Assignment: Identifying Critical Nodes in a Computer Network Using DFS

Course: Algorithms: Design & Analysis

March 17, 2025

1 Problem Statement

A critical node (articulation point) in a computer network is a node whose removal increases the number of connected components. Such nodes are crucial in network resilience analysis. In this assignment, you will:

1. Model a computer network as an undirected graph, where nodes represent routers/servers, and edges represent connections.
2. Implement an algorithm to detect articulation points using Depth-First Search (DFS).
3. Simulate network failure by removing a critical node and analyzing the resulting connectivity.

2 Input Format

The input file (`network_<file number>.txt`) will contain:

- An integer V (number of nodes) and E (number of edges).
- E lines, each containing two integers u, v indicating an undirected edge between u and v.

Example Input:

```
6 7
0 1
0 2
1 3
2 3
3 4
3 5
4 5
```

3 Output Format

- A list of articulation points (if any).
- A network failure simulation in which one articulation point is removed, showing the resulting disconnected components.

Example Output:

```
Critical Nodes (Articulation Points): 3
Removing node 3 will disconnect the network into two components: [0, 1, 2] and [4, 5]
```

4 Algorithm Requirements

Your implementation should:

- Use DFS traversal to compute the discovery time (disc []) and the lowest reachable ancestor (low[]). So we maintain an additional array low[] such that: low[u] = min(disc[u], disc[w]) , Here w is an ancestor of u and there is a back edge from some descendant of u to w.
- Identify nodes that satisfy the conditions of the articulation point:
  - A root with two or more children.
  - A non-root node where low[child]≥disc[node].
- Simulate network failure by removing an articulation point and computing new connected components.

5 Implementation Details

Use **Python** for implementation. Your program should:

- Read input from files named `network_<file number>.txt` located in the **network** folder (`https://tinyurl.com/2p9n3hvb`).
- Compute **articulation points** using **Depth-First Search (DFS)**.
- Simulate and output the effect of **network failures** caused by removing articulation points.

Ensure that your program dynamically processes all files in the **network** directory, handling multiple input files systematically.

6 Evaluation Criteria

Your assignment will be graded based on:

- **Correctness (50%)**: Identifies articulation points accurately using DFS.
- **Simulation (30%)**: Visualize the effect of **network failures** caused by removing articulation points.
- **Code Readability (20%)**: Well-commented and structured code.

7 Submission Guidelines

- Submit your Python script as `network_analysis_<Your name>.py`.
- Provide a short report (`report.pdf`) covering:
  - Your approach to solving the problem.
  - Challenges faced and resolutions.
  - Example output from your program.

8 Expected Learning Outcomes

By completing this assignment, you will:

- Understand network resilience and single points of failure.
- Implement Tarjan's Algorithm for articulation point detection.
- Simulate and analyze network failures.

Assignment: Identifying Critical Nodes in a Computer Network Using DFS

Course: Algorithms: Design & Analysis

bq08283

1 Introduction

We start by identifying articulation points in an undirected graph using **Tarjan's algorithm**. Articulation points are nodes that, when removed, increase the graph's number of connected components. After identifying these critical nodes, we delete them from the original network and calculate the resulting connected components. Furthermore, we address edge cases where no articulation points are located, ensuring that the application returns a clear message indicating that the network is still connected.

2 Approach

2.1 Graph Input

Code Snippet: Reading Input

```
for i in range(1, 9):
    file_name = f'network_{i}.txt'
    print(f"\nProcessing {file_name}:")

    with open(file_name, 'r') as file:
        V, E = map(int, file.readline().split())
        graph = nx.Graph()

        # Reading edges from the input file
        for _ in range(E):
            u, v = map(int, file.readline().split())
            graph.add_edge(u, v)
```

2.2 Articulation Points Detection

A DFS-based algorithm (Tarjan's algorithm) was implemented to find articulation points.

