed in the **1st column** of **2nd row.**
3. Now, we must place the **third queen.** This queen should not be in the same row, column & diagonal of both the first as well as the second queen. Therefore,

- With respect to First **queen**: 1st row, 3rd column & 2nd/4th column of the 2nd row, 1st column of 3rd row is excluded.
- With respect to **Second Queen**: 2nd row, 1st column, & 1st,3rd row of 2nd column & 4th row of 3rd column (as it is diagonal to second queen) will be excluded.
- Thus, the **third queen** is placed in **4th column** of **3rd row.**

Finally, the **fourth queen** is placed in such a manner that it does not clash with any of the other three queens. Thus, **fourth queen** will be positioned in the **2nd column** of **4th row**.

Constraint Satisfaction Problem (CP) approach to the N-queens problem

A CP / CP solver works by systematically trying all possible assignments of values to the variables in a problem, to find the feasible solutions. In the 4-queens problem, the solver starts at the leftmost column and successively places one queen in each column, at a location that is not attacked by any previously placed queens.

Propagation and backtracking

There are two key elements to a constraint programming search:

- *Propagation* — Each time the solver assigns a value to a variable, the constraints add restrictions on the possible values of the unassigned variables. These restrictions *propagate* to future variable assignments. For example, in the 4-queens problem, each time the solver places a queen, it can't place any other queens on the row and diagonals the current queen is on. Propagation can speed up the search significantly by reducing the set of variable values the solver must explore.
- *Backtracking* occurs when either the solver can't assign a value to the next variable, due to the constraints, or it finds a solution. In either case, the solver backtracks to a previous stage and changes the value of the variable at that stage to a value that hasn't already been tried. In the 4-queens example, this means moving a queen to a new square on the current column.

Attachment:

1) Implementation
2) Output

Implementation

```
# Function to check if two queens threaten each other or not
def isSafe(mat, r, c):

  # return false if two queens share the same column
  for i in range(r):
    if mat[i][c] == 'Q':
      return False

  # return false if two queens share the same `\` diagonal
  (i, j) = (r, c)
  while i >= 0 and j >= 0:
    if mat[i][j] == 'Q':
      return False
    i = i - 1
    j = j - 1

  # return false if two queens share the same `/` diagonal
  (i, j) = (r, c)
  while i >= 0 and j < len(mat):
    if mat[i][j] == 'Q':
      return False
    i = i - 1
    j = j + 1

  return True


def printSolution(mat):
  for r in mat:
    print(str(r).replace(',', '').replace('\'', ''))
  print()


def nQueen(mat, r):
  # if `N` queens are placed successfully, print the solution
  if r == len(mat):
    printSolution(mat)
    return

  # place queen at every square in the current row `r`
  # and recur for each valid movement
  for i in range(len(mat)):

    # if no two queens threaten each other
    if isSafe(mat, r, i):
      # place queen on the current square
      mat[r][i] = 'Q'
```

```
        # recur for the next row
        nQueen(mat, r + 1)

        # backtrack and remove the queen from the current square
        mat[r][i] = '-'


if __name__ == '__main__':

    # `N × N` chessboard
    N = 4

    # `mat[][]` keeps track of the position of queens in
    # the current configuration
    mat = [['-' for x in range(N)] for y in range(N)]

    nQueen(mat, 0)
```

3) Problem Statement: Implement Water-Jug Problem using Rule Based Reasoning Technique.

Theory:

Water Jug Problem Definition,
"You are given two jugs, a 4-liter one and a 3-liter one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 litres of water into a 4-liter jug?"

A rule-based artificial intelligence produces pre-defined outcomes that are based on a set of certain rules coded by humans. These systems are simple artificial intelligence models which utilize the rule of if-then coding statements. The two major components of rule-based artificial intelligence models are "a set of rules" and "a set of facts".

Representation of water Jug Problem in terms of state-space search,
State: (x, y)
where x represents the quantity of water in a 4-liter jug and y represents the quantity of water in a 3-liter jug.
That is, x = 0, 1, 2, 3, or 4 y = 0, 1, 2, 3
Start state: (0, 0).
Goal state: (2, n) for any n.
Here need to start from the current state and end up in a goal state.

