EXPERIMENT – 9

Aim:

Implement bi-connected components and strongly connected components.

Description:

A bi-connected component (BCC) in an undirected graph is a maximal subgraph such that removing any single vertex does not disconnect the subgraph. It's useful in understanding the resilience of a network; for example, a network is more robust if it has more bi-connected components.

A strongly connected component (SCC) in a directed graph is a maximal subgraph where every vertex is reachable from every other vertex within the subgraph. SCCs are useful for identifying isolated parts of a directed graph.

Algorithm:

1. **BCC:**

- Perform a Depth-First Search (DFS) on the graph.
- Use a stack to keep track of the vertices in the current DFS path.
- Track discovery and low values for each vertex.
- Discovery time represents the time at which the vertex was first visited.
- Low value represents the smallest discovery time reachable from that vertex.
- For each vertex, check if it forms a biconnected component with its DFS child nodes.
- If a vertex's low value is higher than the discovery time of its parent, it indicates the formation of a new BCC.

2. **SCC:**

- Perform a DFS on the graph and record the finish time for each vertex.
- Reverse the direction of all edges in the graph.
- Perform a DFS on the reversed graph in the order of decreasing finish times.
- Each DFS tree in the reversed graph represents a strongly connected component.

Code for BCC:

```python
class BiconnectedComponents:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[] for _ in range(vertices)]
        self.time = 0
        self.stack = []

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bcc_util(self, u, discovery, low, parent):
        discovery[u] = low[u] = self.time
        self.time += 1
        children = 0

        for v in self.graph[u]:
            if discovery[v] == -1:
                parent[v] = u
                children += 1
                self.stack.append((u, v))

                self.bcc_util(v, discovery, low, parent)
                low[u] = min(low[u], low[v])

                if (parent[u] == -1 and children > 1) or (parent[u] != -1 and low[v] >= discovery[u]):
                    bcc = []
                    while self.stack[-1] != (u, v):
                        bcc.append(self.stack.pop())
                    bcc.append(self.stack.pop())
                    print("Bi-connected Component:", bcc)

            elif v != parent[u] and discovery[v] < discovery[u]:
                low[u] = min(low[u], discovery[v])
                self.stack.append((u, v))

    def find_bcc(self):
        discovery = [-1] * self.V
        low = [-1] * self.V
        parent = [-1] * self.V

        for i in range(self.V):
            if discovery[i] == -1:
                self.bcc_util(i, discovery, low, parent)

        if self.stack:
            bcc = []
            while self.stack:
                bcc.append(self.stack.pop())
            print("Bi-connected Component:", bcc)

# Example Usage:
bcc_graph = BiconnectedComponents(5)
bcc_graph.add_edge(0, 1)
bcc_graph.add_edge(1, 2)
bcc_graph.add_edge(2, 0)
bcc_graph.add_edge(1, 3)
bcc_graph.add_edge(3, 4)
bcc_graph.find_bcc()
```

Output:

```
[Running] python -u "c:\My learnings\python\tempCodeRunnerFile.python"
Bi-connected Component: [(3, 4)]
Bi-connected Component: [(1, 3)]
Bi-connected Component: [(2, 0), (1, 2), (0, 1)]
```

Code for SCC:

```
class StronglyConnectedComponents:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[] for _ in range(vertices)]

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if not visited[i]:
                self.dfs(i, visited, stack)
        stack.append(v)

    def reverse_graph(self):
        reversed_graph = [[] for _ in range(self.V)]
        for u in range(self.V):
            for v in self.graph[u]:
                reversed_graph[v].append(u)
        return reversed_graph

    def fill_order(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if not visited[i]:
                self.fill_order(i, visited, stack)
        stack.append(v)

    def dfs_util(self, graph, v, visited):
        visited[v] = True
        print(v, end=' ')
        for i in graph[v]:
            if not visited[i]:
                self.dfs_util(graph, i, visited)

    def find_scc(self):
        stack = []
        visited = [False] * self.V

        for i in range(self.V):
            if not visited[i]:
                self.fill_order(i, visited, stack)

        reversed_graph = self.reverse_graph()
        visited = [False] * self.V

        while stack:
            i = stack.pop()
            if not visited[i]:
                self.dfs_util(reversed_graph, i, visited)
                print("")

# Example Usage:
scc_graph = StronglyConnectedComponents(5)
scc_graph.add_edge(0, 2)
scc_graph.add_edge(2, 1)
scc_graph.add_edge(1, 0)
scc_graph.add_edge(0, 3)
scc_graph.add_edge(3, 4)
print("Strongly Connected Components:")
scc_graph.find_scc()
```

