BFS

```
# Create a graph given in the above diagram.
graph = {
    'A': ['B', 'C', 'D'],
    'B': ['A'],
    'C': ['A', 'D'],
    'D': ['A', 'C', 'E'],
    'E': ['D'],
}
# make an empty queue for bfs
visited = []
queue = []
# to print a BFS of a graph
def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        # Remove the front vertex or the vertex at the 0th index from the queue
and print that vertex.
        v = queue.pop(0)
        print(v, end=" ")
        # Get all adjacent nodes of the removed node v from the graph hash table.
        # If an adjacent node has not been visited yet,
        # then mark it as visited and add it to the queue.
        for neigh in graph[v]:
            if neigh not in visited:
```

```python
        visited.append(neigh)
        queue.append(neigh)
# Call the BFS function with the starting node 'A'
bfs(visited, graph, 'A')
```

DFS

```
graph = {
   'A': ['B', 'C', 'D'],
   'B': ['A'],
   'C': ['A', 'D'],
   'D': ['A', 'C', 'E'],
   'E': ['D'],
}
visited=set()#set to keep track of visited nodes of graph
def dfs(visited,graph,node):
   if(node not in visited):
      print(node,end=" ")
      visited.add(node)
      for neigh in graph[node]:
         dfs(visited,graph,neigh)
dfs(visited,graph,'A')
```

Greedy BEST FIRST SEARCH

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
h = [11, 12, 13, 14, 5, 10, 7, 8, 9, 0, 1, 2, 3, 4]
def best_first_search(actual_Src, target, n):
    Closed_list = [False] * n
    Open_list = PriorityQueue()
    Open_list.put((h[actual_Src], actual_Src))
    Closed_list[actual_Src] = True
    while Open_list.empty() == False:
        u = Open_list.get()[1]
        # Displaying the path having the lowest cost
        print(u, end=" ")
        if u == target:
            break
        for v, c in graph[u]:
            if Closed_list[v] == False:
                Closed_list[v] = True
                Open_list.put((h[v], v))
    print()
# Function for adding edges to graph
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
# The nodes shown in the above example (by alphabets) are
```

```
# implemented using integers addedge(x, y, cost);
addedge(0, 1, 3)
addedge(1, 10, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 5)
source = 0
target = 9
best_first_search(source, target, v)
```

A STAR ALGO

```
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]
# Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
```

```
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}
def aStarAlgo(start_node, stop_node, graph_nodes):
    open_set = {start_node}  # Initialize open_set with the starting node
    closed_set = set()
    g = {start_node: 0}  # store distance from the starting node
    parents = {start_node: start_node}  # parents contain an adjacency map of all
nodes
    while open_set:
        n = None
        # node with the lowest f() is found
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or graph_nodes[n] is None:
            pass
        else:
            for (m, weight) in get_neighbors(n, graph_nodes):
                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 is 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, graph_nodes):
    if v in graph_nodes:
        return graph_nodes[v]
    else:
        return None
# Call the function with the provided graph_nodes
aStarAlgo('A', 'J', Graph_nodes)
```

DECISION TREES

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

# Load dataset
ds = pd.read_csv(r"diabetes.csv")
df = pd.DataFrame(ds)

# Display dataset info
print(df.head())
print(df.shape)

# Splitting into dependent and independent variables
x = df.iloc[:, :-1]
y = df.iloc[:, -1]
print("Dependent Variables")
print(x)
print("Independent Variables")
print(y)

# Splitting dataset into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
random_state=0)
```

```
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)

# Decision Tree model
dt = DecisionTreeClassifier(criterion="entropy", max_depth=3)
dt.fit(x_train, y_train)
# Predictions
y_pred = dt.predict(x_test)
print(y_pred)

# Accuracy
print(metrics.accuracy_score(y_test, y_pred))

# Classification report
print(metrics.classification_report(y_test, y_pred))
```

NLTK

```
pip install nltk


import nltk
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')


from nltk.tokenize import sent_tokenize
text="I have a rendezvous with Death, At some disputed barricade.When Spring comes back with rustling shade,And apple blossoms fill coming the air. “I have a rendezvous with Death” When Spring brings back blue days came and fair. And I have learned too to laugh with only my teeth and shake hands without my heart. I have also learned to say, ’Goodbye’,when I mean ‘Good-riddance’:to say ‘Glad to meet you’, without being glad; and to say ‘It’s been nice talking to you’, after being bored"
tokenized_text=sent_tokenize(text)
print(tokenized_text)


from nltk.tokenize import word_tokenize
tokenized_word=word_tokenize(text)
print(tokenized_word)


from nltk.probability import FreqDist
fdisk=FreqDist(tokenized_word)
print(fdisk)


fdisk.most_common(2)
```

```
from nltk.corpus import stopwords
stop_words=set(stopwords.words("english"))
print(stop_words)

filtered_sent=[]
for w in tokenized_word:
    if w not in stop_words:
        filtered_sent.append(w)
print("Tokenized Sentence:",tokenized_word,end="\n")
print("\n")
print("Filtered Sentence:",filtered_sent)

from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize,word_tokenize
ps=PorterStemmer()
stemmed_words=[]
for w in filtered_sent:
        stemmed_words.append(ps.stem(w))
print("Filtered Sentence:",filtered_sent)
print("\n")
print("Stemmed Sentence:",stemmed_words)

import nltk


nltk.download('wordnet')
```

```
from nltk.stem import WordNetLemmatizer

wrl=WordNetLemmatizer()

words=[]

for words in filtered_sent:

   print(words+" ---> "+wrl.lemmatize(words))


nltk.download('averaged_perceptron_tagger')


wrl = WordNetLemmatizer()

lemmatized_words = [wrl.lemmatize(word) for word in
nltk.word_tokenize(text)]

# Perform part-of-speech tagging

final = nltk.pos_tag(lemmatized_words)

final


nltk.download('tagsets')


nltk.help.upenn_tagset('JJ')
```

ANN

```
import pandas as pd

from sklearn.preprocessing import LabelEncoder

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.neural_network import MLPClassifier

from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score


# Load dataset

irisdata = pd.read_csv(r" iris.csv")


# Display dataset info

print(irisdata.head())

print(irisdata.info())

print(irisdata.shape)


# Splitting into dependent and independent variables

df = pd.DataFrame(irisdata)

x = df.iloc[:, :-1]

y = df.iloc[:, -1]

print("Independent Varibales")

print(x)

print("Dependent Variables")

print(y)
```

```
# Encoding categorical labels
le = LabelEncoder()
y = le.fit_transform(y)
# Splitting dataset into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)

# Standardizing features
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
# Neural Network model
mlp = MLPClassifier(hidden_layer_sizes=(10, 10, 10, 10, 10), max_iter=1000)
mlp.fit(x_train, y_train)

# Predictions
prediction = mlp.predict(x_test)
print(prediction)

# Evaluation
print(confusion_matrix(y_test, prediction))
print(classification_report(y_test, prediction))
print(accuracy_score(y_test, prediction))
```