Production Rules for Water Jug Problem

1	(x, y) is X<4 -> (4, Y)	Fill the 4-liter jug
2	(x, y) if Y<3 -> (x, 3)	Fill the 3-liter jug
3	(x, y) if x>0 -> (x-d, d)	Pour some water out of the 4-liter jug.
4	(x, y) if Y>0 -> (d, y-d)	Pour some water out of the 3-liter jug.
5	(x, y) if x>0 -> (0, y)	Empty the 4-liter jug on the ground
6	(x, y) if y>0 -> (x,0)	Empty the 3-liter jug on the ground
7	(x, y) if X+Y >= 4 and y>0 -> (4, y-(4-x))	Pour water from the 3-liter jug into the 4-liter jug until the 4-liter jug is full
8	(x, y) if X+Y>=3 and x>0 -> (x-(3-y), 3))	Pour water from the 4-liter jug into the 3-liter jug until the 3-liter jug is full.
9	(x, y) if X+Y <=4 and y>0 -> (x+y, 0)	Pour all the water from the 3-liter jug into the 4-liter jug.
10	(x, y) if X+Y<=3 and x>0 -> (0, x+y)	Pour all the water from the 4-liter jug into the 3-liter jug.

11	(0, 2) -> (2, 0)	Pour the 2-liter water from the 3-liter jug into the 4-liter jug.
12	(2, Y) -> (0, y)	Empty the 2-liter in the 4-liter jug on the ground.

The solution to Water Jug Problem

1. Current state = (0, 0)

2. Loop until the goal state (2, 0) reached

– Apply a rule whose left side matches the current state

– Set the new current state to be the resulting state

(0, 0) – Start State

(0, 3) – Rule 2, Fill the 3-liter jug

(3, 0) – Rule 9, Pour all the water from the 3-liter jug into the 4-liter jug.

(3, 3) – Rule 2, Fill the 3-liter jug

(4, 2) – Rule 7, Pour water from the 3-liter jug into the 4-liter jug until the 4-liter jug is full.

(0, 2) – Rule 5, Empty the 4-liter jug on the ground

(2, 0) – Rule 9, Pour all the water from the 3-liter jug into the 4-liter jug.

Goal State reached

Attachment:
Implementation
Output

Implementation

```
def pour(jug1, jug2):
    max1, max2, fill = 3, 4, 2  #Change maximum capacity and final capa
city
    print("%d\t%d" % (jug1, jug2))
    if jug2 is fill:
        return
    elif jug2 is max2:
        pour(0, jug1)
    elif jug1 != 0 and jug2 is 0:
        pour(0, jug1)
    elif jug1 is fill:
        pour(jug1, 0)
    elif jug1 < max1:
        pour(max1, jug2)
    elif jug1 < (max2-jug2):
        pour(0, (jug1+jug2))
    else:
        pour(jug1-(max2-jug2), (max2-jug2)+jug2)

print("JUG1\tJUG2")
pour(0, 0)
```

4) Problem Statement: Write a program for the Information Retrieval System using appropriate NLP tools (such as NLTK, Open NLP, …) a. Text tokenization b. Count word frequency c. Remove stop words d. POS tagging

Theory:

What is Natural Language Processing?

Computers and machines are great at working with tabular data or spreadsheets. However, as human beings generally communicate in words and sentences, not in the form of tables. Much information that humans speak or write is unstructured. So it is not very clear for computers to interpret such. In natural language processing (NLP), the goal is to make computers understand the unstructured text and retrieve meaningful pieces of information from it. Natural language Processing (NLP) is a subfield of **artificial intelligence**, in which its depth involves the interactions between computers and humans.

Applications of NLP:

- Machine Translation.
- Speech Recognition.
- Sentiment Analysis.
- Question Answering.
- Summarization of Text.
- Chatbot.
- Intelligent Systems.
- Text Classifications.
- Character Recognition.
- Spell Checking.
- Spam Detection.
- Autocomplete.
- Named Entity Recognition.
- Predictive Typing.

Current challenges in NLP:

1. Breaking sentences into tokens.
2. Tagging parts of speech (POS).
3. Building an appropriate vocabulary.
4. Linking the components of a created vocabulary.
5. Understanding the context.
6. Extracting semantic meaning.
7. Named Entity Recognition (NER).
8. Transforming unstructured data into structured data.
9. Ambiguity in speech.

