Algorithm Lab(CS2271)- Assignment 3

Made By:- Ramesh Chandra Soren(2022CSB086)
Deep Saha(2022CSB078)
Santosh Methre(2022CSB084)

1. Implement Kruskal's algorithm and Prim's algorithm for minimum spanning tree problem using conventional data structures for graph and test for correctness using small examples.

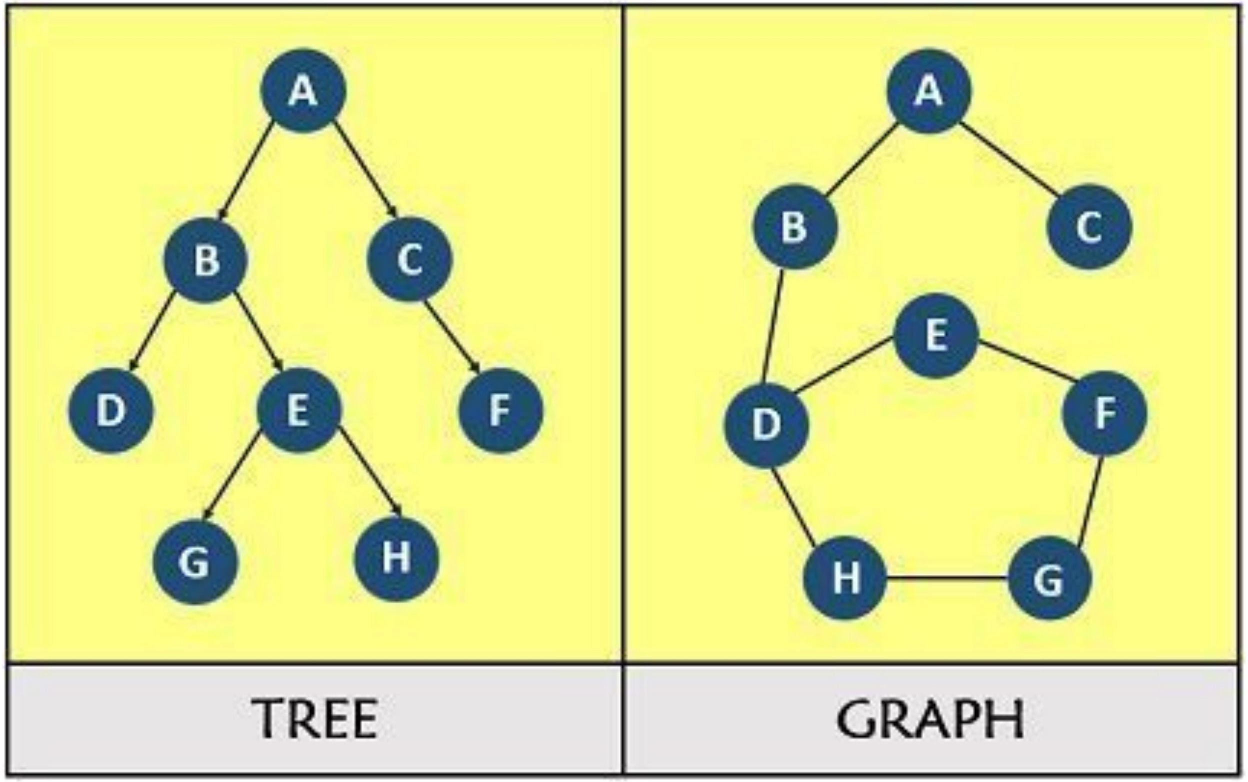
Difference between tree and graph

Trees are hierarchical data structures with a single root node, where each node can have multiple child nodes but only one parent.

They do not contain cycles or loops, ensuring a connected structure with exactly one path between any two nodes.

On the other hand, graphs are non-linear data structures that can have more than one path between vertices and can form cycles.

Graphs do not have a designated root node and can be directed or undirected, allowing for various relationships between nodes. While trees are commonly used for hierarchical data representation like file systems and organization charts, graphs find applications in scenarios such as social networks, maps, and network optimization


A
B
C
D
E
F
G
H
TREE
A
B
C
E
D
F
H
G
GRAPH

There are two main data structures commonly used for representing graphs:

1. Adjacency List
    - An adjacency list is a collection of lists, where each list represents the set of vertices adjacent to a particular vertex.
    - In an undirected graph, for every edge (u, v), vertex u's list contains v, and vertex v's list contains u.
    - In a directed graph, for every edge (u, v), vertex u's list contains v, but vertex v's list may or may not contain u, depending on the direction of the edge.
    - Advantages:
        - Space-efficient for sparse graphs (graphs with relatively few edges compared to the maximum possible number of edges).
        - Adding or removing an edge takes constant time on average.
        - Iterating over the neighbors of a vertex is efficient.
    - Disadvantages:
        - Finding if an edge exists between two vertices can take linear time in the number of edges, as you need to search through the adjacency lists.
        - Requires additional space to store the lists, which can be significant for dense graphs.

1. <u>Adjacency Matrix</u>
    - An adjacency matrix is a 2D array (or matrix) of size V x V, where V is the number of vertices in the graph.
    - For an undirected graph, if there is an edge between vertices u and v, both matrix[u][v] and matrix[v][u] are set to 1 (or the edge weight if it's a weighted graph).
    - For a directed graph, if there is an edge from u to v, matrix[u][v] is set to 1 (or the edge weight), but matrix[v][u] may or may not be set, depending on the direction of the edge.
    - Advantages:
        - Checking if an edge exists between two vertices takes constant time.
        - Iterating over the neighbors of a vertex is efficient for dense graphs.
    - Disadvantages:
        - Space-inefficient for sparse graphs, as it requires O(V^2) space, even if there are few edges.
        - Adding or removing an edge takes constant time, but it may require shifting elements in the matrix, which can be inefficient for large matrices.
        - Requires additional space to store edge weights if it's a weighted graph.

Kruskal's Algorithm:

1. Theory :
    - Kruskal's algorithm processes edges in ascending order of weight, adding edges that do not form cycles until V-1 edges are added, forming the minimum spanning tree.
    - It uses a priority queue to sort edges by weight and a union-find data structure to detect cycles.
2. Implementation Steps :
    - Sort the edges by weight.
    - Initialize an empty minimum spanning tree.
    - Iterate through the sorted edges, adding each edge that does not create a cycle.
    - Use a union-find data structure to check for cycles efficiently.

