Experiment no: 3

Breath First Search

**Aim:** To implement BFS algorithm

**Theory:** Breadth-First Search (BFS) is a fundamental graph traversal algorithm used to explore or traverse a graph or tree data structure. Unlike Depth-First Search (DFS), which explores as far as possible along a branch before backtracking, BFS explores the graph level by level.

The basic process of the Breadth-First Search is as follows:

1. Starting Point: The algorithm begins at a selected node as the starting point.

2. Exploring Adjacent Nodes: It visits all the neighboring nodes of the starting node at the current level before moving to the nodes at the next level.

3. Queue: BFS uses a queue data structure to keep track of the nodes to visit. Nodes are added to the queue as they are discovered and processed in the order they were added (FIFO order).

4. Level-wise Exploration: It explores the graph level by level. In each level, it visits all nodes adjacent to the current nodes before moving to the next level.

5. Marking Visited Nodes: Typically, nodes are marked as visited to prevent revisiting nodes and infinite loops in graphs with cycles.

**Code:**

from queue import Queue

def BFS(vertices, edges, start, end):

assert start in vertices, "Start must be a vertex"

assert end in vertices, "End must be a vertex"

def print\_sol(current, start):

if current == start:

print(f"{current} ", end="")

return

print\_sol(parent[current], start)

print(f"-> {current} ", end="")

return

parent = {}

q = Queue()

q.put(start)

while not q.empty():

current = q.get()

if current in parent.values():

continue

print(f"{current} visited!")

if current == end:

print("End reached!")

break

for next in edges[current]:

if next not in parent.keys():

parent[next] = current

q.put(next)

if end in parent:

print\_sol(end, start)

Gvertices = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J",

"K"}

Gedges = {

"A": ["B", "C", "D"],

"B": ["A", "E", "F", "C"],

"C": ["A", "B", "F", "G"],

"D": ["A", "G"],

"E": ["B", "H"],

"F": ["B", "C", "G", "H", "I"],

"G": ["C", "F", "D", "I"],

"H": ["E", "F", "I", "K", "J"],

"I": ["F", "G", "H", "K"],

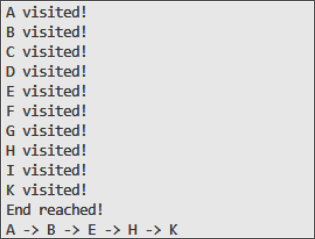
"J": ["H"],

"K": ["H", "I"]

}

BFS(Gvertices, Gedges, "A", "K")

**Output:**



**Conclusion:** The BFS algorithm was studied and implemented in Python.

Experiment no: 4

Depth First Search

**Aim:** To implement DFS algorithm

**Theory:** Depth-First Search (DFS) is a fundamental graph traversal algorithm used to explore or traverse a graph or tree data structure. It operates by visiting as far as possible along a branch or path and then backtracking.

The basic process of the Depth-First Search is as follows:

1. Starting Point: The algorithm begins at a selected node as the starting point.

2. Visiting Nodes: It visits the neighboring nodes of the starting node and explores as far as possible along each branch before backtracking.

3. Stack (or Recursion): DFS uses a stack to keep track of the nodes to visit. Alternatively, it can be implemented using recursion.

4. Backtracking: When there are no more unvisited adjacent nodes, the algorithm backtracks to the most recently visited node that has unexplored neighbors and continues the exploration.

5. Marking Visited Nodes: It typically marks nodes as visited to avoid infinite loops in graphs with cycles.

**Code:**

def DFS(vertices, edges, start, end):

assert start in vertices, "Start must be a vertex"

assert end in vertices, "End must be a vertex"

def print\_sol(current, start):

if current == start:

print(f"{current} ", end="")

return

print\_sol(parent[current], start)

print(f"-> {current} ", end="")

def dfs\_helper(current):

visited.add(current)

print(f"{current} visited!")

if current == end:

print("End reached!")

return True

for next\_vertex in edges[current]:

if next\_vertex not in visited:

parent[next\_vertex] = current

if dfs\_helper(next\_vertex):

return True

return False

parent = {}

visited = set()

dfs\_helper(start)

if end in parent:

print\_sol(end, start)

# Example usage

Gvertices = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J",

"K"}

Gedges = {

"A": ["B", "C", "D"],

"B": ["A", "E", "F", "C"],

"C": ["A", "B", "F", "G"],

"D": ["A", "G"],

"E": ["B", "H"],

"F": ["B", "C", "G", "H", "I"],

"G": ["C", "F", "D", "I"],

"H": ["E", "F", "I", "K", "J"],

"I": ["F", "G", "H", "K"],

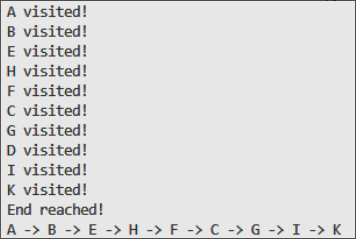
"J": ["H"],

"K": ["H", "I"]

}

DFS(Gvertices, Gedges, "A", "K")

**Output:**



**Conclusion:** The DFS algorithm was studied and implemented in Python.

Experiment no: 5

Decision Trees

**Aim:** To implement Decision trees.

**Theory:** Decision Trees are predictive models used in machine learning and data analysis for making decisions by learning simple decision rules inferred from data features. They are a popular supervised learning method that can perform both classification and regression tasks.

During training, the Decision Tree algorithm selects the best attribute to split the data based on a metric such as entropy or Gini impurity, which measures the level of impurity or randomness in the subsets. The goal is to find the attribute that maximizes the information gain or the reduction in impurity after the split.

**Construction of Decision Tree:** A tree can be “learned” by splitting the source set into subsets based on Attribute Selection Measures. Attribute selection measure (ASM) is a criterion used in decision tree algorithms to evaluate the usefulness of different attributes for splitting a dataset. The goal of ASM is to identify the attribute that will create the most homogeneous subsets of data after the split, thereby maximizing the information gain. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a

node all has the same value of the target variable, or when splitting no longer adds value to the predictions. The construction of a decision tree classifier does not require any domain knowledge or parameter setting and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high-dimensional data.