Easy to use NLP libraries:

a. NLTK (Natural Language Toolkit):

The NLTK Python framework is generally used as an education and research tool. It's not usually used on production applications. However, it can be used to build exciting programs due to its ease of use.

Features:

- Tokenization.
- Part Of Speech tagging (POS).
- Named Entity Recognition (NER).
- Classification.
- Sentiment analysis.
- Packages of chatbots.

Exploring Features of NLTK:

a. Open the text file for processing:

First, we are going to open and read the file which we want to analyze.

```
#Open the text file :
text_file = open("Natural_Language_Processing_Text.txt")

#Read the data :
text = text_file.read()

#Datatype of the data read :
print (type(text))
print("\n")

#Print the text :
print(text)
print("\n")
#Length of the text :
print (len(text))
```

Small code snippet to open and read the text file and analyse it.

```
<class 'str'>


Once upon a time there was an old mother pig who had three little pigs and not enough food to feed them. So when they were old
enough, she sent them out into the world to seek their fortunes.

The first little pig was very lazy. He didn't want to work at all and he built his house out of straw. The second little pig wo
rked a little bit harder but he was somewhat lazy too and he built his house out of sticks. Then, they sang and danced and play
ed together the rest of the day.

The third little pig worked hard all day and built his house with bricks. It was a sturdy house complete with a fine fireplace
and chimney. It looked like it could withstand the strongest winds.


675
```

Text string file.

Next, notice that the data type of the text file read is a **String**. The number of characters in our text file is **675**.

b. Import required libraries:

For various data processing cases in NLP, we need to import some libraries. In this case, we are going to use NLTK for Natural Language Processing. We will use it to perform various operations on the text.

```
#Import required libraries :
import nltk
from nltk import sent_tokenize
from nltk import word_tokenize
```

Importing the required libraries.

c. Sentence tokenizing:

By tokenizing the text with sent_tokenize( ), we can get the text as sentences.

```
#Tokenize the text by sentences :
sentences = sent_tokenize(text)

#How many sentences are there? :
print (len(sentences))

#Print the sentences :
#print(sentences)
sentences
```

Using sent_tokenize( ) to tokenize the text as sentences.

```
9

['Once upon a time there was an old mother pig who had three little pigs and not enough food to feed them.',
 'So when they were old enough, she sent them out into the world to seek their fortunes.',
 'The first little pig was very lazy.',
 "He didn't want to work at all and he built his house out of straw.",
 'The second little pig worked a little bit harder but he was somewhat lazy too and he built his house out of sticks.',
 'Then, they sang and danced and played together the rest of the day.',
 'The third little pig worked hard all day and built his house with bricks.',
 'It was a sturdy house complete with a fine fireplace and chimney.',
 'It looked like it could withstand the strongest winds.']
```

Text sample data.

In the example above, we can see the entire text of our data is represented as sentences and also notice that the total number of sentences here is **9**.

d. Word tokenizing:

By tokenizing the text with word tokenize( ), we can get the text as words.

```
#Tokenize the text with words :
words = word_tokenize(text)

#How many words are there? :
print (len(words))

#Print words :
print (words)
```

Using word_tokenize() to tokenize the text as words.

```
144
['Once', 'upon', 'a', 'time', 'there', 'was', 'an', 'old', 'mother', 'pig', 'who', 'had', 'three', 'little', 'pigs', 'and', 'no
t', 'enough', 'food', 'to', 'feed', 'them', '.', 'So', 'when', 'they', 'were', 'old', 'enough', ',', 'she', 'sent', 'them', 'ou
t', 'into', 'the', 'world', 'to', 'seek', 'their', 'fortunes', '.', 'The', 'first', 'little', 'pig', 'was', 'very', 'lazy',
'.', 'He', 'did', "n't", 'want', 'to', 'work', 'at', 'all', 'and', 'he', 'built', 'his', 'house', 'out', 'of', 'straw', '.', 'T
he', 'second', 'little', 'pig', 'worked', 'a', 'little', 'bit', 'harder', 'but', 'he', 'was', 'somewhat', 'lazy', 'too', 'and',
'he', 'built', 'his', 'house', 'out', 'of', 'sticks', '.', 'Then', ',', 'they', 'sang', 'and', 'danced', 'and', 'played', 'toge
ther', 'the', 'rest', 'of', 'the', 'day', '.', 'The', 'third', 'little', 'pig', 'worked', 'hard', 'all', 'day', 'and', 'built',
'his', 'house', 'with', 'bricks', '.', 'It', 'was', 'a', 'sturdy', 'house', 'complete', 'with', 'a', 'fine', 'fireplace', 'an
d', 'chimney', '.', 'It', 'looked', 'like', 'it', 'could', 'withstand', 'the', 'strongest', 'winds', '.']
```

Text sample data.

