# Graph Algorithms

Graph algorithms are fundamental tools in computer science, data science, and artificial intelligence, offering solutions to diverse problems ranging from traversal and clustering to classification and probabilistic inference. This module explores a range of graph algorithms, including Breadth-First Search (BFS), Depth-First Search (DFS), Minimum Spanning Trees (MST), Decision Trees, Bayesian Networks, Louvain Algorithm, Planetoid for Semi-Supervised Learning, and Spectral Clustering. Each algorithm is analyzed in terms of its mathematical foundations, implementation, applications, and limitations. Recent advances in graph-based feature engineering, scalability, and machine learning integration are also discussed, providing a comprehensive overview for practitioners and researchers.

*This document is an extension of the research and lecture notes completed at Johns Hopkins University, Whiting School of Engineering, Engineering for Professionals, Artificial Intelligence Master's Program, Computer Science Master's Program and Data Science Master's Program.*

# Contents

# 1 Introduction to Graph Algorithms

Graph algorithms form the backbone of computational solutions to problems involving networks, relationships, and structures. From social networks and transportation systems to biological networks and recommendation engines, graphs provide a versatile framework for representing and analyzing complex systems.

This module focuses on a curated selection of graph algorithms, each addressing specific challenges in graph theory and its applications. Traversal algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) offer foundational methods for exploring graph structures. Optimization techniques like Minimum Spanning Trees (MST) are crucial for network design and clustering. Advanced algorithms such as Bayesian Networks and Louvain are employed in probabilistic reasoning and community detection, respectively, while Spectral Clustering facilitates dimensionality reduction and clustering.

Machine learning on graphs is also explored, with algorithms like Planetoid providing robust frameworks for semi-supervised learning. Emphasis is placed on the mathematical formulations, implementation details, advantages, limitations, and practical applications of these algorithms. By examining both classical methods and recent innovations, this document aims to bridge theoretical knowledge and real-world applications in graph analytics.

## 2 Types of Graph Algorithms Problems

Graph algorithms address a wide range of problems that are critical in computer science, data science, and artificial intelligence. Below, we present specific types of graph algorithm problems with a focus on the following algorithms: Breadth-First Search, Depth-First Search, Minimum Spanning Trees, Decision Trees, Bayesian Networks, Louvain Algorithm for Graph-Based Feature Engineering, Planetoid for Semi-Supervised Learning on Graphs, and Spectral Clustering for Dimensionality Reduction and Clustering.

**Traversal Algorithms:**

*Breadth-First Search (BFS):* Systematically explores nodes level-by-level, identifying shortest paths in unweighted graphs and connected components.

*Depth-First Search (DFS):* Explores nodes as deeply as possible before backtracking, commonly used for cycle detection, topological sorting, and finding connected components.

**Minimum Spanning Tree Problems:**

*Minimum Spanning Trees (MST):* Finds the subset of edges that connect all vertices with the minimum possible total edge weight while avoiding cycles. Examples include Kruskal's and Prim's algorithms, applicable in network design and clustering.

**Decision-Making and Probabilistic Inference:**

*Decision Trees:* Solve graph-based classification and regression problems by splitting data at each node based on feature values. Decision trees are widely used in machine learning for interpretable models.

*Bayesian Networks:* Represent probabilistic dependencies among variables using a directed acyclic graph (DAG). Used for probabilistic inference, causal reasoning, and applications in fields such as medical diagnosis and risk analysis.

**Community Detection and Feature Engineering:**

*Louvain Algorithm for Graph-Based Feature Engineering:* Detects communities in graphs using modularity optimization, enabling applications in network analysis and graph-based feature extraction for machine learning.

**Semi-Supervised Learning:**

*Planetoid for Semi-Supervised Learning on Graphs:* Combines graph structure and node features to optimize classification performance on graphs, particularly in scenarios with labeled and unlabeled data.

**Dimensionality Reduction and Clustering:**

*Spectral Clustering Algorithm:* Leverages the eigenvalues and eigenvectors of the graph Laplacian to embed nodes into a lower-dimensional space, enabling dimensionality reduction and clustering of data points embedded in graph structures.

Each of these algorithms addresses specific types of problems, from traversal and clustering to classification and probabilistic inference, making them versatile tools in solving real-world challenges in data science, machine learning, and artificial intelligence.

# 3 Mathematical Foundations, Graph Representation and Basics

## 3.1 Graph Terminology

A graph $G$ is defined as a pair $(V, E)$, where $V$ is the set of *vertices* (or nodes) and $E$ is the set of *edges* (or links) connecting pairs of vertices [2]. The key types of graphs include:

- **Directed Graph (Digraph)**: Each edge has a direction, represented as an ordered pair $(u, v)$ indicating a one-way link from vertex $u$ to vertex $v$.
- **Undirected Graph**: Edges are bidirectional, represented as unordered pairs $\{u, v\}$.
- **Weighted Graph**: Each edge has an associated weight or cost, denoted as $w(u, v)$ [11].
- **Unweighted Graph**: Edges do not carry any weight, making traversal cost uniform across edges.

## 3.2 Graph Representation Methods

Graphs can be represented in several ways depending on their structure and density [3, 4]. The two most common representations are:

### Adjacency Matrix

An adjacency matrix is a $|V| \times |V|$ matrix $A$, where $A[i][j] = 1$ if there is an edge from vertex $i$ to vertex $j$, and $A[i][j] = 0$ otherwise. This representation is ideal for **dense graphs** where the number of edges $E$ is close to $|V|^2$.

- **Time Complexity**: Checking if an edge exists: $O(1)$.
- **Space Complexity**: $O(|V|^2)$.

### Adjacency List

An adjacency list represents each vertex as a list of its neighboring vertices. It is more efficient for **sparse graphs** where $E$ is much smaller than $|V|^2$.

- **Time Complexity**: Checking if an edge exists: $O(d)$, where $d$ is the degree of the vertex.
- **Space Complexity**: $O(|V| + |E|)$.

