DS2030 Data Structures and Algorithms for Data Science
Lab 10

November 4th, 2025

Lab Instructions

- Create a folder named **“DS2030_<RollNo.>”** (all letters in capital) in **“home”** directory. Eg- **DS2030_142402022**
- Name the script files in the given format **“<your_roll_no>_<Name>_Lab10.py”**
- Make sure the **folder, files, classes, functions and attributes** are named as instructed in the lab sheet.
- We will not be able to evaluate your work if the folder is not properly named or is not located in the home directory.
- Make sure to save your progress before leaving the lab.
- Do not shut down the system after completing the lab.
- You are not allowed to share code with your classmates nor allowed to use code from other sources.

Graph Connectivity and Bipartiteness

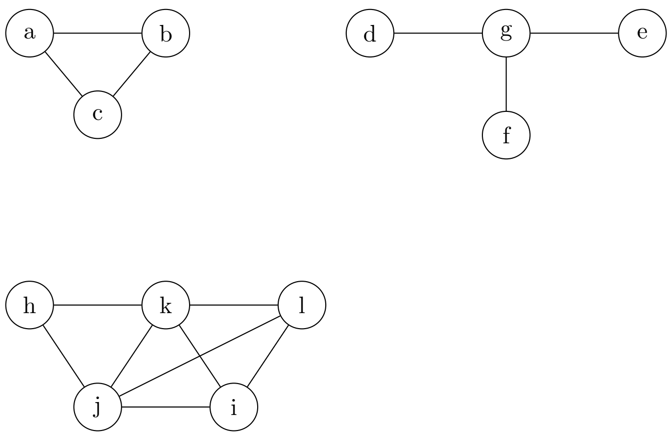
Problem Statement

Graphs are fundamental data structures used to represent relationships between entitiessuch as friendships in social networks, data links in communication systems, or molecular interactions in biology. Analyzing a graph often requires determining which vertices are mutually reachable and understanding the structure of these connected regions.
Your task is to analyze a given undirected graph and:

1. Identify all **connected components** in the graph.
2. Determine whether each component is **bipartite** (that is, whether vertices can be colored with two colors so that no two connected vertices share the same color).

Both functionalities must be implemented using the **Breadth-First Search (BFS)** traversal algorithm.

Example Graph



This graph consists of **three connected components**:

$$\{a, b, c\}, \quad \{d, e, f, g\}, \quad \{h, i, j, k, l\}$$

Each component should be identified and tested for bipartiteness by your program.

Tasks

1. **Build the Graph**

   Implement a function to construct an adjacency list representation of the graph.

   ```python
   def build_graph(n, edges):
   ```

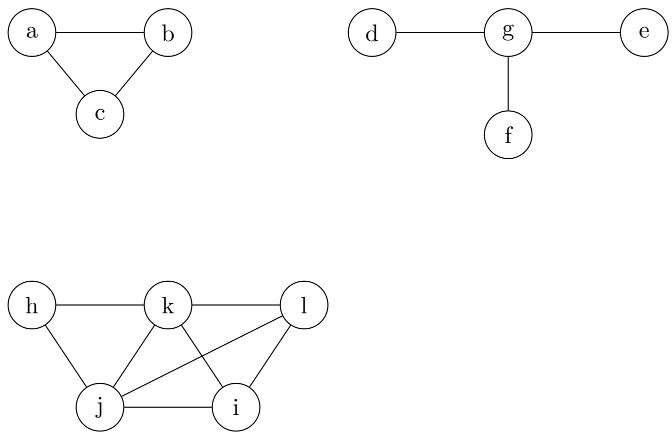
   Input:

   - `n`: total number of vertices (labeled 0 to n-1 or a to l)
   - `edges`: list of tuples (u, v) representing undirected edges

   Output:

   - A dictionary mapping each vertex to a list of its neighbors.

   ### Example Graph

   This graph consists of **three connected components**:

   $$\{a, b, c\}, \quad \{d, e, f, g\}, \quad \{h, i, j, k, l\}$$

   Your program should correctly identify each of these components.

2. **Find Connected Components (Using BFS)**

   Implement a function to find all connected components using **Breadth-First Search (BFS)**.

   ```python
   def connected_components(graph):
   ```

   Algorithm Outline:

   (a) Maintain a set `visited`.
   (b) For every unvisited vertex, perform a BFS traversal.
   (c) Collect all reachable vertices as one connected component.
   (d) Repeat until all vertices are visited.

Example

For the above graph, the output should be:

$$\{a, b, c\}, \quad \{d, e, f, g\}, \quad \{h, i, j, k, l\}$$

3. **Check if a Graph is Bipartite(Using BFS Coloring)**

Implement a function to test if a given connected component of the graph is **bipartite**.

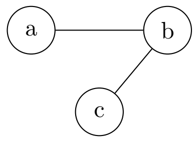
A bipartite graph is one whose vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent.

```
def is_bipartite(graph, start):
    """
    Parameters:
        graph (dict): adjacency list
        start (str): starting vertex
    Returns:
        bool: True if the graph is bipartite, False otherwise
    """
```

Idea:

- Assign two alternating colors (e.g., Red and Blue) using BFS.
- If any edge connects vertices of the same color, the component is not bipartite.

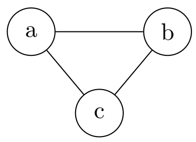
Example 1 Bipartite Graph



This graph is **bipartite**, because we can divide its vertices into sets

$$U = \{a, c\}, \quad V = \{b\}$$

Example 2 Non-Bipartite Graph



This graph is **not bipartite**, because it contains an odd-length cycle.

4. **Combine Both Functionalities**

For each connected component in the graph:

- Print the vertices in that component.
- Check whether that component is bipartite.
- Print `Bipartite` or `Not Bipartite` accordingly.