Next, we can see the entire text of our data is represented as words and also notice that the total number of words here is **144**.

e. Find the frequency distribution:

Let’s find out the frequency of words in our text.

```
#Import required libraries :
from nltk.probability import FreqDist

#Find the frequency :
fdist = FreqDist(words)

#Print 10 most common words :
fdist.most_common(10)
```

Using FreqDist() to find the frequency of words in our sample text.

```
[('.', 9),
 ('and', 7),
 ('little', 5),
 ('a', 4),
 ('was', 4),
 ('pig', 4),
 ('the', 4),
 ('house', 4),
 ('to', 3),
 ('out', 3)]
```

Printing the ten most common words from the sample text.

Notice that the most used words are punctuation marks and stopwords. We will have to remove such words to analyze the actual text.

List of stop words:

```
from nltk.corpus import stopwords

#List of stopwords
stopwords = stopwords.words("english")
print(stopwords)
```

Importing the list of stop words.

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'y
ourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those',
'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'a
n', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'b
etween', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'of
f', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both',
'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very',
's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'ar
en', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "have
n't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

Text sample data. Removing stop words:

```
#Empty list to store clean words :
clean_words = []

for w in words_no_punc:
    if w not in stopwords:
        clean_words.append(w)

print(clean_words)
print("\n")
print(len(clean_words))
```

Cleaning the text sample data.

```
['upon', 'time', 'old', 'mother', 'pig', 'three', 'little', 'pigs', 'enough', 'food', 'feed', 'old', 'enough', 'sent', 'world',
'seek', 'fortunes', 'first', 'little', 'pig', 'lazy', 'want', 'work', 'built', 'house', 'straw', 'second', 'little', 'pig', 'wo
rked', 'little', 'bit', 'harder', 'somewhat', 'lazy', 'built', 'house', 'sticks', 'sang', 'danced', 'played', 'together', 'res
t', 'day', 'third', 'little', 'pig', 'worked', 'hard', 'day', 'built', 'house', 'bricks', 'sturdy', 'house', 'complete', 'fin
e', 'fireplace', 'chimney', 'looked', 'like', 'could', 'withstand', 'strongest', 'winds']


65
```

Cleaned data.

Final frequency distribution:

```
#Frequency distribution :
fdist = FreqDist(clean_words)

fdist.most_common(10)
```

```
[('little', 5),
 ('pig', 4),
 ('house', 4),
 ('built', 3),
 ('old', 2),
 ('enough', 2),
 ('lazy', 2),
 ('worked', 2),
 ('day', 2),
 ('upon', 1)]
```

Displaying the final frequency distribution of the most common words found.

Part of Speech Tagging (PoS tagging):

Why do we need Part of Speech (POS)?

Can you help me with the can?

Sentence example, “can you help me with the can?”

Parts of speech (PoS) tagging is crucial for syntactic and semantic analysis. Therefore, for something like the sentence above, the word “can” has several semantic meanings. The first “can” is used for question formation. The second “can” at the end of the sentence is used to represent a container. The first “can” is a verb, and the second “can” is a noun. Giving the word a specific meaning allows the program to handle it correctly in both semantic and syntactic analysis

Python Implementation:

a. A simple example demonstrating PoS tagging.

```
#PoS tagging :
tag = nltk.pos_tag(["Studying","Study"])
print (tag)
```

```
[('Studying', 'VBG'), ('Study', 'NN')]
```

Figure 89: PoS tagging example.

```
#PoS tagging example :

sentence = "A very beautiful young lady is walking on the beach"

#Tokenizing words :
tokenized_words = word_tokenize(sentence)

for words in tokenized_words:
    tagged_words = nltk.pos_tag(tokenized_words)

tagged_words
```

```
[('A', 'DT'),
 ('very', 'RB'),
 ('beautiful', 'JJ'),
 ('young', 'JJ'),
 ('lady', 'NN'),
 ('is', 'VBZ'),
 ('walking', 'VBG'),
 ('on', 'IN'),
 ('the', 'DT'),
 ('beach', 'NN')]
```

Full Python sample demonstrating PoS tagging.