### Choosing a Representation

- Use an **Adjacency Matrix** when the graph is dense, or when constant-time edge lookup is required.
- Use an **Adjacency List** when the graph is sparse, to save memory and optimize traversal operations.

# 4 Breadth First Search Traversal Algorithms

The **Breadth-First Search (BFS)** algorithm is a fundamental graph traversal method used to explore nodes level-by-level. Starting from a designated source vertex $s$, BFS systematically visits all vertices reachable from $s$ in increasing order of their distance (measured by the number of edges). BFS is widely used in various fields, including network analysis, shortest path discovery, and as a component in regression models when dealing with graph-structured data [2].

## 4.1 Mathematical Formulations

A graph $G$ is defined as:

$$G = (V, E),$$

where:

- $V$ is the set of vertices ($|V| = n$).
- $E$ is the set of edges ($|E| = m$).

Given a source vertex $s \in V$, the BFS traversal discovers all reachable vertices and their distances $d(v)$ from $s$, where $d(v)$ is defined as:

$$d(v) = \min \{k \mid v \text{ is reachable from } s \text{ by a path of length } k\}.$$

The traversal order $\mathbf{X}$ can be expressed as:

$$\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T,$$

where $\mathbf{x}_i$ represents the $i$-th vertex visited.

## 4.2 Breadth-First Search Traversal Algorithm

The BFS algorithm explores vertices in layers, ensuring that all vertices at distance $k$ from the source are visited before any vertices at distance $k + 1$. It uses a **queue** to manage the vertices yet to be explored, and a **visited** array to avoid revisiting nodes.

## 4.3 Implementation Details

To effectively use BFS for graph traversal, consider the following steps:

1. **Data Preprocessing**:
   - Represent the graph using an **adjacency list** or **adjacency matrix**.
   - Ensure data is cleaned, and vertex labels are appropriately mapped.
2. **Model Processing**:
   - Initialize the queue and visited array.

**Algorithm 1** Breadth-First Search (BFS)

**Require:** $G = (V, E)$ is a graph, $s \in V$ is the source vertex
**Ensure:** BFS traversal order of vertices

```
function BFS(G, s)
    Initialize an empty queue Q
    Initialize a boolean array visited of size |V| and set all entries to false
    visited[s] ← true
    Enqueue s into Q
    while Q is not empty do
        v ← Dequeue from Q
        print v                                              ▷ Visit the current vertex
        for all neighbor u of v such that (v, u) ∈ E and visited[u] = false do
            visited[u] ← true
            Enqueue u into Q
        end for
    end while
end function
```

- Perform the BFS traversal as described in Algorithm 1.

3. **Evaluation**:
   - Track traversal order and distances from the source vertex.
   - For large datasets, measure execution time and memory usage.

### 4.4 Advantages and Limitations

**Advantages:**

- Guarantees finding the **shortest path** in unweighted graphs [3, 4].
- Simple and easy to implement.
- Useful for **connected component detection** and **cycle detection**.

**Limitations:**

- BFS can be **memory-intensive** for large graphs, as it stores many vertices in the queue.
- Inefficient for graphs with extremely large or infinite search spaces [11].
- Not suitable for graphs with weighted edges (use **Dijkstra's algorithm** for weighted graphs).

### 4.5 Analysis of the Breadth-First Search (BFS) Traversal Algorithm

Given a graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges, the Breadth-First Search (BFS) algorithm explores all reachable vertices from a source vertex $s$ in a level-by-level manner. This section provides a detailed analysis of the BFS algorithm, including its time complexity and a correctness proof.

**Time Complexity Analysis**

The BFS algorithm processes each vertex and edge in a systematic manner. The following analysis explains the time complexity of BFS [3, 4, 11].

- Initialization: - Initializing the queue $Q$ and the boolean array **visited** takes $O(|V|)$ time.
- Traversal Loop: - Each vertex is enqueued and dequeued exactly once. Therefore, the operations related to vertices (enqueue, dequeue, and marking as visited) take $O(|V|)$ time.
- Edge Exploration: - For each vertex $v$, the algorithm explores all its neighbors. The total number of edge explorations across the entire graph is $O(|E|)$.

**Overall Time Complexity:**

Combining the vertex and edge processing, the total time complexity of the BFS algorithm is:

$$O(|V| + |E|).$$

This complexity reflects the fact that each vertex and edge is processed at most once. For a sparse graph where $|E| \approx |V|$, the time complexity is linear. For a dense graph where $|E| \approx |V|^2$, the complexity remains $O(|V| + |E|)$ [2].

**Summary of Time Complexity Analysis**

- Time Complexity: $O(|V| + |E|)$
- Space Complexity: $O(|V|)$ for storing the **visited** array and the queue.

The BFS algorithm is efficient for traversing large graphs, especially when the graph is sparse.

### 4.6 Correctness Proof

The correctness of the BFS algorithm can be proven by induction on the distance from the source vertex $s$.

**Base Case:**

- The algorithm starts by marking the source vertex $s$ as visited and enqueuing it. This ensures that $s$ is visited first, which is correct.

**Inductive Hypothesis:**

- Assume that all vertices at distance $k$ from $s$ are visited correctly by the BFS algorithm.

**Inductive Step:**

- For each vertex $v$ at distance $k$ that is dequeued, the algorithm explores all its unvisited neighbors $u$ and enqueues them. Since these neighbors $u$ are exactly at distance $k + 1$, the algorithm visits all vertices at distance $k + 1$ correctly.

By induction, the BFS algorithm correctly visits all reachable vertices from $s$ in increasing order of their distance from $s$ [3, 4, 11].

**Summary of Correctness Proof**

- BFS correctly visits vertices in increasing order of their distance from the source vertex $s$.
- The use of a queue ensures that vertices are processed level-by-level.
- The visited array guarantees that each vertex is visited at most once, preventing cycles from causing infinite loops.