Output:

```
[Running] python -u "c:\My learnings\python\tempCodeRunnerFile.python"
Strongly Connected Components:
0 1 2
3
4
```

Result:

- Bi-Connected Components (BCC): [(2, 0), (1, 2), (0, 1)], [(3, 4)], [(1, 3)]
- Strongly Connected Components (SCC): {0, 1, 2}, {3}, {4}

EXPERIMENT – 10

Aim:

Implement Dijkstra's, Bellman-Ford, Floyd-Warshall.

Description:

Dijkstra's algorithm finds the shortest path from a single source to all other vertices in a weighted graph with non-negative weights. It is commonly used in network routing and pathfinding problems.

The **Bellman-Ford algorithm** finds the shortest path from a single source to all other vertices, even if the graph has negative weights. It's slower than Dijkstra's algorithm but works with negative weights.

The **Floyd-Warshall algorithm** finds the shortest paths between all pairs of vertices in a weighted graph, including those with negative weights. It uses dynamic programming and can detect negative cycles.

Algorithm:

1.Dijkstra's algorithm:

1. Initialize the distance to the source as 0 and all others as infinity.
2. Set all nodes as unvisited and track the minimum distance from the source.
3. For the current node, check its unvisited neighbors and update their distances if a shorter path is found.
4. Mark the current node as visited and move to the nearest unvisited node.
5. Repeat until all nodes have been visited.

2.Bellman-Ford algorithm:

1. Initialize the distance to the source as 0 and all other vertices as infinity.
2. Relax all edges V-1 times (where V is the number of vertices).
3. For each edge (u, v), if the distance to v is greater than the distance to u plus the edge weight, update the distance to v.
4. Check for negative-weight cycles by verifying if any edge can still be relaxed.

3.Floyd's-Warshall algorithm:

1. Initialize a matrix dist where dist[i][j] is the weight of the edge between vertices i and j.
2. For each vertex k, update the distances so that dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).

3. Repeat until all pairs of shortest paths are updated.

Code (Dijkstra's algorithm):

```python
class Dijkstra:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v, weight):
        self.graph[u][v] = weight
        self.graph[v][u] = weight  # Remove this line for directed graphs

    def min_distance(self, dist, visited):
        min_val = float('inf')
        min_index = -1
        for v in range(self.V):
            if not visited[v] and dist[v] < min_val:
                min_val = dist[v]
                min_index = v
        return min_index

    def dijkstra(self, src):
        dist = [float('inf')] * self.V
        dist[src] = 0
        visited = [False] * self.V

        for _ in range(self.V):
            u = self.min_distance(dist, visited)
            visited[u] = True

            for v in range(self.V):
                if self.graph[u][v] and not visited[v] and dist[u] + self.graph[u][v] < dist[v]:
                    dist[v] = dist[u] + self.graph[u][v]

        return dist

# Example Usage
dijkstra_graph = Dijkstra(5)
dijkstra_graph.add_edge(0, 1, 10)
dijkstra_graph.add_edge(0, 4, 5)
dijkstra_graph.add_edge(1, 2, 1)
dijkstra_graph.add_edge(2, 3, 4)
dijkstra_graph.add_edge(4, 1, 3)
dijkstra_graph.add_edge(4, 2, 9)
dijkstra_graph.add_edge(4, 3, 2)
result = dijkstra_graph.dijkstra(0)
print("Shortest distances from source 0:", result)
```