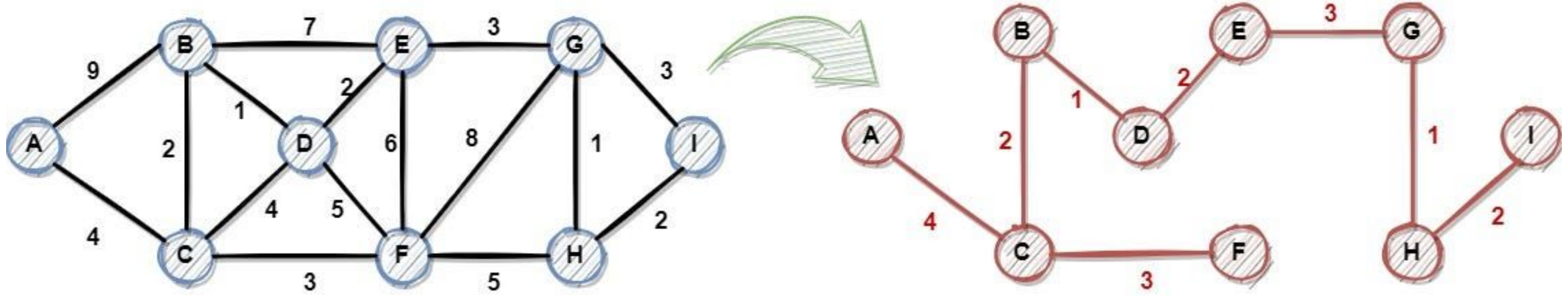
Pseudocode:

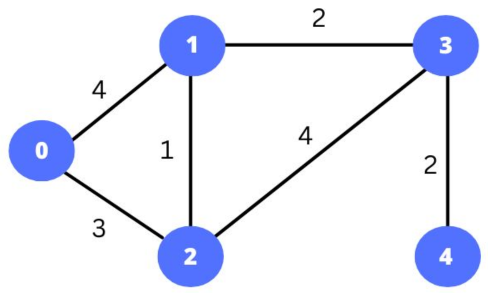
```
KRUSKAL(G):
    $A = \emptyset$
    foreach $v \in G.V$:
    MAKE-SET(v)
    sort the edges of G.E by increasing order of weight w
    foreach $(u, v) \in G.E$, taken in increasing order by weight:
    if FIND-SET(u) $\neq$ FIND-SET(v):
    $A = A \cup \{(u, v)\}$
    UNION(u, v)
    return A
```

Kruskal's Algorithm:

- Time Complexity: O(E log E) or O(E log V)
- Efficient for sparse graphs
- Practical for complex network clustering and transportation planning
- Builds MST by sorting edges by weight and adding them incrementally without forming cycles

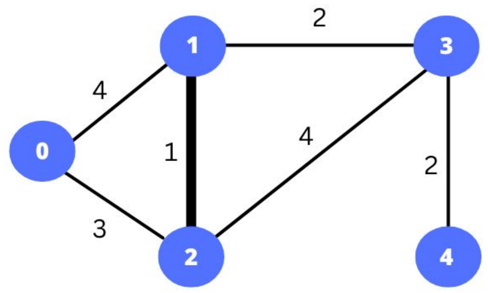
Kruskal's Algorithm





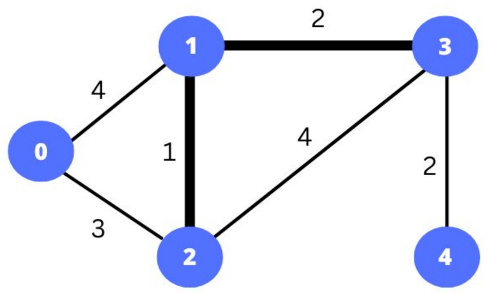
1
2
3
4
4
0
1
4
2
3
2
4

(a)



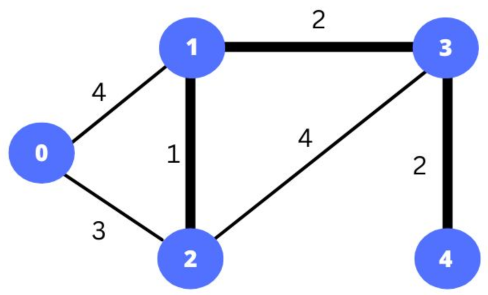
1
2
3
4
0
1
4
2
3
2
4

(b)



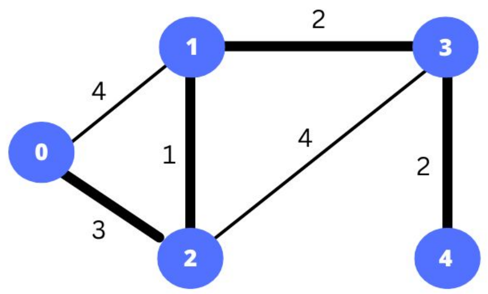
1
2
3
4
0
1
4
2
3
2
4

(c)



1
2
3
4
0
1
4
2
3
2
4

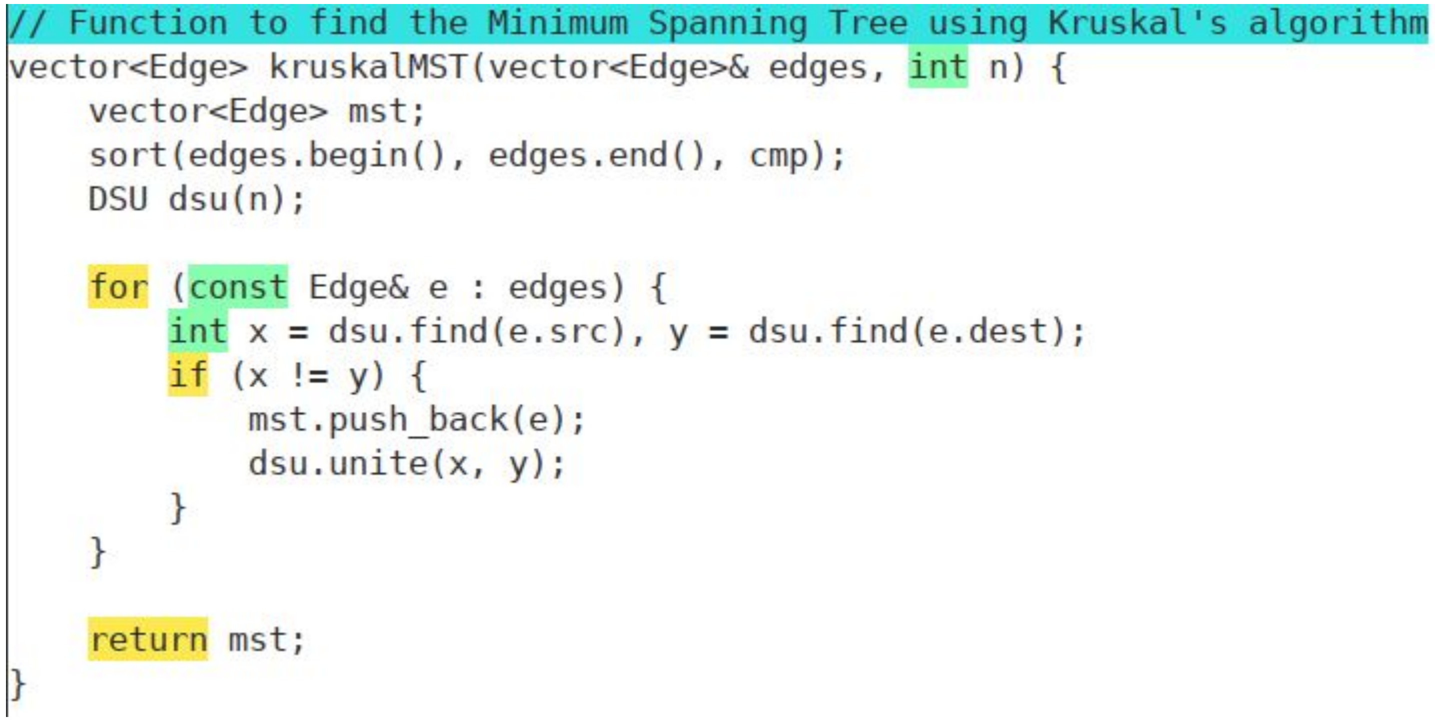
(d)