### 4.7 Summary of Breadth-First Search Traversal

The **Breadth-First Search (BFS)** algorithm is a fundamental graph traversal method that explores vertices in a level-by-level manner, starting from a designated source vertex $s$. BFS ensures that all vertices at distance $k$ from $s$ are visited before any vertices at distance $k+1$. It uses a **queue** to maintain the exploration order and a **visited** array to track which vertices have been visited, preventing revisits. BFS guarantees finding the **shortest path** in unweighted graphs due to its systematic traversal of neighbors. The algorithm has a time complexity of $O(|V|+|E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges [3, 4]. BFS is widely used in various applications, such as **network analysis**, **pathfinding**, **web crawling**, and **graph-based data analysis**. Its correctness is ensured by the queue-based level-order traversal, which processes vertices in the order they are discovered [2, 11].

# 5 Depth First Search Traversal Algorithms

The **Depth-First Search (DFS)** algorithm is a fundamental graph traversal technique that explores vertices by visiting as deeply as possible along each branch before backtracking. DFS starts from a designated source vertex $s$ and uses a recursive or stack-based approach to explore all reachable vertices. In the context of data science and regression analysis, DFS is particularly useful for tasks such as analyzing hierarchical structures, detecting cycles in dependency graphs, and traversing decision trees [3, 4, 2]. DFS is also essential in applications like topological sorting and finding connected components.

## 5.1 Mathematical Formulations

A graph $G$ is defined as:

$$G = (V, E),$$

where:

- $V$ is the set of vertices ($|V| = n$).
- $E$ is the set of edges ($|E| = m$).

The traversal order $\mathbf{X}$ produced by DFS can be expressed as:

$$\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T,$$

where $\mathbf{x}_i$ represents the $i$-th vertex visited during the traversal. For a vertex $v$, DFS recursively explores all neighbors $u$ such that $(v, u) \in E$ and $u$ has not been visited yet.

## 5.2 Depth-First Search Traversal Algorithm

The DFS algorithm explores vertices by visiting a neighbor, then recursively visiting the neighbor's neighbors, backtracking when no unvisited neighbors remain. The algorithm uses a **visited** array to track whether a vertex has been explored.

## 5.3 Implementation Details

To effectively use DFS for graph traversal, the following steps are crucial:

1. **Data Preprocessing**:
   - Represent the graph $G$ using an **adjacency list** for efficient neighbor lookups.
   - Ensure the graph data is cleaned, and vertex identifiers are properly indexed.
2. **Model Processing**:
   - Initialize a boolean array **visited** to track which vertices have been explored.
   - Execute the DFS traversal by calling the recursive function 'DFS_Visit' on the source vertex $s$.

**Algorithm 2** Depth-First Search (DFS)

**Require:** $G = (V, E)$ is a graph, $s \in V$ is the source vertex
**Ensure:** DFS traversal order of vertices

```
function DFS(G, s)
    Initialize a boolean array visited of size |V| and set all entries to false
    Initialize an empty list X to store the traversal order
    DFS_Visit(s, visited, X)
end function
function DFS_Visit(v, visited, X)
    visited[v] ← true
    Append v to X
    print v                                              ▷ Visit the current vertex
    for all neighbor u of v such that (v, u) ∈ E and visited[u] = false do
        DFS_Visit(u, visited, X)
    end for
end function
```

3. **Evaluation**:
   - Track the traversal order **X** to verify the sequence of visited vertices.
   - For large graphs, measure the execution time and memory usage.

### 5.4 Advantages and Limitations

**Advantages:**

- **Simple and Efficient**: DFS is straightforward to implement and has a time complexity of $O(|V| + |E|)$.
- **Memory-Efficient for Sparse Graphs**: Uses stack space proportional to the depth of the recursion.
- **Applications**: Suitable for tasks like detecting cycles, topological sorting, and finding connected components [11].

**Limitations:**

- **Not Suitable for Shortest Paths**: DFS does not guarantee the shortest path in unweighted graphs (use BFS instead).
- **Stack Overflow**: For very deep or infinite graphs, recursion can lead to stack overflow.
- **Order Sensitivity**: The traversal order depends on the order in which neighbors are visited.

### 5.5 Analysis of the Depth-First Search (DFS) Traversal Algorithm

Given a graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges, the Depth-First Search (DFS) algorithm explores all reachable vertices from a source vertex $s$ by visiting as deeply as possible before backtracking. This section provides a detailed analysis of the DFS algorithm, including its time complexity, correctness proof, and summaries.

### Time Complexity Analysis

The DFS algorithm processes each vertex and edge in a systematic manner. The time complexity of DFS can be broken down as follows [3, 4, 11]:

- Initialization: - Initializing the boolean array **visited** of size $|V|$ takes $O(|V|)$ time.
- Traversal Loop: - Each vertex $v$ is visited exactly once, and the function `DFS_Visit` is called recursively for each unvisited neighbor. Therefore, the total time spent on vertex operations is $O(|V|)$.
- Edge Exploration: - For each vertex $v$, DFS explores all its neighbors. The total number of edge explorations across the entire graph is $O(|E|)$.

### Overall Time Complexity:

Combining the vertex and edge processing, the total time complexity of the DFS algorithm is:

$$O(|V| + |E|).$$

This complexity reflects that each vertex and edge is processed at most once. The DFS algorithm is efficient for both sparse and dense graphs, as the time complexity scales with the number of vertices and edges.

### Summary of Time Complexity Analysis

- Time Complexity: $O(|V| + |E|)$
- Space Complexity: $O(|V|)$ for storing the **visited** array and the recursion stack.

The DFS algorithm is suitable for traversing large graphs, especially when the graph is sparse, and provides an efficient way to explore all reachable nodes from a source vertex.

## 5.6 Correctness Proof

The correctness of the DFS algorithm can be established using induction and the structure of recursive calls [2, 3, 4].