DFS-based algorithm (Tarjan's algorithm)

```python
def dfs(u, graph, parent, disc, low, visited, articulation_points, time):
    visited[u] = True
    disc[u] = low[u] = time
    time += 1
    children = 0

    for v in graph[u]:
        if not visited[v]:
            parent[v] = u
            children += 1
            dfs(v, graph, parent, disc, low, visited, articulation_points, time)

            low[u] = min(low[u], low[v])

            # Condition 1: A root with two or more children
            if parent[u] == -1 and children > 1:
                articulation_points.add(u)

            # Condition 2: A non-root node where low[child] ≥ disc[node].
            if parent[u] != -1 and low[v] >= disc[u]:
                articulation_points.add(u)

        elif v != parent[u]:  # here v is an ancestor of u and there is a back edge from some descendant of u to v
            low[u] = min(low[u], disc[v])
```

A node u is identified as an articulation point if,

1. **It is the root of the DFS tree and has more than one child.** This is because removing the root with several children will split the tree into distinct subtrees.

2. **It is not the root, and there exists a child v such that $low[v] \geq disc[u]$.** This condition assures that no back edge connects v or its descendants to an ancestor of u. Therefore, u is crucial for maintaining connection.

Code Snippet: Articulation Point Conditions

```python
# Condition 1
if parent[u] == -1 and children > 1:
    articulation_points.add(u)

# Condition 2
if parent[u] != -1 and low[v] >= disc[u]:
    articulation_points.add(u)
```

3 Cases

3.1 Case 1: Graph with Articulation Points

In this example, the graph includes one or more articulation points. Removing these points increases the graph's connected components. For example, take the graph below,

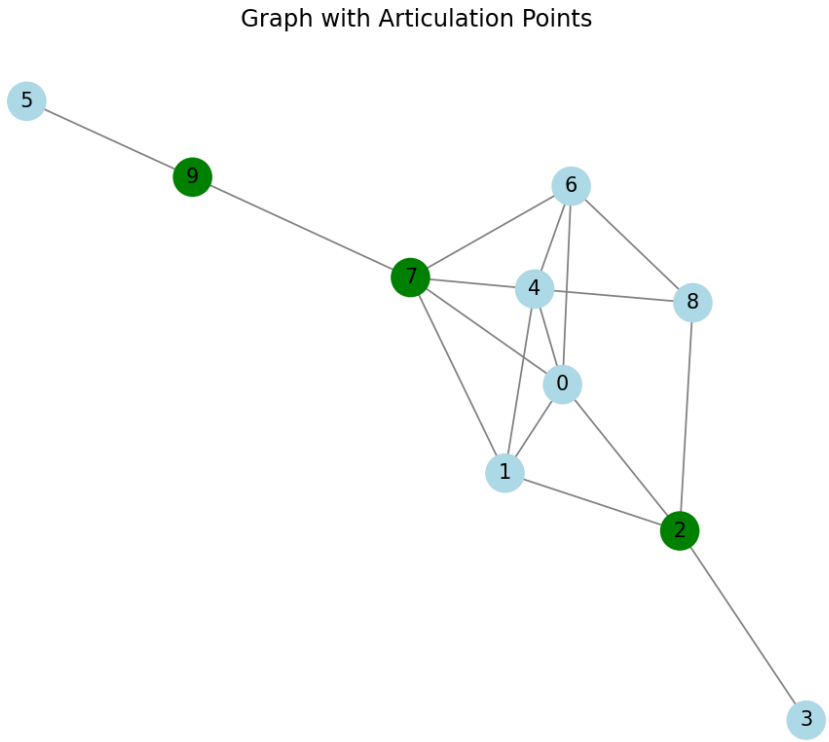
Graph Input:

```
10 17
4 6
9 5
8 6
7 1
4 8
8 2
0 1
0 7
1 2
0 4
7 9
6 7
4 7
2 0
1 4
2 3
6 0
```

Connected Components before removal of Articulation Points

```
Connected Components: [{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}]
```

