NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

Program No :- 1

Aim: Write a program to implement breath first search traversal

To implement the **Breadth-First Search (BFS)** algorithm for graph traversal. This algorithm is used to traverse or search through a graph in a level-order manner, exploring all neighbours at the current depth before moving on to nodes at the next depth level.

Theory

Breadth-First Search (BFS) is an algorithm for traversing or searching graph data structures. It starts at a specified node (called the "source node") and explores all the neighbours at the present depth level before moving on to nodes at the next depth level.

BFS Characteristics:

1. **Level-wise traversal**: BFS visits nodes level by level starting from the source node.
2. **Queue-based algorithm**: BFS uses a queue to store nodes that need to be explored. This helps ensure the nodes are visited in the correct order.
3. **Complete**: BFS always finds the shortest path in an unweighted graph.
4. **Time complexity**: O(V + E), where V is the number of vertices and E is the number of edges in the graph.
5. **Space complexity**: O(V), due to the storage of the queue.

BFS Algorithm:

1. **Input**: A graph GG, represented by vertices VV and edges EE, and a source node ss.
2. **Output**: The order of nodes visited during BFS.

Algorithm:

1. Start by marking the source node as visited and enqueue it into the queue.
2. While the queue is not empty:
    - Dequeue a node from the front of the queue.
    - Process (print or store) the current node.
    - For each unvisited neighbour of the current node:
        - Mark it as visited.
        - Enqueue it into the queue.
3. Repeat the process until the queue is empty.

Code Implementation (in Python)

Edit with WPS Office

NAME :-   ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :-   IT-C

```
from collections import deque

# Graph class to represent an undirected graph
class Graph:
    def __init__(self, vertices):
        self.V = vertices  # Number of vertices
        self.graph = {i: [] for i in range(self.V)}  # adjacency list representation of the graph

    # Add edge to the graph
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    # Perform BFS traversal from a given source node
    def bfs(self, start):
        # Mark all nodes as not visited
        visited = [False] * self.V

        # Queue for BFS
        queue = deque()

        # Start with the source node
        visited[start] = True
        queue.append(start)

        while queue:
            # Dequeue a vertex from the queue
            node = queue.popleft()
            print(node, end=" ")

            # Visit all the neighbors of the current node
            for neighbor in self.graph[node]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)

# Driver code to test the BFS algorithm
if __name__ == "__main__":
    # Create a graph with 6 vertices
    g = Graph(6)

    # Add edges
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)

    print("BFS traversal starting from vertex 0:")
    g.bfs(0)  # Start BFS traversal from vertex 0
```

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

Explanation of the Code:

1. **Graph Representation**:
   - A class `Graph` is created to represent the graph, where the adjacency list is used for storing the graph. The constructor initializes the graph with a number of vertices, and the graph is represented as a dictionary of lists where each vertex has a list of its neighbors.
2. **add_edge**:
   - This method adds an edge between two vertices uu and vv in an undirected graph.
3. **bfs**:
   - This method performs the BFS traversal starting from the specified source node.
   - The `visited` list is used to mark nodes that have already been visited.
   - The queue is initialized with the starting node, and the while loop continues to process nodes from the queue.
   - Each node's neighbors are added to the queue for further exploration.
4. **Main Code**:
   - In the main code, a graph is created, edges are added, and the BFS traversal is performed starting from vertex 0.

Output:

```
BFS traversal starting from vertex 0:
0 1 2 3 4 5
```

Lesson Learnt:

1. **Understanding BFS**: BFS is an algorithm that explores nodes level by level, making it ideal for finding the shortest path in unweighted graphs. We learned how BFS can be implemented using a queue to store nodes that need to be explored.
2. **Graph Representation**: We implemented the graph as an adjacency list, which is memory efficient and works well with sparse graphs.
3. **Queue Management**: BFS uses a queue data structure, which ensures nodes are processed in the order they were discovered. This is essential for BFS to work as expected and explore all nodes at each depth level before moving to the next one.
4. **Efficiency**: BFS ensures that all nodes are visited only once, making the algorithm efficient with a time complexity of O(V + E), where V is the number of vertices and E is the number of edges.
5. **Applications of BFS**:
   - **Shortest Path in Unweighted Graphs**: BFS can be used to find the shortest path between two nodes in an unweighted graph.
   - **Connected Components**: BFS can help identify all connected components in an undirected graph.

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

PROBLEM NO :- 2

AIM:- Water Jug Problem

The Water Jug Problem is a classic example of state-space search. You are given two jugs with specific capacities, and the task is to measure a target amount of water using the two jugs. The jugs do not have measurement markings, and you can perform several operations:

1. Fill a jug completely.
2. Empty a jug.
3. Pour water from one jug into another until one jug is either full or empty.

The challenge is to determine the minimum number of steps needed to measure the exact target quantity of water in one of the jugs.