### Base Case:

The algorithm starts by visiting the source vertex $s$, marking it as visited, and adding it to the traversal list $\mathbf{X}$. This step ensures that $s$ is correctly visited.

### Inductive Hypothesis:

Assume that for a vertex $v$, the DFS algorithm correctly visits $v$ and all its reachable neighbors.

**Inductive Step:**

For each unvisited neighbor $u$ of $v$, the function `DFS_Visit` is called recursively. By the inductive hypothesis, $u$ and all vertices reachable from $u$ are visited correctly. This recursive process continues until all reachable vertices are visited.

**Conclusion:**

By induction, the DFS algorithm visits all vertices reachable from the source $s$ exactly once, ensuring a complete traversal of the graph.

**Summary of Correctness Proof**

- The DFS algorithm correctly visits all vertices reachable from the source vertex $s$.
- The use of a recursive approach ensures that the traversal explores as deeply as possible before backtracking.
- The **visited** array guarantees that each vertex is visited at most once, preventing infinite loops.

### 5.7 Summary of Depth-First Search Traversal Algorithm

The **Depth-First Search (DFS)** algorithm is a fundamental graph traversal technique that explores vertices by visiting as deeply as possible along each branch before backtracking. Starting from a source vertex $s$, DFS uses a recursive or stack-based approach to visit all reachable vertices. It employs a **visited** array to ensure each vertex is explored only once, preventing infinite loops. DFS has a time complexity of $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges [3, 4]. The algorithm is particularly useful for tasks such as detecting cycles, analyzing hierarchical structures, topological sorting, and finding connected components [2, 11]. DFS is efficient for sparse graphs and is straightforward to implement, though it does not guarantee the shortest path in unweighted graphs and may encounter stack overflow for very deep recursions.

# 6 Minimum Spanning Trees

**Kruskal's Algorithm** is a greedy algorithm used to find the *Minimum Spanning Tree (MST)* of a connected, weighted graph $G = (V, E)$. The MST is a subset of the edges $E$ that connects all the vertices $V$ while minimizing the total edge weight and avoiding cycles [3, 4]. In data science and regression analysis, MSTs can be used for clustering, network design, and identifying underlying structures in data. For example, MSTs help in constructing hierarchical clusters or reducing complex networks to their most essential connections [11].

## 6.1 Mathematical Formulations

A graph $G$ is defined as:

$$G = (V, E),$$

where:

- $V$ is the set of vertices, $|V| = n$.
- $E$ is the set of weighted edges, $|E| = m$.

The goal of Kruskal's algorithm is to find a subset $T \subseteq E$ such that: 1. $T$ forms a spanning tree of $G$. 2. The total weight of $T$ is minimized.

The total weight $W(T)$ of the MST $T$ is given by:

$$W(T) = \sum_{(u,v)\in T} w(u, v),$$

where $w(u, v)$ is the weight of edge $(u, v)$ [2].

## 6.2 Minimum Spanning Tree Algorithm

Given a graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of weighted edges, the following algorithm finds the Minimum Spanning Tree (MST) using Kruskal's algorithm. The MST is a subset of edges that connects all the vertices with the minimum possible total edge weight and without any cycles.

Kruskal's algorithm works by sorting all the edges by weight and adding edges to the MST in order of increasing weight, ensuring no cycles are formed. The algorithm uses the **disjoint-set (union-find)** data structure to detect cycles efficiently. The steps are:

1. Sort all edges in non-decreasing order of their weights. 2. Initialize an empty list to store the edges of the MST. 3. For each edge $(u, v)$: - If $u$ and $v$ belong to different sets, add $(u, v)$ to the MST and merge the sets. 4. Stop when the MST contains $|V| - 1$ edges.

## 6.3 Implementation Details

1. **Data Preprocessing**:

**Algorithm 3** Kruskal's Algorithm for Minimum Spanning Tree

**Require:** $G = (V, E)$ is a connected, weighted graph with $|V|$ vertices and $|E|$ edges.
**Ensure:** $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|V|-1}]^T$ representing the edges of the Minimum Spanning Tree.

```
function KRUSKAL(G)
    Initialize an empty list X to store the edges of the MST
    Sort all edges E by increasing edge weight
    Initialize a disjoint-set (union-find) data structure for the vertices V
    for all edge (u, v) ∈ E in sorted order do
        if FIND(u) ≠ FIND(v) then
            UNION(u, v)
            Append (u, v) to X
        end if
    end for
    return X
end function
function FIND(v)
    if v is not the root then
        return FIND(parent of v)
    else
        return v
    end if
end function
function UNION(u, v)
    Make the root of u the parent of the root of v
end function
```

- Represent the graph $G = (V, E)$ with vertices $V$ and a list of edges $E$ with weights.
- Ensure the edges are sorted by weight.

2. **Model Processing**:
   - Initialize a disjoint-set data structure for tracking connected components.
   - Execute the main loop of Kruskal's algorithm to process each edge in sorted order and build the MST.
3. **Evaluation**:
   - Verify that the MST contains $|V| - 1$ edges.
   - Check the total weight of the MST and compare it to known benchmarks for correctness.

### 6.4 Advantages and Limitations

**Advantages:**

- **Efficiency**: Kruskal's algorithm has a time complexity of $O(|E| \log |E|)$, making it efficient for sparse graphs [3, 4].
- **Simplicity**: Conceptually simple and easy to implement with sorting and the disjoint-set data structure.

- **Applications**: Suitable for network design, clustering, and constructing roadmaps or communication networks.

**Limitations:**

- **Sorting Overhead**: Sorting the edges can be computationally expensive for dense graphs.
- **Memory Usage**: The disjoint-set data structure requires additional memory for storing parent and rank information.
- **Not Suitable for Dynamic Graphs**: If edges are frequently added or removed, Kruskal's algorithm may need to be rerun entirely [11].