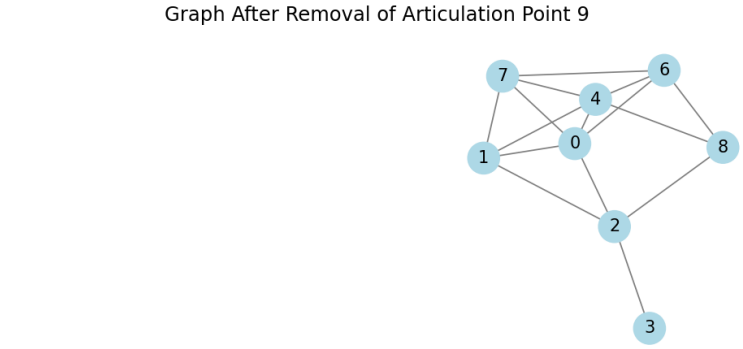
Visualization:

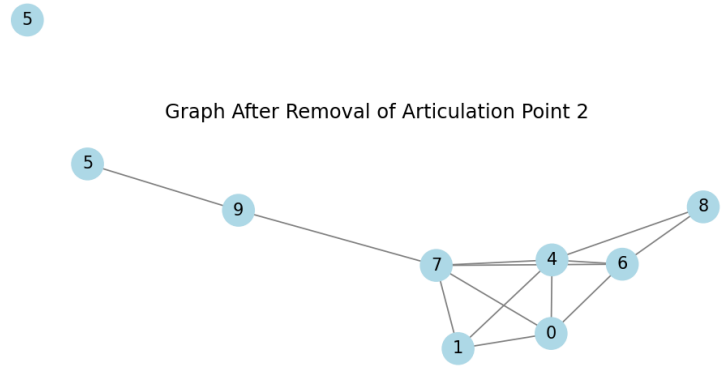


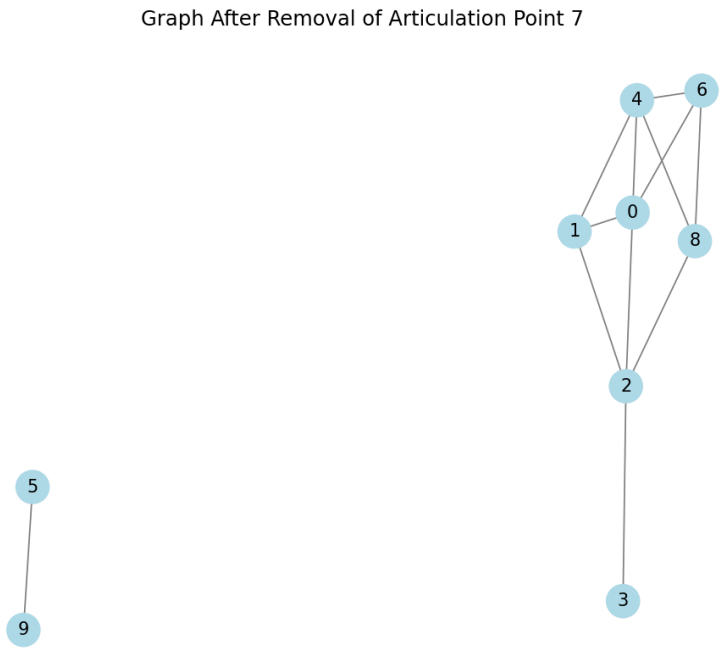
Connected Components after removal of each Articulation Point

```
Critical Nodes (Articulation Points): [2, 7, 9]
Removing node 9 will disconnect the network into 2 component
    (s): [{0, 1, 2, 3, 4, 6, 7, 8}, {5}]
Removing node 2 will disconnect the network into 2 component
    (s): [{0, 1, 4, 5, 6, 7, 8, 9}, {3}]
Removing node 7 will disconnect the network into 2 component
    (s): [{0, 1, 2, 3, 4, 6, 8}, {9, 5}]
```