Attachment: Implementation, Output

Problem Statement: Write a program for the Tic-Tac-Toe game using mini-max algorithm

Theory:

Minimax Algorithm

The Minimax algorithm is a recursive algorithm used in decision-making and game theory. It delivers an optimal move for the player, considering that the competitor is also playing optimally. This algorithm is widely used for game playing in Artificial Intelligence, such as chess, tic-tac-toe, and myriad double players games.

In this algorithm, two players play the game; one is called **'*MAX*'**, and the other is **'*MIN*.**' The goal of players is to minimize the opponent's benefit and maximize self-benefit. The MiniMax algorithm conducts a depth-first search to explore the complete game tree and then proceeds down to the leaf node of the tree, then backtracks the tree using recursive calls.

To better understand, let us consider an example of tic-tac-toe, a two-player game in which each player plays turn by turn.

State Space representation of Tic-Tac-Toe

Each state is denoted by the positions occupied by the letters of the two players in a 3x3 matrix and other empty places.

1. ***Initial State:*** An empty 3x3 matrix.
2. ***Intermediate State:*** Any arrangement of a 3x3 matrix obtained after applying valid rules on the current state.
3. ***Final State:*** Same letters in a complete row or complete column, or complete diagonal wins the game.
4. ***Rules:*** The players will get turns one after the other; they can mark their letters on the empty cell.

MiniMax Illustration using Tic-Tac-Toe game example

To begin with, we are considering two rivals, Zoro and Sanji, from the One Piece (a famous Japanese Manga series). They decided to play the tic-tac-toe to resolve an argument. Zoro starts the game by picking 'X' and Sanji picks 'O' and plays second. In this example, Zoro will try to

get the maximum possible score, and Sanji will try to minimize the possible score of Zoro. Figure 1 shows the game tree after an intermediate stage.

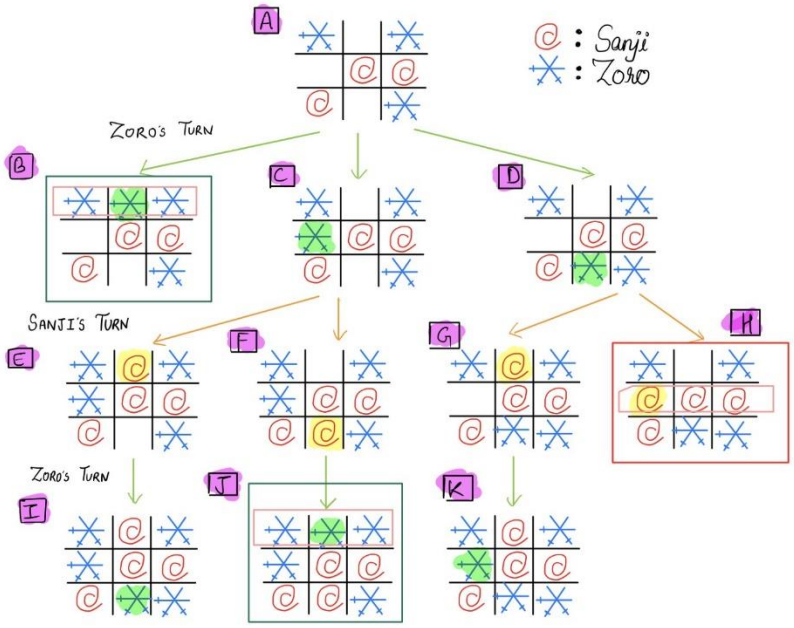


Fig 1. Game Tree of the tic-tac-toe game after a certain state.

During an intermediate state (A) in the game, Zoro has three choices to pick from, i.e., state '*B*,' '*C*,' and '*D*.' Picking state '*B'* will lead to the victory of Zoro, while choosing *'C'* and *'D'* will result in further rounds of the game. In case Zoro attempts for the *'C'* and *'D'* state, Sanji will play accordingly, and so on. There will be scenarios when both win disjointly as Sanji wins in state *'H,'* and Zoro wins in state *'J.'* The other leaf nodes will result in a draw.