1
2
3
4
0
1
4
2
3
2
4

(e)

Implementation in C++



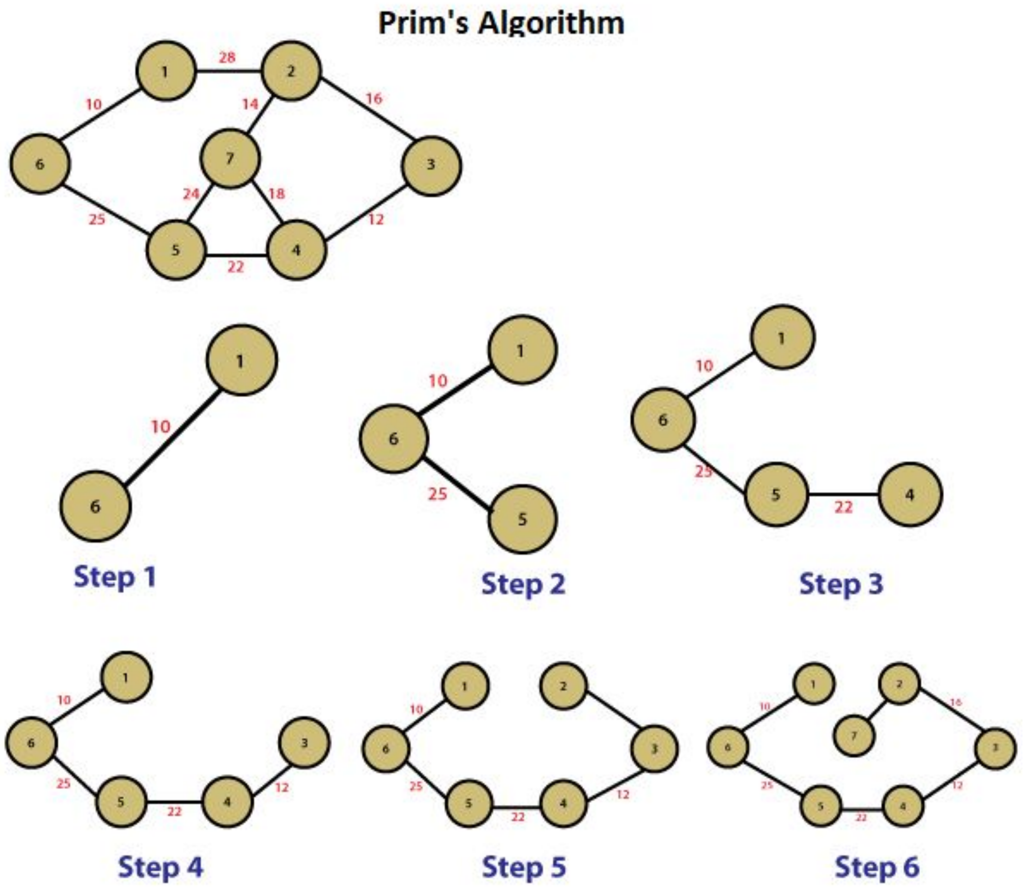
```cpp
// Function to find the Minimum Spanning Tree using Kruskal's algorithm
vector<Edge> kruskalMST(vector<Edge>& edges, int n) {
    vector<Edge> mst;
    sort(edges.begin(), edges.end(), cmp);
    DSU dsu(n);

    for (const Edge& e : edges) {
        int x = dsu.find(e.src), y = dsu.find(e.dest);
        if (x != y) {
            mst.push_back(e);
            dsu.unite(x, y);
        }
    }

    return mst;
}
```

Prim's Algorithm:

1. **Theory:**
    - Prim's algorithm starts with a single node and grows the tree by adding the minimum weight edge at each step until all vertices are included.
    - It maintains two sets of vertices: one in the MST and the other not yet included, selecting the minimum weight edge connecting the two sets.
2. **Implementation Steps:**
    - Initialize the MST with a starting vertex.
    - Maintain sets of included and fringe vertices.
    - Find the minimum weight edge connecting the sets and add it to the MST.
    - Repeat until all vertices are included.

Prim's Algorithm:

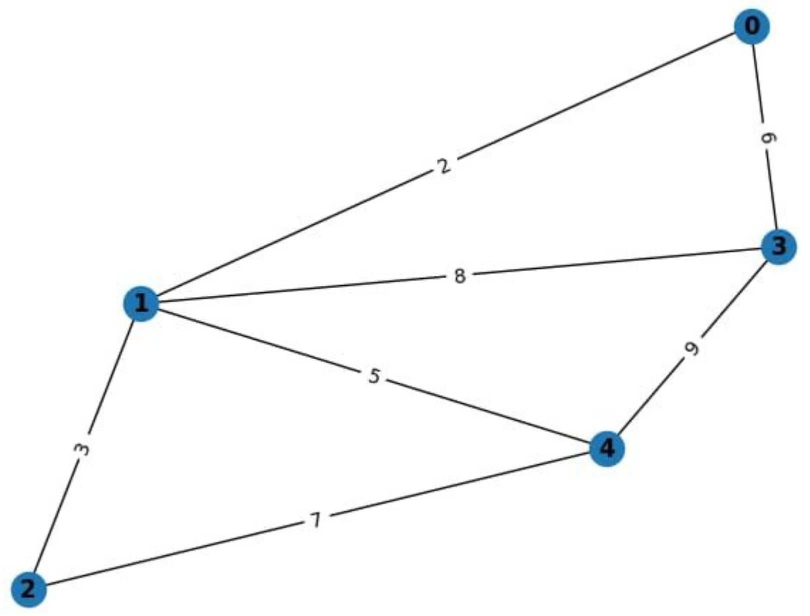
- Time Complexity: O(V^2) or O(E log V) with Fibonacci heaps
- Efficient for dense graphs
- Begins MST construction from the vertex with the minimum weight
- Suitable for solving the Traveling Salesman Problem and designing road and rail networks



Pseudocode:

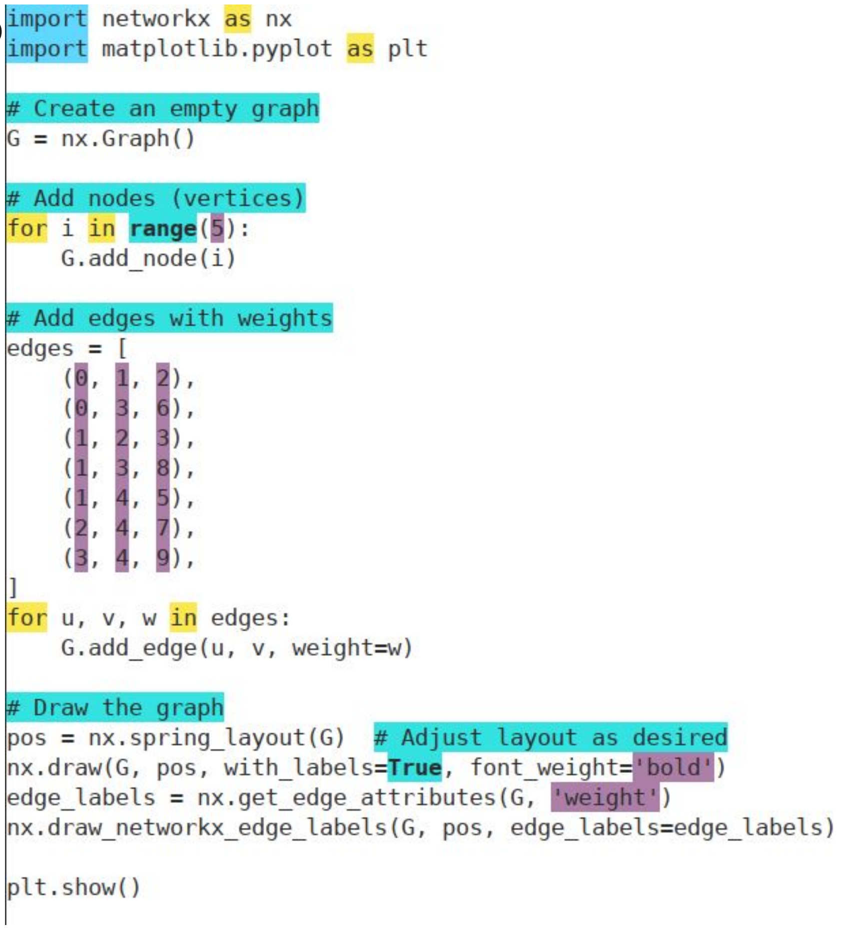
```
PRIM(G, w, r):
          foreach u ∈ G.V:
          u.key = ∞
          u.parent = NIL
          r.key = 0
          Q = G.V
          while Q ≠ ∅:
          u = EXTRACT-MIN(Q)
          foreach v ∈ G.Adj[u]:
          if v ∈ Q and w(u, v) < v.key:
          v.parent = u
          v.key = w(u, v)
```

The graph i have is given below:



How i plotted the graph?

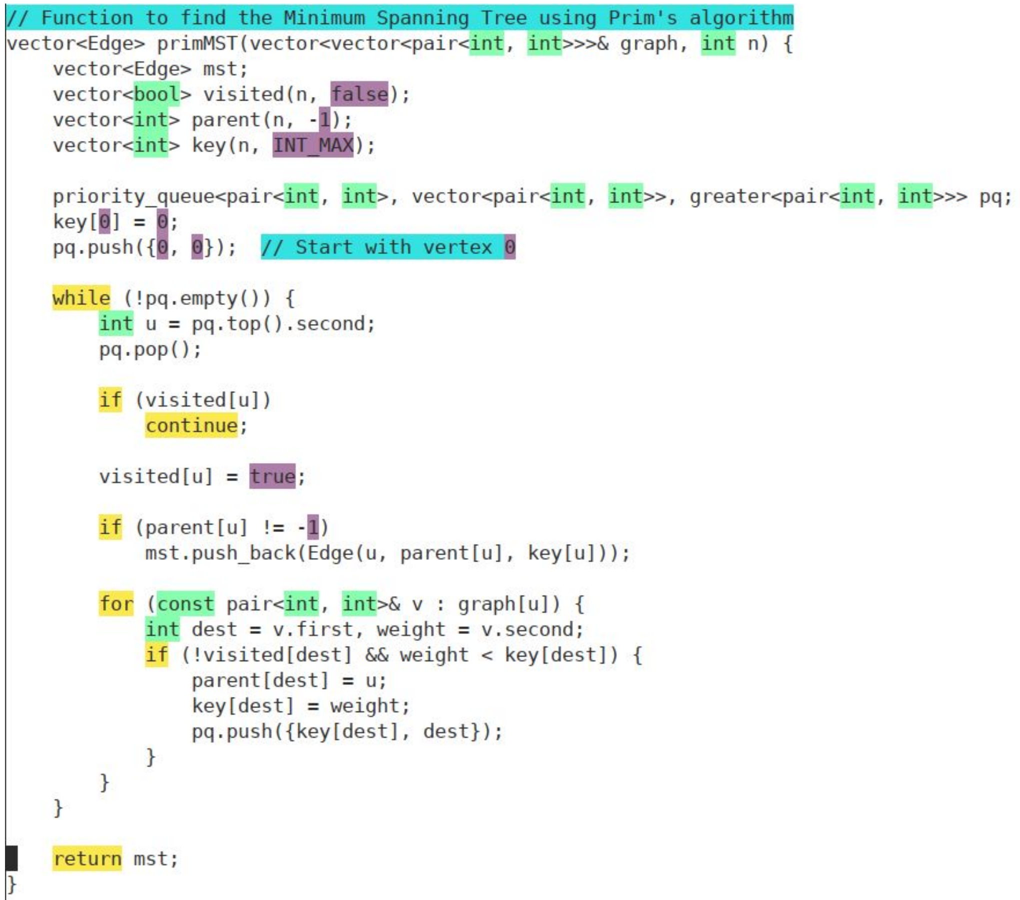
- The code utilizes the NetworkX library in Python for graph creation and manipulation.
- It adds nodes (vertices) and edges with weights to the graph.
- The graph is then visualized using Matplotlib, with node labels and edge weights displayed.