Visualization







3.2 Case 2: Graph with No Articulation Points

In this situation, the graph has no articulation points, hence it remains connected since there is no removal of any node. For example, take the graph below,

Graph Input:

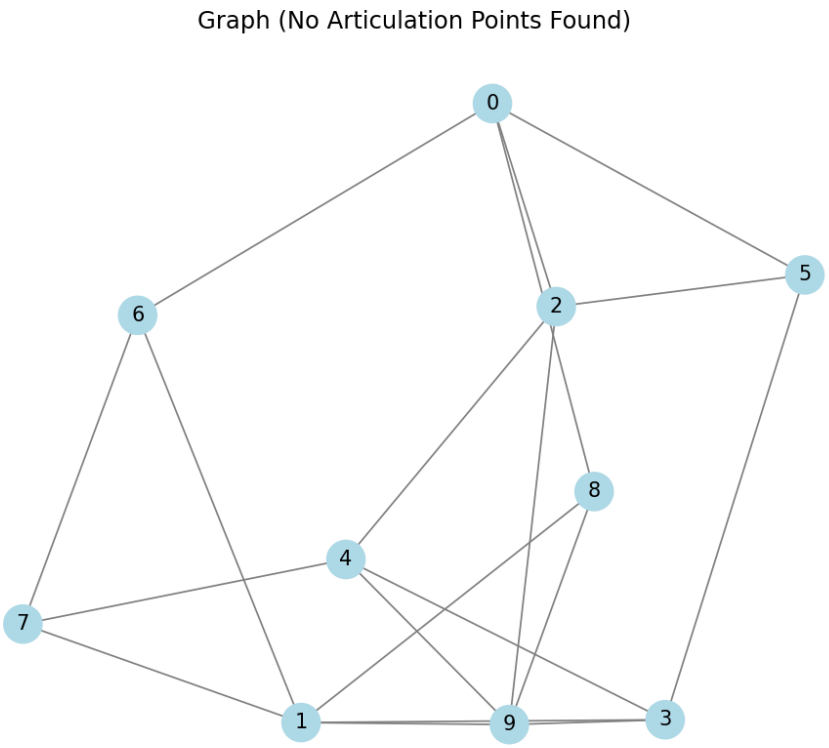
```
10 19
4 3
9 2
8 0
0 2
9 8
0 5
1 3
7 1
3 9
5 3
9 1
9 4
2 4
6 1
7 6
4 7
```

```
5 2
8 1
6 0
```

Connected Components before removal of Articulation Points

```
Connected Components: [{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}]
```

Visualization



Output

```
Network remains connected since no articulation points are
    found.
```

4 Challenges and Resolutions

4.1 Incorrect Connected Components

- Challenge: Initially, the algorithm computed connected components inaccurately after deleting articulation points.
- Resolution: The graph was properly copied using $G\_failed = graph.copy()$, and nodes were removed one by one.

4.2 Edge Cases

- Challenge: Handling graphs with no articulation points.
- Resolution: A check was added to verify if the articulation_points set was empty.