### 6.5 Analysis of the Minimum Spanning Trees Algorithm

Given a graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of weighted edges, the Minimum Spanning Tree (MST) is a subset of edges that connects all vertices with the minimum possible total edge weight and no cycles. **Kruskal's Algorithm** is a greedy method for finding the MST by iteratively adding the smallest edge that does not form a cycle.

#### Time Complexity Analysis

Kruskal's algorithm processes the edges of a graph $G = (V, E)$ in sorted order to construct the MST. The time complexity can be broken down as follows [3, 4, 11]:

1. Sorting Edges: The edges $E$ are sorted by their weights. Sorting $|E|$ edges takes $O(|E| \log |E|)$ time using efficient sorting algorithms such as *MergeSort* or *HeapSort*.
2. Union-Find Operations: The algorithm uses the **disjoint-set (union-find)** data structure to manage connected components and detect cycles. - Each edge requires two 'Find' operations and possibly one 'Union' operation. With path compression and union by rank, these operations take $O(\log |V|)$ time on average. - For $|E|$ edges, the total time for union-find operations is $O(|E| \log |V|)$.

#### Overall Time Complexity:

Combining the sorting and union-find steps, the total time complexity of Kruskal's algorithm is:

$$O(|E| \log |E| + |E| \log |V|).$$

Since $|E| \geq |V| - 1$ in a connected graph, this simplifies to:

$$O(|E| \log |E|).$$

#### Summary of Time Complexity Analysis

- Time Complexity: $O(|E| \log |E|)$
- Space Complexity: $O(|V|)$ for the disjoint-set data structure.

Kruskal's algorithm is efficient for sparse graphs, where $|E|$ is much smaller than $|V|^2$, and the sorting step dominates the running time.

### 6.6 Correctness Proof

The correctness of Kruskal's algorithm can be proven using the principles of **greedy algorithms** and the properties of Minimum Spanning Trees [2, 3, 4].

**Proof Outline:**

1. Greedy Choice Property: Kruskal's algorithm selects the smallest edge that does not form a cycle. This choice is locally optimal and contributes to a globally optimal solution (i.e., the MST).
2. Cycle Prevention: The disjoint-set (union-find) structure ensures that an edge is added only if it connects two different components, thereby preventing cycles.
3. Spanning Tree Property: By adding $|V| - 1$ edges without forming cycles, the algorithm guarantees that all vertices are connected, forming a spanning tree.
4. Minimal Weight: Suppose there exists a different spanning tree $T'$ with a lower total weight than the tree $T$ produced by Kruskal's algorithm. It can be shown (by exchanging edges) that $T'$ must include the edges selected by Kruskal's algorithm, leading to a contradiction. Therefore, $T$ is a Minimum Spanning Tree.

**Conclusion:**

By the principles of greedy algorithms, Kruskal's algorithm correctly constructs a Minimum Spanning Tree.

**Summary of Correctness Proof**

- Greedy Choice: The algorithm selects the smallest edge that maintains the spanning tree property.
- Cycle Prevention: The union-find structure ensures no cycles are formed.
- Optimality: The resulting spanning tree has the minimum total edge weight.

Kruskal's algorithm is guaranteed to produce a correct MST for any connected, weighted graph.

### 6.7 Summary of Minimum Spanning Trees Algorithm

**Kruskal's Algorithm** is a greedy approach to finding the *Minimum Spanning Tree (MST)* of a connected, weighted graph $G = (V, E)$. The algorithm selects edges in increasing order of weight while ensuring no cycles are formed by using a disjoint-set (union-find) structure. It efficiently constructs the MST with a time complexity of $O(|E| \log |E|)$, making it suitable for sparse graphs. Kruskal's algorithm is widely used in applications such as clustering, network design, and data analysis where identifying minimal connections is crucial [3, 11]. The algorithm guarantees correctness through the greedy choice and cycle prevention properties, ensuring the resulting tree connects all vertices with the smallest possible total edge weight [2].

# 7 Decision Trees

**Decision Tree Traversal** is a fundamental algorithm for making predictions using a decision tree. A decision tree is a hierarchical model that represents decisions and their possible outcomes as a tree-like structure. The algorithm starts at the root node and traverses the tree based on input feature values $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$, making decisions at each internal node until a leaf node is reached. Decision trees are commonly used in classification and regression tasks. In regression analysis, decision trees can approximate complex, non-linear relationships between features and target variables by recursively partitioning the input space [6].

## 7.1 Mathematical Formulations

A decision tree can be formally defined as a directed graph $G = (V, E)$, where:

- $V$: Set of nodes, including decision nodes and leaf nodes.
- $E$: Set of directed edges connecting nodes.

The traversal is guided by a feature vector $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$. At each decision node $v$:

$$\text{Decision: } \begin{cases} v \leftarrow \text{LeftChild}(v) & \text{if } \mathbf{X}[\text{feature}] \leq \text{threshold}, \\ v \leftarrow \text{RightChild}(v) & \text{otherwise.} \end{cases}$$

The algorithm stops at a leaf node $v$, where the output $y$ is defined as:

$$y = \text{NodeOutcome}(v).$$

## 7.2 Decision Trees Algorithm

Given a decision tree $G = (V, E)$, the algorithm starts at the root node and evaluates the feature values $\mathbf{X}$ against thresholds at each decision node. Based on the comparison, it traverses either the left or right child until a leaf node is reached. The predicted outcome or class label is returned from the leaf node.