Example Output

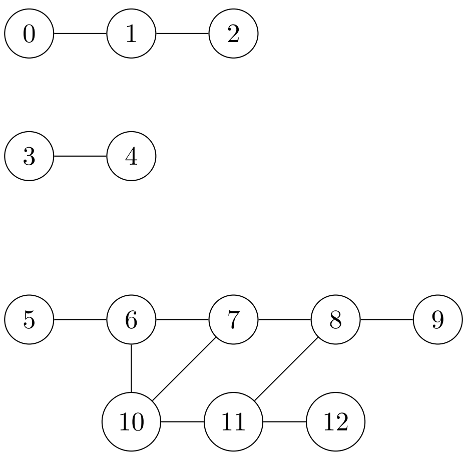
```
Component 1: {a, b, c}
Not Bipartite

Component 2: {d, e, f, g}
Bipartite

Component 3: {h, i, j, k, l}
Bipartite
```

Starter Code

```python
# In this lab, you will:
# 1. Build an undirected graph using adjacency list representation.
# 2. Use BFS to identify all connected components.
# 3. For each component, check whether it is bipartite.
#
# HINTS:
# - Use a queue (FIFO) structure for BFS.
# - Use dictionaries or sets to keep track of visited nodes and colors.
# - Remember that an undirected edge (u, v) implies connections both ways.
# ===============================================================


# ---------------------------------------------------------------
# TASK 1: Build the Graph
# ---------------------------------------------------------------
def build_graph(n, edges):
    """
    Build an undirected graph as an adjacency list.

    Parameters:
        n (int): Number of vertices (0 to n-1)
        edges (list of tuples): Each tuple (u, v) represents an undirected edge.

    Returns:
        dict: A dictionary mapping each vertex to its list of neighbors.

    Directions:
    1. Initialize a dictionary with all vertices (0 to n-1) mapped to empty lists.
    2. For every edge (u, v):
       - Add v to the adjacency list of u.
       - Add u to the adjacency list of v (since the graph is undirected).
    3. Return the adjacency list.
    """
    pass


# ---------------------------------------------------------------
# TASK 2: Find Connected Components using BFS
# ---------------------------------------------------------------
def connected_components(graph):
    """
    Find all connected components using BFS traversal.

    Parameters:
        graph (dict): Adjacency list of the graph.

    Returns:
        list: A list of connected components (each component is a list of vertices).

    Directions:
    1. Initialize an empty set 'visited' to track visited vertices.
    2. For each vertex in the graph:
       - If it has not been visited, start a BFS from it.
       - Collect all vertices reachable from it as one component.
    3. Append this component to the list of components.
    4. Return the list of components.
    """
    pass


# ---------------------------------------------------------------
# TASK 3: Check if a Component is Bipartite using BFS Coloring
# ---------------------------------------------------------------
def is_bipartite(graph, component):
```

```
    """
    Check whether the given connected component is bipartite using BFS.

    Parameters:
        graph (dict): Adjacency list of the graph.
        component (list): Vertices belonging to one connected component.

    Returns:
        bool: True if the component is bipartite, False otherwise.

    Directions:
    1. Initialize a dictionary 'color' to store colors of vertices (0 or 1).
    2. For each uncolored vertex in the component:
       - Assign a starting color (say 0).
       - Perform BFS using a queue.
         - For each neighbor:
             - If not colored, assign opposite color and continue.
             - If colored with the same color as the current vertex     Not bipartite.
    3. If BFS completes without conflict, return True.
    """
    pass


# --------------------------------------------------------------
# TASK 4: Display Results
# --------------------------------------------------------------
def analyze_graph(graph):
    """
    Analyze the graph:
    - Identify connected components.
    - Check bipartiteness of each.

    Directions:
    1. Use connected_components() to get all components.
    2. For each component:
       - Print the list of vertices.
       - Call is_bipartite() and print True/False accordingly.
    """
    pass
```

The graph shown below is the input used in the test function. It serves as a sample structure to verify the correctness of graph construction, connected component identification, and bipartite testing.



Test Function

```python
def test_graph_connectivity():
    """
    Test the functionalities on a sample graph.

    Graph layout (as visualized below):

    Description:
        - Component 1: {0,1,2}     Bipartite
        - Component 2: {3,4}     Bipartite
        - Component 3: {5  12}     NOT Bipartite (odd cycle)
    """
    n = 13
    edges = [
        (0, 1), (1, 2),                    # Component 1
        (3, 4),                            # Component 2
        (5, 6), (6, 7), (7, 8), (8, 9),    # Chain in Component 3
        (6, 10), (10, 11), (11, 12),
        (8, 11), (10, 7)                   # Odd cycle connections
    ]

    graph = build_graph(n, edges)

    print("Graph Representation (Adjacency List):")
    for node in sorted(graph.keys()):
        print(f"{node}: {sorted(graph[node])}")

    print("\n============================")
    print("Connected Components and Bipartiteness")
    print("============================")
    analyze_graph(graph)


if __name__ == "__main__":
    test_graph_connectivity()
```

Sample Output

```
Graph Representation (Adjacency List):
0: [1]
1: [0, 2]
2: [1]
3: [4]
4: [3]
5: [6]
6: [5, 7, 10]
7: [6, 8, 10]
8: [7, 9, 11]
9: [8]
10: [6, 7, 11]
11: [8, 10, 12]
12: [11]

============================
Connected Components and Bipartiteness
============================
Component 1: [0, 1, 2]     Bipartite: True
Component 2: [3, 4]     Bipartite: True
Component 3: [5, 6, 7, 8, 9, 10, 11, 12]     Bipartite: False
```