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

# Create an empty graph
G = nx.Graph()

# Add nodes (vertices)
for i in range(5):
    G.add_node(i)

# Add edges with weights
edges = [
    (0, 1, 2),
    (0, 3, 6),
    (1, 2, 3),
    (1, 3, 8),
    (1, 4, 5),
    (2, 4, 7),
    (3, 4, 9),
]
for u, v, w in edges:
    G.add_edge(u, v, weight=w)

# Draw the graph
pos = nx.spring_layout(G)  # Adjust layout as desired
nx.draw(G, pos, with_labels=True, font_weight='bold')
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

plt.show()
```

Implementation in C++

primMST function now implements Prim's algorithm using an adjacency list representation of the graph and a priority queue to efficiently find the vertex with the minimum key value.

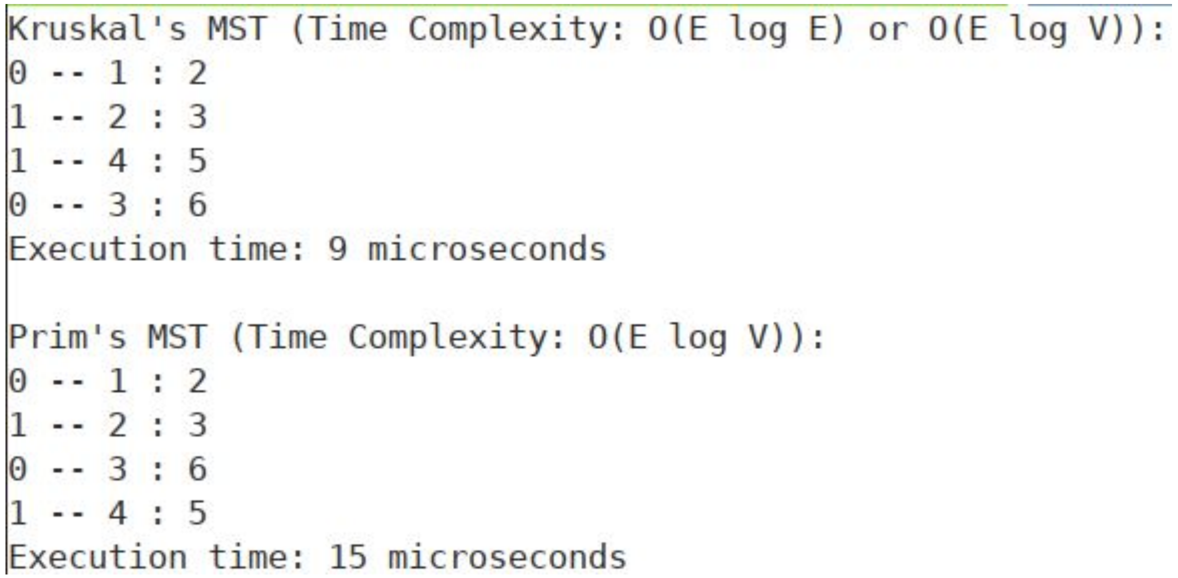


```cpp
// Function to find the Minimum Spanning Tree using Prim's algorithm
vector<Edge> primMST(vector<vector<pair<int, int>>>& graph, int n) {
    vector<Edge> mst;
    vector<bool> visited(n, false);
    vector<int> parent(n, -1);
    vector<int> key(n, INT_MAX);

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    key[0] = 0;
    pq.push({0, 0});  // Start with vertex 0

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        if (visited[u])
            continue;

        visited[u] = true;

        if (parent[u] != -1)
            mst.push_back(Edge(u, parent[u], key[u]));

        for (const pair<int, int>& v : graph[u]) {
            int dest = v.first, weight = v.second;
            if (!visited[dest] && weight < key[dest]) {
                parent[dest] = u;
                key[dest] = weight;
                pq.push({key[dest], dest});
            }
        }
    }

    return mst;
}
```

Comparison:

- Kruskal's algorithm is faster in sparse graphs due to its sorting-based approach.
- Prim's algorithm performs better in dense graphs by starting from the minimum weight vertex.
- Kruskal's algorithm offers connected components, while Prim's provides connected graphs.
- Each algorithm has unique applications based on graph characteristics and problem requirements.

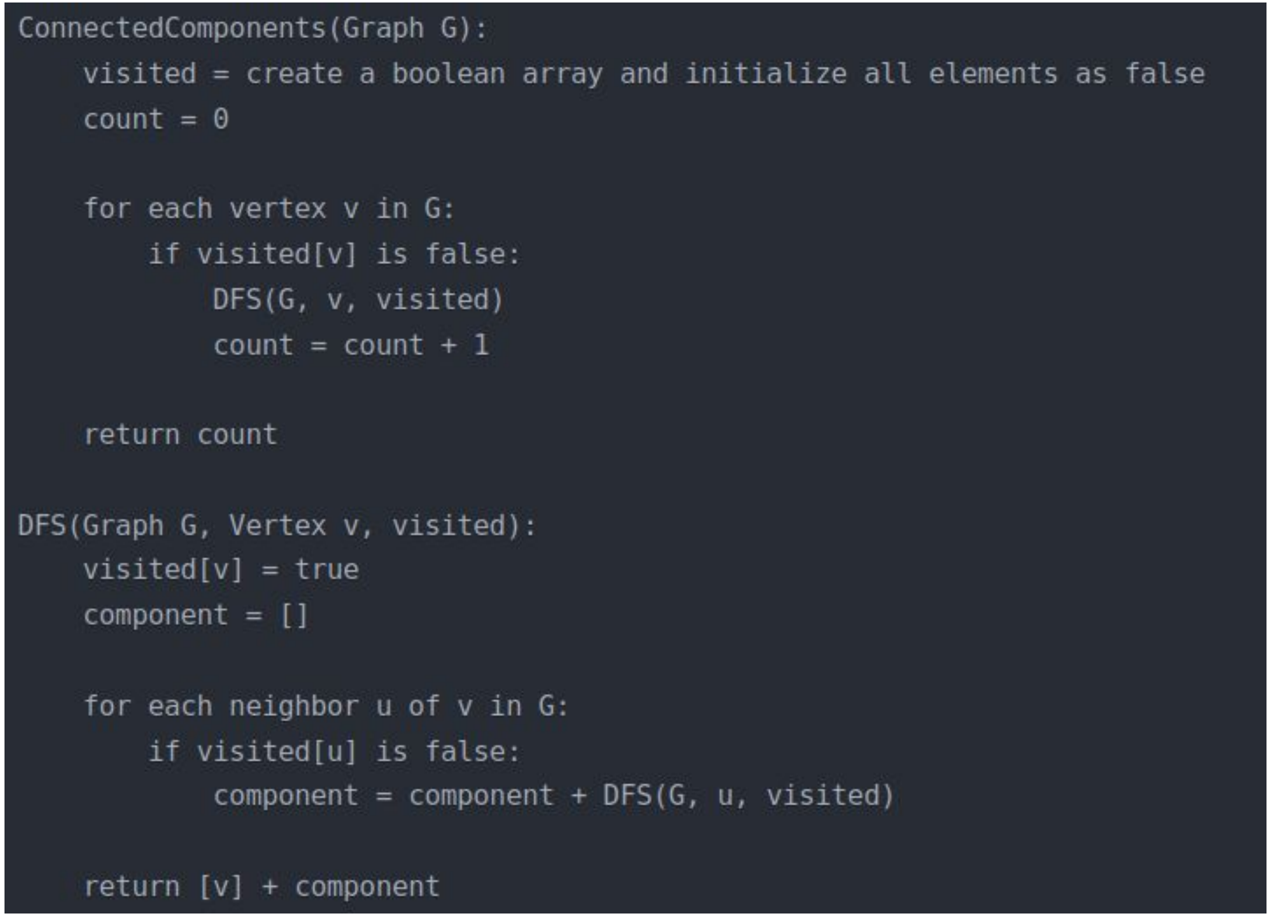
Output of my code:


```
Kruskal's MST (Time Complexity: O(E log E) or O(E log V)):
0 -- 1 : 2
1 -- 2 : 3
1 -- 4 : 5
0 -- 3 : 6
Execution time: 9 microseconds

Prim's MST (Time Complexity: O(E log V)):
0 -- 1 : 2
1 -- 2 : 3
0 -- 3 : 6
1 -- 4 : 5
Execution time: 15 microseconds
```

2. We need to scale up the things - from toy problems to real life problems. For this, you need to study some large datasets provided by reputed Universities, like

- For large datasets, you may need to use more efficient data structures and algorithms. For example, in Prim's algorithm, you can use a Fibonacci heap or a pairing heap instead of a binary heap (priority queue) to improve the time complexity from O(E log V) to O(E + V log V).
- You may also need to optimize memory usage by using hash maps or compressed sparse row/column formats to store the adjacency lists or matrices more efficiently.
- Additionally, you can consider parallelizing the algorithms to leverage multiple cores or processors for faster execution on large datasets.

pseudocode:



```
ConnectedComponents(Graph G):
    visited = create a boolean array and initialize all elements as false
    count = 0

    for each vertex v in G:
        if visited[v] is false:
            DFS(G, v, visited)
            count = count + 1

    return count

DFS(Graph G, Vertex v, visited):
    visited[v] = true
    component = []

    for each neighbor u of v in G:
        if visited[u] is false:
            component = component + DFS(G, u, visited)

    return [v] + component
```

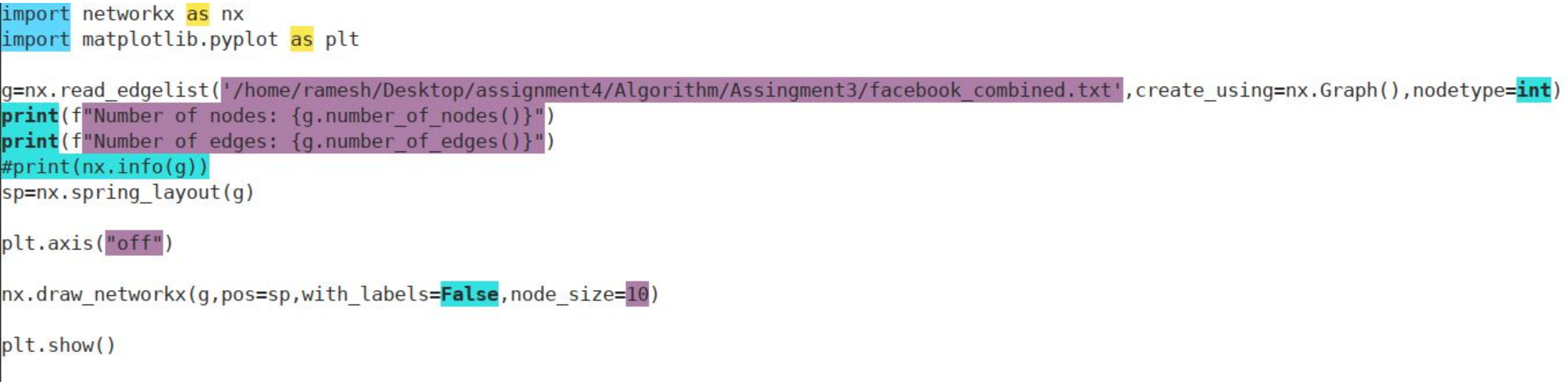
(a) SNAP - Stanford Network Analysis Project - datasets collected by Stanford University.

Title: Facebook Social Circles Dataset

1. Introduction
    - The Facebook Social Circles dataset consists of 'circles' or 'friends lists' obtained from Facebook.
    - This dataset was collected from survey participants using a Facebook app.
    - It includes node features (profiles), circles, and ego networks.
2. Dataset Information
    - **Anonymization:** Facebook data has been anonymized to protect user privacy. Original Facebook IDs have been replaced with new values.
    - **Feature Obscuration:** Feature vectors have been provided, but their interpretation has been obscured. For example, features like "political=Democratic Party" have been replaced with "political=anonymized feature 1".
    - **Interpretation Limitations:** While it's possible to determine similarities between users' features (such as political affiliations), the specific meanings of these features remain unknown.

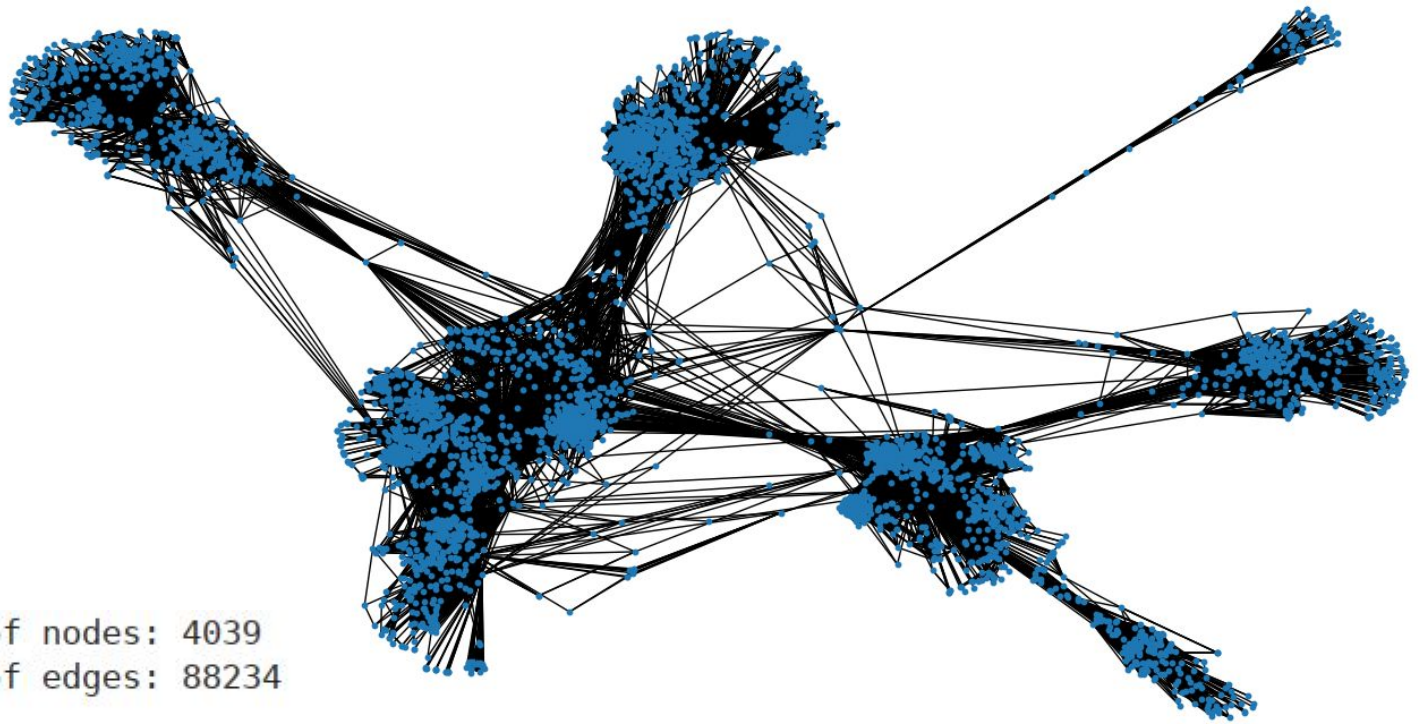
1. <u>Data Sources</u>
    - **Facebook:** The primary dataset under consideration, containing anonymized Facebook friend lists and associated features.
    - **Google+ and Twitter:** Similar datasets are also available from these platforms for comparative analysis.
2. <u>Purpose</u>
    - The dataset serves as a valuable resource for social network analysis, privacy-preserving data mining, and machine learning research.
    - It enables researchers to explore social network structures, user behaviors, and feature correlations while respecting user privacy.
3. <u>Applications</u>
    - Social Network Analysis
    - Community Detection
    - Feature Correlation Studies
    - Privacy-Preserving Data Mining
    - Machine Learning Model Training and Evaluation

How i plotted facebook dataset?



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

g=nx.read_edgelist('/home/ramesh/Desktop/assignment4/Algorithm/Assingment3/facebook_combined.txt',create_using=nx.Graph(),nodetype=int)
print(f"Number of nodes: {g.number_of_nodes()}")
print(f"Number of edges: {g.number_of_edges()}")
#print(nx.info(g))
sp=nx.spring_layout(g)

plt.axis("off")

nx.draw_networkx(g,pos=sp,with_labels=False,node_size=10)

plt.show()
```