**Algorithm 4** Decision Tree Traversal

**Require:** $G = (V, E)$ is a decision tree, $s$ is the root node, $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$ is the input vector
**Ensure:** Predicted outcome or class label

```
function TraverseDecisionTree(G, s, X)
    Initialize v ← s                                  ▷ Start at the root node
    while v is not a leaf node do
        feature ← NodeFeature(v)                      ▷ Feature to evaluate at node v
        threshold ← NodeThreshold(v)                  ▷ Threshold at node v
        if X[feature] ≤ threshold then
            v ← LeftChild(v)
        else
            v ← RightChild(v)
        end if
    end while
    return NodeOutcome(v)                             ▷ Return the outcome at the leaf node
end function
```

### 7.3 Implementation Details

To use the decision tree traversal algorithm effectively, consider the following steps:

1. **Data Preprocessing**:
   - Clean the input data $\mathbf{X}$ and ensure that all features correspond to the expected feature set in the decision tree.
   - Handle missing values and normalize input features if necessary.
2. **Model Processing**:
   - Traverse the tree starting at the root node $s$.
   - At each decision node, evaluate the feature and threshold to decide the traversal path.
3. **Evaluation**:
   - Verify the output by comparing the predicted outcome against the expected label.
   - Analyze the decision path for interpretability, particularly for critical predictions.

### 7.4 Advantages and Limitations

**Advantages:**

- **Interpretability**: The traversal path provides a clear explanation of the decision process.
- **Non-Parametric**: Decision trees make no assumptions about the underlying data distribution, making them versatile.
- **Efficiency**: Traversal has a time complexity of $O(d)$, where $d$ is the depth of the tree.

**Limitations:**

- **Overfitting**: Decision trees can overfit the training data, leading to poor generalization.
- **Bias from Data Splits**: The algorithm's performance heavily depends on the quality of splits at each node.
- **Instability**: Small changes in the training data can result in different tree structures.

### 7.5 Analysis of the Decision Trees Algorithm

The Decision Tree Traversal Algorithm processes input data $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$ by traversing from the root node to a leaf node. The time complexity depends on the depth $d$ of the decision tree and the number of features $N$ in the input vector:

- Node Traversal: - At each node $v$, the algorithm evaluates a single feature and compares it to a threshold. This operation takes $O(1)$ time. - The traversal progresses through $d$ nodes, where $d$ is the depth of the tree.
- Tree Depth: - In a balanced tree, $d \approx \log(|V|)$, where $|V|$ is the number of nodes. - In an unbalanced tree, $d$ can be as large as $|V|$.

**Overall Time Complexity:**

For a balanced decision tree, the traversal complexity is:

$$O(\log |V|),$$

while for an unbalanced tree, it can degrade to:

$$O(|V|).$$

**Summary of Time Complexity Analysis**

- Time Complexity: $O(d)$, where $d$ is the depth of the tree.
- Space Complexity: $O(1)$, as the algorithm operates iteratively without additional storage requirements.

The algorithm is efficient for balanced decision trees but may become slower for deep, unbalanced trees.

### 7.6 Correctness Proof

The correctness of the Decision Tree Traversal Algorithm can be proven using induction on the structure of the decision tree.

**Base Case:**

For a tree with a single node (leaf node), the algorithm correctly returns the outcome stored at the node, as no comparisons are needed.

**Inductive Hypothesis:**

Assume the algorithm correctly traverses any subtree with depth $k$ and returns the correct outcome based on the input vector $\mathbf{X}$.

**Inductive Step:**

For a tree with depth $k+1$: - The algorithm evaluates the feature and threshold at the root node. - Based on the comparison, it correctly selects either the left or right subtree. - By the inductive hypothesis, the algorithm correctly traverses the selected subtree (depth $k$) and returns the correct outcome.

**Conclusion:**

By induction, the algorithm correctly traverses any decision tree and computes the appropriate outcome or class label.

**Summary of Correctness Proof**

- Inductive Proof: The algorithm's correctness is established by induction on the depth of the decision tree.
- Feature Evaluation: Each decision node evaluates the correct feature and threshold.
- Outcome Guarantee: The algorithm always terminates at a leaf node, ensuring a valid output.

The Decision Tree Traversal Algorithm is guaranteed to produce accurate results for any well-defined decision tree.

### 7.7 Summary - Decision Trees Algorithm

The Decision Tree Traversal Algorithm processes input data $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$ by traversing a hierarchical decision structure $G = (V, E)$ from the root node to a leaf node. At each decision node, the algorithm evaluates a feature value against a threshold to determine the traversal path. The algorithm has a time complexity of $O(d)$, where $d$ is the depth of the tree, and it guarantees correctness by following the decision rules encoded in the tree. Decision trees are widely used in classification, regression, and data analysis due to their interpretability, efficiency, and non-parametric nature [6].

# 8 Bayesian Networks

The Bayesian Networks algorithm provides a structured way to perform probabilistic reasoning in complex systems using a directed acyclic graph (DAG). Each node represents a random variable, while directed edges represent conditional dependencies between variables. The algorithm is designed for tasks such as regression analysis, where relationships among variables are critical, and applications such as medical diagnosis, risk assessment, and causal inference in machine learning. Bayesian Networks leverage the conditional probability distribution (CPD) at each node to represent the joint distribution of the system efficiently [10, 8].

## 8.1 Mathematical Formulations

A Bayesian Network is a DAG $G = (V, E)$ that represents a joint probability distribution:

$$P(X_1, X_2, \ldots, X_N) = \prod_{i=1}^{N} P(X_i \mid \text{Parents}(X_i)),$$

where $X_i$ is a node in the graph, and $\text{Parents}(X_i)$ represents its parent nodes in the DAG.

The goal of the algorithm is to compute the posterior probability of a query variable $Q$, given evidence $\mathbf{X}$:

$$P(Q \mid \mathbf{X}) = \frac{P(Q, \mathbf{X})}{P(\mathbf{X})}.$$

This involves:

Marginalization: Sum over all hidden variables $H$:

$$P(Q, \mathbf{X}) = \sum_{H} \prod_{i=1}^{N} P(X_i \mid \text{Parents}(X_i)).$$

Normalization: Normalize the joint probability:

$$P(Q \mid \mathbf{X}) = \frac{P(Q, \mathbf{X})}{\sum_{Q} P(Q, \mathbf{X})}.$$

## 8.2 Bayesian Networks Algorithm

Given a directed acyclic graph (DAG) $G = (V, E)$, where $V$ is the set of vertices representing random variables and $E$ is the set of directed edges representing dependencies, the following algorithm performs probabilistic inference using a Bayesian Network. Each node is associated with a conditional probability distribution (CPD) that quantifies the effect of its parents.