At the leaf node, we give values for the backtracking purpose until the initial state is reached. For instance, if Zoro wins, we give +1 (*GREEN*); if Sanji wins, then Zoro will get -1 (*YELLOW*), and 0 (*LIGHT BROWN*) in case of no winner.

Following are the main steps involved in solving the two-player game tree:

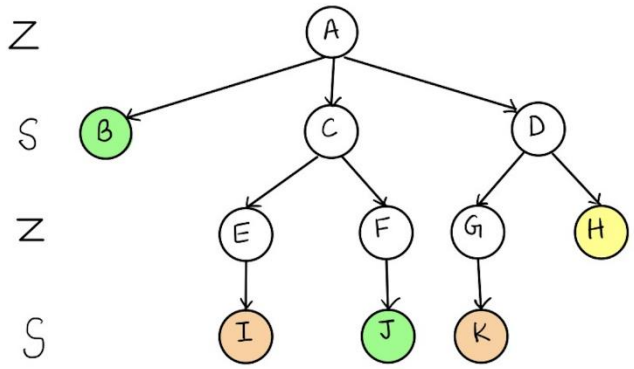


Fig 2. Game tree on Step 1.

Step 1: In the first step, the algorithm generates the entire game tree and applies the utility values (+1/-1/0) for leaf nodes. First of all, we will fill the utility value for node *'E.'* As it is Zoro's turn, he will try to maximize the score by picking 0 from *'I.'* Similarly, for node *'G,'* the utility value will be 0. For node *'F,'* Zoro will select +1. Figure 3 shows updated game tree.

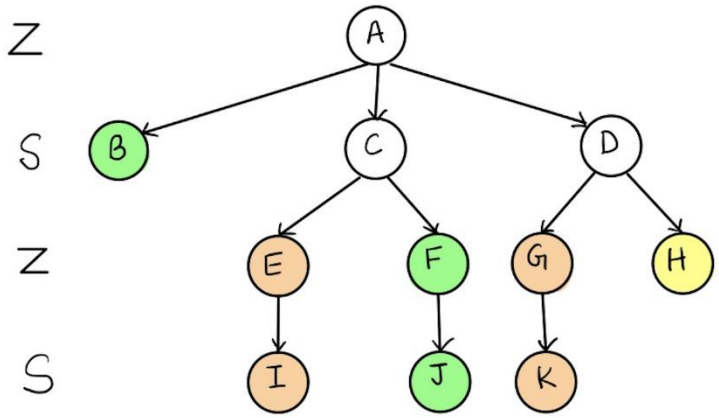


Fig 3. Game tree on Step 2.

Step 2: Now, we will fill the utility values for *'C'* and *'D.'* As it is Sanji's turn so he will try to minimize Zoro's score, so for *'C,'* he will pick a minimum of *'E'* and *'F,'* which is 0, and for *'D'* he will pick minimum from *'G'* and *'H,'* which is -1. The updated game tree is shown in Figure 4.

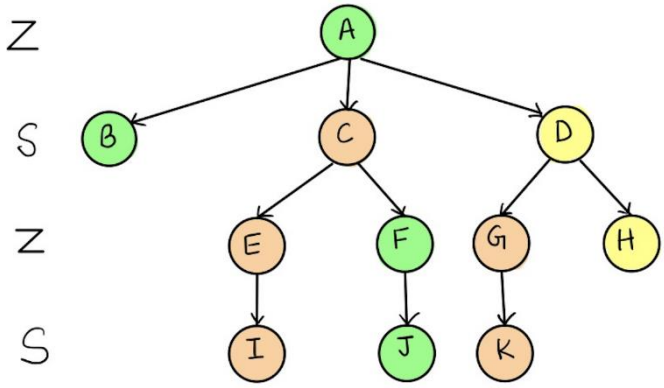


Fig 4. Game tree on Step 3.

Step 3: Figure 4 demonstrates the final tree, with each node having its utility values. At the level of node *'A,'* we have Zoro's turn, so he will pick the maximum from *'B,' 'C,'* and *'D,'* which is +1. Hence, Zoro can win the game.