Facebook dataset from SNAP



```
Number of nodes: 4039
Number of edges: 88234
```

FACEBOOK DATASET

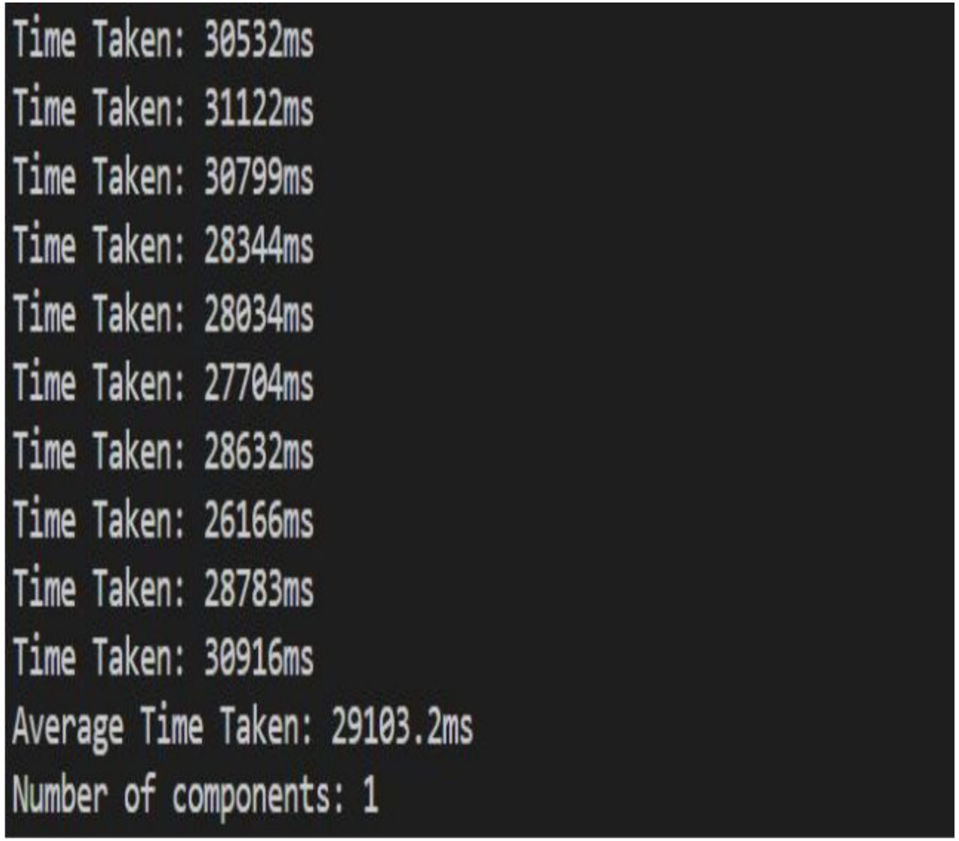
This dataset consists of 'circles' (or 'friends lists') from Facebook.

Facebook data was collected from survey participants using this Facebook app.

N= 4039

M= 88234

Implementation is without path compression.

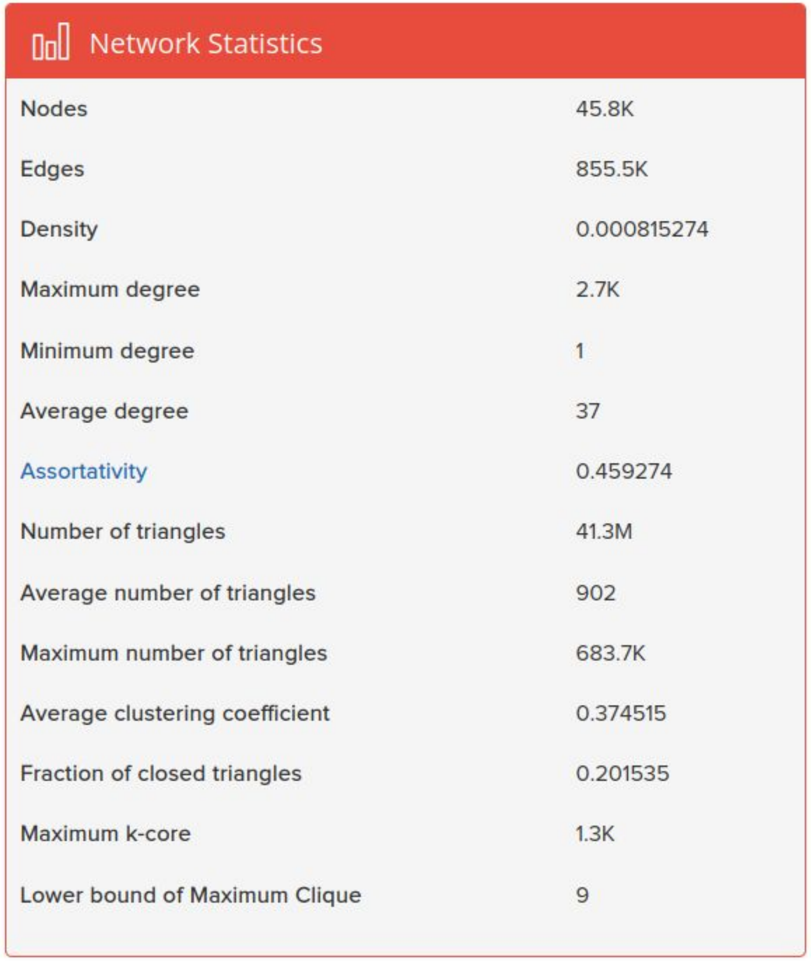


```
Time Taken: 30532ms
Time Taken: 31122ms
Time Taken: 30799ms
Time Taken: 28344ms
Time Taken: 28034ms
Time Taken: 27704ms
Time Taken: 28632ms
Time Taken: 26166ms
Time Taken: 28783ms
Time Taken: 30916ms
Average Time Taken: 29103.2ms
Number of components: 1
```