The Bayesian Networks algorithm operates in three key steps:

Initialize Factors: Extract CPDs from the DAG and represent each node as a factor $\phi_v$, encoding the dependencies of the node and its parents.

Variable Elimination: Marginalize out hidden variables $H$ by summing over their values, combining relevant factors.

**Algorithm 5** Bayesian Network Inference Algorithm

**Require:** $G = (V, E)$: A Bayesian Network, $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$: Observed variables, $Q$: Query variable
**Ensure:** $P(Q \mid \mathbf{X})$: Posterior probability of $Q$ given evidence $\mathbf{X}$

**function** BayesianNetworkInference($G, \mathbf{X}, Q$)
    **Step 1: Initialize Factors**
    Extract CPDs from $G$ and represent each node $v$ as a factor $\phi_v$ based on its CPD.
    **Step 2: Apply Variable Elimination**
    Identify all variables $H$ that are not in $Q \cup \text{ObservedVariables}$.
    **for all** variable $h \in H$ **do**
        Identify all factors involving $h$.
        Compute a new factor by summing out $h$ (marginalization):

$$\phi_{\text{new}} = \sum_h \prod \phi_{\text{relevant}}.$$

        Replace the old factors with $\phi_{\text{new}}$.
    **end for**
    **Step 3: Normalize the Result**
    Multiply remaining factors to compute the unnormalized joint distribution:

$$P(Q, \mathbf{X}) = \prod \phi_{\text{remaining}}.$$

    Normalize to obtain the posterior probability:

$$P(Q \mid \mathbf{X}) = \frac{P(Q, \mathbf{X})}{\sum_Q P(Q, \mathbf{X})}.$$

    **return** $P(Q \mid \mathbf{X})$
**end function**

Normalize: Compute the posterior probability $P(Q \mid \mathbf{X})$ by dividing the joint probability by its normalization constant.

### 8.3 Implementation Details

The implementation of the Bayesian Networks algorithm involves the following steps:

1. **Data Preprocessing**: To implement the Bayesian Networks algorithm, the first step is to construct the directed acyclic graph (DAG) $G = (V, E)$, where the nodes $V$ represent variables and the edges $E$ represent dependencies between them. Once the graph structure is defined, the conditional probability distributions (CPDs) for each node are estimated from the data using parameter estimation techniques, such as maximum likelihood estimation (MLE) or Bayesian estimation.
2. **Model Processing**: The implementation of the Bayesian Networks algorithm begins by initializing the factors $\phi_v$ from the conditional probability distributions (CPDs) associated with each node. Next, variable elimination is applied by marginalizing out hidden variables that are not part of the query or evidence. Finally, the remaining factors are multiplied to compute the unnormalized joint probability, which forms the basis for further computations in the inference process.
3. **Evaluation**: The final step in the Bayesian Networks algorithm is to normalize the result to compute the posterior probability, ensuring that the probabilities sum to one. The performance of the algorithm is then evaluated by comparing its predictions against the ground truth in specific applications, such as medical diagnosis or risk analysis, to assess its accuracy and effectiveness.

### 8.4 Advantages and Limitations

The Bayesian Networks algorithm offers several advantages. It provides a **compact representation**, efficiently representing complex joint distributions by leveraging conditional independencies. Additionally, it enables **causal reasoning**, offering a framework for causal inference and understanding dependencies among variables. Furthermore, the algorithm demonstrates **flexibility**, as it can handle both discrete and continuous variables and supports missing data through marginalization.

The Bayesian Networks algorithm has several limitations. **Scalability** is a concern, as the computational complexity increases with the number of variables and edges, making inference challenging for large networks. Additionally, **structure learning**, which involves determining the DAG from data, is NP-hard and often requires heuristic methods. Finally, **parameter estimation** can be demanding, as accurately estimating the conditional probability distributions (CPDs) may require large datasets, particularly when dealing with complex dependencies [7].

### 8.5 Analysis of the Bayesian Networks Algorithm

The time complexity of the Bayesian Networks algorithm depends on the structure of the directed acyclic graph (DAG) and the specific steps involved in the inference process. The key contributors to the complexity are:

1. **Initialization of Factors**: Extracting the conditional probability distributions (CPDs) for each node and representing them as factors requires $O(|V|)$ operations, where $|V|$ is the number of vertices (nodes) in the graph.
2. **Variable Elimination**: The complexity of variable elimination depends on the *treewidth* of the graph, which is a measure of how "tree-like" the graph structure is. For a graph with treewidth $w$, eliminating a variable involves summing over $O(2^w)$ possible values. For a graph with $n$ nodes and $m$ edges, the complexity of this step is approximately:
$$O(n \cdot 2^w),$$
where $w$ is generally much smaller than $n$ for sparse graphs.
3. **Normalization**: Computing the posterior probability involves normalizing the joint distribution. This step requires summing over all possible values of the query variable $Q$, resulting in $O(|Q|)$ operations.
4. **Overall Complexity**: The total time complexity is dominated by the variable elimination step and can be expressed as:
$$O(n \cdot 2^w),$$
where $w$ is the treewidth of the graph.

### Summary of Time Complexity Analysis

The Bayesian Networks algorithm is efficient for graphs with small treewidths, making it suitable for sparse or tree-like graphs. However, it can become computationally expensive for dense graphs or those with high treewidth.

## 8.6 Correctness Proof

The correctness of the Bayesian Networks algorithm is based on its ability to compute the posterior probability $P(Q \mid \mathbf{X})$ accurately using the properties of probabilistic graphical models. The proof relies on the following key principles:

### Factorization of Joint Probability

The Bayesian Network represents the joint probability distribution as a product of local conditional probabilities:
$$P(X_1, X_2, \ldots, X_N) = \prod_{i=1}^{N} P(X_i \mid \text{Parents}(X_i)).$$
This factorization is consistent with the dependencies encoded in the DAG.