Therefore, in order to win after the mentioned given state, Zoro has one path that he can opt, i.e., $A \rightarrow B$, which is highlighted as red in Figure 5. Path A → C will result in a draw, and Path A → D will result in Sanji's win.

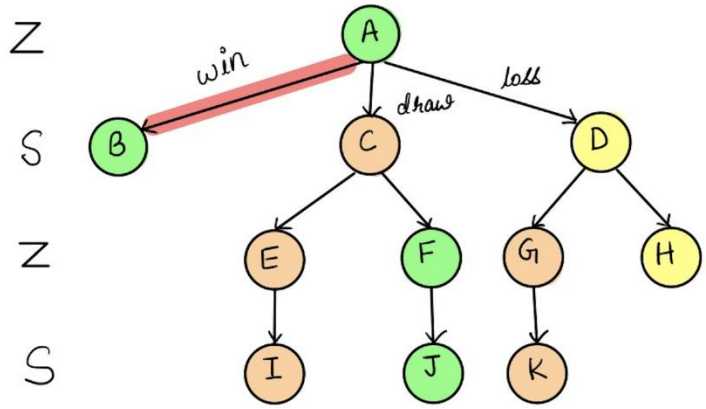


Fig 5. The path that Zoro should take to win the game against Sanji.

Algorithmic Steps of Tic-Tac-Toe Game Implementation are as follows

Each state is denoted by the positions occupied by the letters of the two players in a 3x3 matrix and other empty places.

1. Initial State: An empty 3x3 matrix.

2. Intermediate State: Any arrangement of a 3x3 matrix obtained after applying valid rules on the current state.

3. Final State: Same letters in a complete row or complete column, or complete diagonal wins the game.

4. Rules: The players will get turns one after the other; they can mark their letters on the empty cell.

Attachment: Implementation and Output

```
from math import inf as infinity
from random import choice
import platform
import time
from os import system



HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]


def evaluate(state):

    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0

    return score


def wins(state, player):

    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False


def game_over(state):
```

```
    return wins(state, HUMAN) or wins(state, COMP)


def empty_cells(state):

    cells = []

    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell == 0:
                cells.append([x, y])

    return cells


def valid_move(x, y):

    if [x, y] in empty_cells(board):
        return True
    else:
        return False


def set_move(x, y, player):

    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False


def minimax(state, depth, player):

    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y
```

```
            if player == COMP:
                if score[2] > best[2]:
                    best = score  # max value
            else:
                if score[2] < best[2]:
                    best = score  # min value

        return best


def clean():
    """
    Clears the console
    """
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')


def render(state, c_choice, h_choice):
    """
    Print the board on console
    :param state: current state of the board
    """

    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)


def ai_turn(c_choice, h_choice):
    """
    It calls the minimax function if the depth < 9,
    else it choices a random coordinate.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
```

```
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]

    set_move(x, y, COMP)
    time.sleep(1)


def human_turn(c_choice, h_choice):

    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
            can_move = set_move(coord[0], coord[1], HUMAN)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
```

```
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')


def main():
    """
    Main function that calls all functions
    """
    clean()
    h_choice = ''  # X or O
    c_choice = ''  # X or O
    first = ''  # if human is the first

    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Setting computer's choice
    if h_choice == 'X':
        c_choice = 'O'
    else:
        c_choice = 'X'

    # Human may starts first
    clean()
    while first != 'Y' and first != 'N':
        try:
            first = input('First to start?[y/n]: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Main loop of this game
    while len(empty_cells(board)) > 0 and not game_over(board):
        if first == 'N':
            ai_turn(c_choice, h_choice)
            first = ''
```

```
            human_turn(c_choice, h_choice)
            ai_turn(c_choice, h_choice)

        # Game over message
        if wins(board, HUMAN):
            clean()
            print(f'Human turn [{h_choice}]')
            render(board, c_choice, h_choice)
            print('YOU WIN!')
        elif wins(board, COMP):
            clean()
            print(f'Computer turn [{c_choice}]')
            render(board, c_choice, h_choice)
            print('YOU LOSE!')
        else:
            clean()
            render(board, c_choice, h_choice)
            print('DRAW!')

        exit()


if __name__ == '__main__':
    main()
```