5 Summary

Network File	Articulation Points	Connected Components After Removal
network_1.txt	[2, 7, 9]	[{0, 1, 2, 3, 4, 6, 7, 8}, {5}]
network_2.txt	None	Network remains connected
network_3.txt	[5]	[{4}, {0, 1, 2, 3, 6, 7, 8, 9}]
network_4.txt	None	Network remains connected
network_5.txt	[1]	[{0, 3, 4, 5, 6, 7, 8, 9}, {2}]
network_6.txt	None	Network remains connected
network_7.txt	[9]	[{0, 1, 2, 4, 5, 6, 7, 8}, {3}]
network_8.txt	[9]	[{0, 2, 3, 4, 5, 6, 7, 8}, {1}]

```
import networkx as nx
import matplotlib.pyplot as plt

# Function to perform DFS and find articulation points
def dfs(u, graph, parent, disc, low, visited, articulation_points, time):
    visited[u] = True
    disc[u] = low[u] = time
    time += 1
    children = 0

    for v in graph[u]:
        if not visited[v]:
            parent[v] = u
            children += 1
            dfs(v, graph, parent, disc, low, visited, articulation_points, time)

            low[u] = min(low[u], low[v])

            # Condition 1: A root with two or more children
            if parent[u] == -1 and children > 1:
                articulation_points.add(u)

            # Condition 2: A non-root node where low[child] ≥ disc[node].
            if parent[u] != -1 and low[v] >= disc[u]:
                articulation_points.add(u)

        elif v != parent[u]:  # here v is an ancestor of u and there is a back edge
from some descendant of u to v
            low[u] = min(low[u], disc[v])

# Function to find articulation points in the graph
def find_articulation_points(graph):
    time = 0
    parent = {node: -1 for node in graph.nodes()}
    disc = {node: -1 for node in graph.nodes()}  # Discovery time
    low = {node: -1 for node in graph.nodes()}  # Lowest reacahble ancestor
    visited = {node: False for node in graph.nodes()}
    articulation_points = set()

    # Running DFS from all unvisited nodes
    for u in graph.nodes():
        if not visited[u]:
            dfs(u, graph, parent, disc, low, visited, articulation_points, time)

    return articulation_points

# Function to remove a node and find the resulting connected components
def remove_node_and_simulate(graph, node_to_remove):

    G_failed = graph.copy()

    # Remove the articulation point
    G_failed.remove_node(node_to_remove)

   # Find the connected components in the graph after removing articulation point
    components = list(nx.connected_components(G_failed))
    return components, G_failed

# Function to visualize the graph
```

```
def visualize_graph(graph, title, articulation_points=None):
    pos = nx.spring_layout(graph, seed=42)  # Layout for consistent visualization
    plt.figure(figsize=(10, 8))

    # Draw nodes
    node_colors = []
    for node in graph.nodes():
        if articulation_points and node in articulation_points:
            node_colors.append("green")
        else:
            node_colors.append("lightblue")

    nx.draw_networkx_nodes(graph, pos, node_color=node_colors, node_size=500)
    nx.draw_networkx_edges(graph, pos, edge_color="gray")
    nx.draw_networkx_labels(graph, pos, font_size=12, font_color="black")

    plt.title(title, fontsize=14)
    plt.axis("off")
    plt.show()

# Function to simulate network failure by removing articulation points using
NetworkX
def analyze_failure(graph, articulation_points):

    visualize_graph(graph, "Graph Before Removal of Articulation Points",
articulation_points)

    for node in articulation_points:
        components, G_failed = remove_node_and_simulate(graph, node)
        print(f"Removing node {node} will disconnect the network into
{len(components)} component(s): {components}")
        visualize_graph(G_failed, f"Graph After Removal of Articulation Point
{node}")

# Main function to read input files and execute the solution
def main():

    print("====NETWORK ANALYSIS====")
    for i in range(1, 9):
        file_name = f'network_{i}.txt'
        print(f"\n{file_name}:")

        with open(file_name, 'r') as file:
            V, E = map(int, file.readline().split())
            graph = nx.Graph()

            # Reading edges from the input file
            for _ in range(E):
                u, v = map(int, file.readline().split())
                graph.add_edge(u, v)

        articulation_points = find_articulation_points(graph)
        # print(f"Critical Nodes (Articulation Points):
{sorted(articulation_points)}")

        if not articulation_points:
            print("Network remains connected since no articulation points are
found.")
        else:
```

```
            # Remove all articulation points and simulating the network failure
            print(f"Critical Nodes (Articulation Points):
{sorted(articulation_points)}")
            # Simulate the network failure after removing critical nodes
            analyze_failure(graph, articulation_points)

if __name__ == '__main__':
    main()
```