Output:

```
[Running] python -u "c:\My learnings\python\tempCodeRunnerFile.python"
Shortest distances from source 0: [0, 8, 9, 7, 5]
```

Code (Bellman-Ford algorithm):

```
class BellmanFord:
    def __init__(self, vertices):
        self.V = vertices
        self.edges = []

    def add_edge(self, u, v, weight):
        self.edges.append((u, v, weight))

    def bellman_ford(self, src):
        dist = [float('inf')] * self.V
        dist[src] = 0

        for _ in range(self.V - 1):
            for u, v, weight in self.edges:
                if dist[u] != float('inf') and dist[u] + weight < dist[v]:
                    dist[v] = dist[u] + weight

        for u, v, weight in self.edges:
            if dist[u] != float('inf') and dist[u] + weight < dist[v]:
                print("Graph contains a negative-weight cycle")
                return None

        return dist

# Example Usage
bf_graph = BellmanFord(5)
bf_graph.add_edge(0, 1, -1)
bf_graph.add_edge(0, 2, 4)
bf_graph.add_edge(1, 2, 3)
bf_graph.add_edge(1, 3, 2)
bf_graph.add_edge(1, 4, 2)
bf_graph.add_edge(3, 2, 5)
bf_graph.add_edge(3, 1, 1)
bf_graph.add_edge(4, 3, -3)
result = bf_graph.bellman_ford(0)
print("Shortest distances from source 0:", result)
```

Output:

```
[Running] python -u "c:\My learnings\python\tempCodeRunnerFile.python"
Shortest distances from source 0: [0, -1, 2, -2, 1]
```

Code (Floyd's -Warshall algorithm):

```
class FloydWarshall:
    def __init__(self, vertices):
        self.V = vertices
        self.dist = [[float('inf')] * vertices for _ in range(vertices)]
        for i in range(vertices):
            self.dist[i][i] = 0

    def add_edge(self, u, v, weight):
        self.dist[u][v] = weight

    def floyd_warshall(self):
        for k in range(self.V):
            for i in range(self.V):
                for j in range(self.V):
                    if self.dist[i][k] + self.dist[k][j] < self.dist[i][j]:
                        self.dist[i][j] = self.dist[i][k] + self.dist[k][j]

        return self.dist

# Example Usage
fw_graph = FloydWarshall(4)
fw_graph.add_edge(0, 1, 3)
fw_graph.add_edge(0, 2, 5)
fw_graph.add_edge(1, 2, -2)
fw_graph.add_edge(2, 3, 1)
fw_graph.add_edge(3, 0, 2)
result = fw_graph.floyd_warshall()
print("All pairs shortest paths:")
for row in result:
    print(row)
```

Output:

```
[Running] python -u "c:\My learnings\python\tempCodeRunnerFile.python"
All pairs shortest paths:
[0, 3, 1, 2]
[1, 0, -2, -1]
[3, 6, 0, 1]
[2, 5, 3, 0]
```

Result:

These implementations provide solutions for shortest path algorithms suited to different types of graph constraints. Each algorithm's result shows the shortest path values from source vertices or between all pairs.

Experiment – 9

Aim:

Demonstration of clustering algorithms- k-Means, Agglomerative and DBSCAN to classify for the standard datasets.

Description:

k-Means clustering is a partition-based clustering algorithm that divides the data into k clusters, where each data point belongs to the cluster with the nearest mean. The objective is to minimize the variance within each cluster.