(b) KONECT - Koblenz Network Collection - datasets compiled by Koblenz University, Deutschland (Germany).

- The KONECT Project, initiated by Jérôme Kunegis, is centered in the field of network science.
- Objective: To collect network datasets, perform analyses, and make findings accessible online.
- Rooted at the University of Koblenz-Landau, Germany.
- KONECT stands for Koblenz Network Collection.
- All source code is Free Software, promoting accessibility and collaboration.
- Components include:
    a. Network analysis toolbox for GNU Octave.
    b. Network extraction library.
    c. Code for generating web pages, statistics, and plots.
- Currently holds 1,326 network datasets across 24 categories.
- Computed 57,019 graph statistics.
- Generated 93,461 plots.

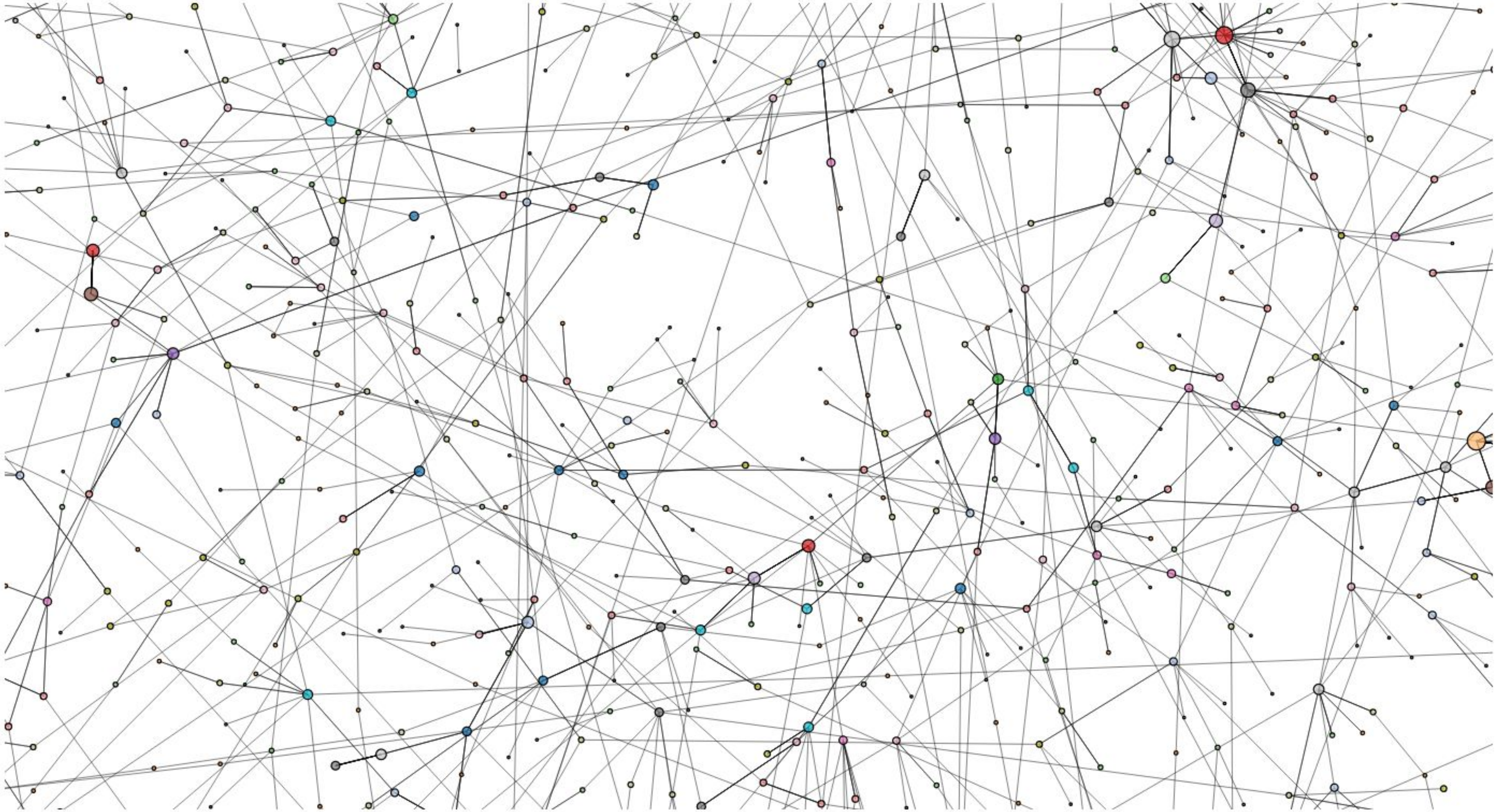
Facebook dataset from konect



Network Statistics

Nodes	45.8K
Edges	855.5K
Density	0.000815274
Maximum degree	2.7K
Minimum degree	1
Average degree	37
Assortativity	0.459274
Number of triangles	41.3M
Average number of triangles	902
Maximum number of triangles	683.7K
Average clustering coefficient	0.374515
Fraction of closed triangles	0.201535
Maximum k-core	1.3K
Lower bound of Maximum Clique	9

Plot of facebook data set from Konect



Les Misérables dataset from Konect

- The Les Misérables dataset is an undirected network representing co-occurrences of characters in Victor Hugo's novel.
- It consists of 77 nodes and 254 edges, where a node represents a character and an edge indicates co-appearance in the same chapter.
- Key observations include:
  - Minimum Cost for Spanning Tree: 105
  - Kruskal Algorithm Runtime: 0.016100 ms
  - Prim's Algorithm Runtime: 0.030710 ms
  - Connected Component Runtime: 0.007642 ms
- These metrics provide insights into the structure and connectivity of the character network in Les Misérables.

Thank You