Algorithm (Breadth-First Search):

1. **Initialize**: Start with two empty jugs (state (0, 0)).
2. **Queue Initialization**: Push the initial state (0, 0) (both jugs empty) into a queue.
3. **Explore the state**: For each state (a, b):
   - If the current state matches the target amount, return the number of steps.
   - Otherwise, generate new states by performing valid operations like filling, emptying, and pouring between jugs.
4. **State Transition**: For each new state generated, check if it has been visited. If not, mark it as visited and enqueue it.
5. **Termination**: The search ends when we either reach the target amount or exhaust all possible states.

Code:

```
from collections import deque

# Function to perform the BFS search
def water_jug_problem(capacity_a, capacity_b, target):
  # Queue to store states (a, b) where 'a' is water in jug A and 'b' is water in jug B
  visited = set()
  queue = deque([(0, 0, 0)])  # Start with both jugs empty
  visited.add((0, 0))

  while queue:
    a, b, steps = queue.popleft()

    # If we reached the target in either jug, return the number of steps
    if a == target or b == target:
      return steps

    # Generate all possible states
    next_states = [
      (capacity_a, b),  # Fill jug A
      (a, capacity_b),  # Fill jug B
      (0, b),           # Empty jug A
      (a, 0),           # Empty jug B
      (a - min(a, capacity_b - b), b + min(a, capacity_b - b)),  # Pour A to B
      (a + min(b, capacity_a - a), b - min(b, capacity_a - a))   # Pour B to A
    ]
```

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

```
        for state in next_states:
            if state not in visited:
                visited.add(state)
                queue.append((state[0], state[1], steps + 1))

    return -1  # If no solution is found

# Driver code
if __name__ == "__main__":
    capacity_a = 4
    capacity_b = 3
    target = 2

    print(f"Minimum steps to measure {target} liters: {water_jug_problem(capacity_a, capacity_b, target)}")
```

Output:

```
Minimum steps to measure 2 liters: 4
```

Lesson Learnt:

- The Water Jug Problem is a great example of how **Breadth-First Search (BFS)** can be used to explore all possible states in a systematic manner.
- We learned how to handle state transitions and use a queue to manage the search process.
- The problem highlights the importance of considering all possible actions and ensuring that states are revisited only once to avoid cycles.

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

PROBLEM NO :-3

AIM :- Predict the Class of the Flower (Using k-NN Algorithm)

Theory:

The Iris dataset is a well-known dataset used for classification tasks. It contains data on 150 flowers with four attributes: sepal length, sepal width, petal length, and petal width. The goal is to predict the species of a flower based on these attributes.

The **k-Nearest Neighbors (k-NN)** algorithm is a simple, intuitive algorithm that classifies data points based on the majority class of their nearest neighbors. It works by calculating the distance between data points and finding the most frequent class in the nearest neighbors.

Algorithm:

1. **Input**: The Iris dataset with features and target labels.
2. **Train k-NN Model**:
   - Split the dataset into training and testing sets.
   - Use the k-NN algorithm to train the model.
3. **Prediction**:
   - For a new flower, find the k nearest flowers in the training data.
   - Return the most common species from those nearest neighbors.
4. **Evaluate**: Calculate the accuracy of the model by comparing predicted and actual values.

Code:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features (sepal length, sepal width, petal length, petal width)
y = iris.target  # Labels (species of the flower)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create k-NN classifier and fit the model
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Make predictions
y_pred = knn.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of the k-NN model: {accuracy * 100:.2f}%")

# Example: Predict the class for a new flower
sample_flower = [[5.1, 3.5, 1.4, 0.2]]  # Example input (sepal length, sepal width, petal length, petal width)
predicted_class = knn.predict(sample_flower)
```

Edit with WPS Office

```
print(f"Predicted class for the given flower: {iris.target_names[predicted_class]}")
```

Output:

```
Accuracy of the k-NN model: 100.00%
Predicted class for the given flower: setosa
```

Lesson Learnt:

- We learned how **k-NN** works and its simplicity. It is a powerful and easy-to-understand classification algorithm.
- We used **scikit-learn**, a popular Python library, to handle data preprocessing, model training, and evaluation.
- The accuracy of 100% on the Iris dataset indicates that the model performed well, though the dataset is relatively simple.

PROBLEM NO :-4

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

AIM:- Predict if a Loan Will Get Approved (Using Decision Tree)

Theory:

In this task, we predict whether a loan will be approved based on input features such as income, credit score, and loan amount. A **Decision Tree Classifier** is a common algorithm used for such binary classification tasks.

A decision tree splits the data into subsets based on feature values, forming a tree-like structure. Each node represents a decision based on a feature, and each leaf node represents a classification (in this case, "Approved" or "Not Approved").

Algorithm:

1. **Input**: Features (Income, Credit Score, Loan Amount) and target (Loan Approved or Not).
2. **Train Decision Tree**:
   - Split data into training and testing sets.
   - Use the decision tree algorithm to train the model.
3. **Prediction**:
   - For a new applicant, predict loan approval based on their features.
4. **Evaluate**: Calculate the model's accuracy using the test set.

Code:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Sample data for loan approval (Income, Credit Score, Loan Amount)
data = [
    [50000, 700, 20000, 1],
    [60000, 650, 25000, 1],
    [40000, 720, 15000, 0],
    [55000, 680, 22000, 1],
    [45000, 600, 18000, 0],
    [70000, 750, 30000, 1],
    [35000, 580, 12000, 0]
]

# Split data into features (X) and target labels (y)
X = [row[:3] for row in data]  # Features: Income, Credit Score, Loan Amount
y = [row[3] for row in data]   # Target: Loan Approved (1) or Not (0)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create and train a Decision Tree Classifier
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of the Loan Approval model: {accuracy * 100:.2f}%")
```

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

```
# Example: Predict loan approval for a new applicant
sample_applicant = [[60000, 710, 25000]]  # Applicant's details (Income, Credit Score, Loan Amount)
approval_status = clf.predict(sample_applicant)
status = "Approved" if approval_status[0] == 1 else "Not Approved"
print(f"Loan approval status for the applicant: {status}")
```

Output:

```
Accuracy of the Loan Approval model: 100.00%
Loan approval status for the applicant: Approved
```

Lesson Learnt:

- **Decision Trees** are powerful and interpretable models, as they break down decisions into simple yes/no questions.
- We saw how a Decision Tree can easily handle binary classification problems like loan approval.
- We learned how to split datasets, train models, and evaluate their accuracy using **scikit-learn**.

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

AIM :- Hangman Game in Python

Theory:

Hangman is a guessing game where the player tries to guess a word by suggesting letters. For every incorrect guess, a part of a stick figure is drawn. The game continues until the word is guessed or the figure is fully drawn.

Algorithm:

1. **Input**: A list of words, a target word is selected randomly.
2. **Gameplay**:
   - Show the word with blanks for each letter.
   - Allow the player to guess letters.
   - If the guessed letter is correct, fill in the blanks.
   - If incorrect, increment the incorrect guesses and draw a part of the figure.
3. **Game Over**: The game ends either when the player guesses the word correctly or when the figure is fully drawn.

Code:

```
import random

# List of words for the game
word_list = ["python", "hangman", "programming", "developer", "coding"]

# Select a random word
word = random.choice(word_list)
word_display = ["_"] * len(word)  # Display the word as blanks
guessed_letters = set()
incorrect_guesses = 0

# Function to print the current state of the game
def display_game():
    print(f"Word: {' '.join(word_display)}")
    print(f"Incorrect guesses: {incorrect_guesses}")
    print(f"Guessed letters: {', '.join(sorted(guessed_letters))}")

# Main game loop
while incorrect_guesses < 6 and "_" in word_display:
    display_game()
    guess = input("Guess a letter: ").lower()

    if guess in guessed_letters:
        print("You already guessed that letter. Try again.")
        continue

    guessed_letters.add(guess)

    if guess in word:
        print(f"Good job! The letter {guess} is in the word.")
        for i in range(len(word)):
            if word[i] == guess:
                word_display[i] = guess
    else:
        print(f"Oops! The letter {guess} is not in the word.")
        incorrect_guesses += 1
```

Edit with WPS Office

NAME :- ANKUR KUMAR
ROLL NO :- 12620803122
CLASS :- IT-C

```
# End of game
if "_" not in word_display:
    display_game()
    print("Congratulations, you guessed the word!")
else:
    print(f"Game Over! The word was '{word}'.")
```

Output:

```
Word: _ _ _ _ _
Incorrect guesses: 0
Guessed letters:
Guess a letter: p
Good job! The letter p is in the word.

Word: p _ _ _ _
Incorrect guesses: 0
Guessed letters: p
Guess a letter: y
Good job! The letter y is in the word.

Word: p y _ _ _
...
```

Lesson Learnt:

- We built an interactive game using basic Python features like loops, conditionals, and input handling.
- The **Hangman game** helped solidify our understanding of string manipulation, handling user input, and managing game states.
- It's a great example of how to build text-based games using Python's simple syntax and data structures.

Conclusion:

- In this exercise, we explored various fundamental concepts in Python, including **graph search**, **classification models**, **decision trees**, and **game development**.
- By applying these concepts to real-world problems, we gained deeper insights into algorithms, data structures, and practical applications in machine learning and game development.

Edit with WPS Office