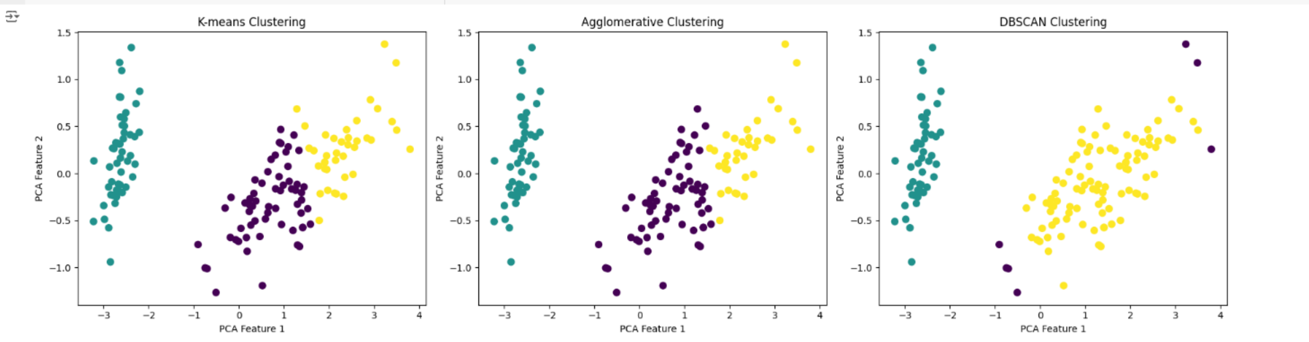
Agglomerative clustering is a type of hierarchical clustering that starts with each data point as an individual cluster and iteratively merges the closest clusters until only a single cluster remains or a specified number of clusters is reached. The merging decision is based on linkage criteria (e.g., single, complete, average).

DBSCAN is a density-based clustering algorithm that groups data points if they are within a certain distance (epsilon) from each other, forming clusters with varying shapes. It also identifies noise (outliers) that doesn't belong to any cluster. DBSCAN requires two parameters: epsilon (maximum distance between points in a cluster) and min_samples (minimum points required to form a dense region).

Program:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN
from sklearn.decomposition import PCA

# Load the Iris dataset
iris = load_iris()
X = iris.data

# Reduce dimensions for visualization
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)

# Define number of clusters for K-means and Agglomerative
n_clusters = 3

# K-means clustering
kmeans = KMeans(n_clusters=n_clusters, random_state=0)
kmeans_labels = kmeans.fit_predict(X_reduced)

# Agglomerative clustering
agglo = AgglomerativeClustering(n_clusters=n_clusters)
agglo_labels = agglo.fit_predict(X_reduced)

# DBSCAN clustering
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan_labels = dbscan.fit_predict(X_reduced)

# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(18, 5))

# K-means plot
axs[0].scatter(X_reduced[:, 0], X_reduced[:, 1], c=kmeans_labels, s=50, cmap='viridis')
axs[0].set_title('K-means Clustering')
axs[0].set_xlabel('PCA Feature 1')
axs[0].set_ylabel('PCA Feature 2')

# Agglomerative plot
axs[1].scatter(X_reduced[:, 0], X_reduced[:, 1], c=agglo_labels, s=50, cmap='viridis')
axs[1].set_title('Agglomerative Clustering')
axs[1].set_xlabel('PCA Feature 1')
axs[1].set_ylabel('PCA Feature 2')

# DBSCAN plot
axs[2].scatter(X_reduced[:, 0], X_reduced[:, 1], c=dbscan_labels, s=50, cmap='viridis')
axs[2].set_title('DBSCAN Clustering')
axs[2].set_xlabel('PCA Feature 1')
axs[2].set_ylabel('PCA Feature 2')

plt.tight_layout()
plt.show()
```

Output:



Result:

- **k-Means Clustering:** Groups data into k clusters based on distance from centroids, creating spherical clusters that minimize variance within each group.
- **Agglomerative Clustering:** Uses a hierarchical approach to progressively merge data points into clusters, forming a tree-like structure until the desired number of clusters is reached.
- **DBSCAN Clustering:** Identifies clusters based on density, forming clusters of various shapes and marking low-density points as noise or outliers.