### Variable Elimination

Variable elimination ensures that all hidden variables $H$ are marginalized out:
$$P(Q, \mathbf{X}) = \sum_{H} \prod_{i=1}^{N} P(X_i \mid \text{Parents}(X_i)).$$

The algorithm combines and reduces factors iteratively, preserving the correctness of the computation.

### Normalization

The final step normalizes the joint distribution:

$$P(Q \mid \mathbf{X}) = \frac{P(Q, \mathbf{X})}{P(\mathbf{X})}.$$

This ensures that the posterior probability sums to one, as required by the axioms of probability.

### Summary of Correctness Proof

The Bayesian Networks algorithm is correct because it adheres to the principles of probabilistic inference, factorization, and marginalization. It ensures that the posterior probability $P(Q \mid \mathbf{X})$ is computed accurately for any query and evidence.

## 8.7 Summary - Bayesian Networks Algorithm

The Bayesian Networks algorithm provides a structured framework for performing probabilistic inference in systems with complex dependencies. By leveraging the directed acyclic graph (DAG) structure, the algorithm efficiently computes posterior probabilities through factorization, variable elimination, and normalization. Its scalability depends on the graph's treewidth, making it suitable for applications in medical diagnosis, risk assessment, and machine learning. The algorithm's correctness is guaranteed by its adherence to the principles of probabilistic reasoning and graphical model theory [10, 7, 8].

# 9 Louvain Algorithm for Graph-Based Feature Engineering

The **Louvain algorithm** is a greedy optimization method for detecting communities in weighted graphs. It maximizes *modularity*, a measure of the strength of division of a graph into communities, by iteratively optimizing the modularity locally and aggregating communities into super-nodes. In the context of *graph-based feature engineering*, the Louvain algorithm is used to identify groups of nodes with dense interconnections, which can then serve as high-level features in machine learning models. For example, in regression analysis, these features can represent clusters of strongly related data points, improving the interpretability and performance of predictive models [1, 5].

## 9.1 Mathematical Formulations

The modularity $Q$ of a graph partition is defined as:

$$Q = \frac{1}{2m} \sum_{i,j} \left[ A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j),$$

where:

- $A_{ij}$ is the weight of the edge between nodes $i$ and $j$.
- $k_i$ is the sum of weights of edges connected to node $i$.
- $m = \frac{1}{2} \sum_{i,j} A_{ij}$ is the total weight of all edges in the graph.
- $\delta(c_i, c_j)$ is 1 if nodes $i$ and $j$ are in the same community, and 0 otherwise.

The algorithm seeks to maximize $Q$ by iteratively adjusting community assignments and aggregating communities.

## 9.2 Louvain Algorithm for Graph-Based Feature Engineering

Given a graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of weighted edges, the Louvain algorithm identifies communities (clusters of highly connected nodes) using modularity optimization. The algorithm is particularly useful for extracting features from graphs for machine learning tasks.

The Louvain algorithm is a two-phase process:

**Phase 1: Local Modularity Optimization**: - Each node is moved to the community that yields the maximum increase in modularity $\Delta Q$.

**Phase 2: Community Aggregation**: - Communities are aggregated into super-nodes, and the graph is rebuilt with updated edge weights representing inter-community connections.

These phases are repeated until modularity converges. The final community assignments can be used as graph-based features for machine learning models, capturing underlying relationships in the data.

## 9.3 Implementation Details

To implement the Louvain algorithm for feature engineering, the following steps should be followed:

**Algorithm 6** Louvain Algorithm for Community Detection

**Require:** $G = (V, E)$: A weighted graph, modularity function $Q$
**Ensure:** $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$, where $\mathbf{x}_i$ is the community assignment of vertex $v_i$

**function** LOUVAIN($G$)
    Initialize $\mathbf{X}$, where each vertex $v \in V$ is its own community
    modularity_gain $\leftarrow 1$ ▷ Start with positive modularity gain
    **while** modularity_gain $> 0$ **do** ▷ Repeat until modularity converges
        PHASE1LOCALOPTIMIZATION($G, \mathbf{X}$)
        $G \leftarrow$ PHASE2COMMUNITYAGGREGATION($G, \mathbf{X}$)
        Compute $Q_{\text{new}}$ for the updated graph
        modularity_gain $\leftarrow Q_{\text{new}} - Q_{\text{old}}$
        $Q_{\text{old}} \leftarrow Q_{\text{new}}$
    **end while**
    **return X**
**end function**
**function** PHASE1LOCALOPTIMIZATION($G, \mathbf{X}$)
    **for all** vertex $v \in V$ **do**
        current_community $\leftarrow \mathbf{X}[v]$
        Compute modularity change $\Delta Q$ for moving $v$ to each neighboring community
        best_community $\leftarrow$ argmax($\Delta Q$)
        **if** best_community $\neq$ current_community **then**
            Move $v$ to best_community
            Update $\mathbf{X}[v]$
        **end if**
    **end for**
**end function**
**function** PHASE2COMMUNITYAGGREGATION($G, \mathbf{X}$)
    Create a new graph $G'$, where each community in $\mathbf{X}$ is a single node
    Aggregate edge weights between nodes in the same community
    **return** $G'$
**end function**

1. **Data Preprocessing**: To implement the Louvain algorithm, the first step is to construct a weighted graph $G = (V, E)$, where the nodes $V$ represent entities and the edge weights $E$ represent relationships such as correlation or similarity. Additionally, it is important to normalize the edge weights if necessary to ensure that the graph accurately reflects the relative strengths of the connections between nodes.
2. **Model Processing**:The Louvain algorithm begins by initializing each node as its own community. Modularity is then iteratively optimized by moving nodes to neighboring communities to maximize the modularity gain. Once the local optimization is complete, communities are aggregated into super-nodes, and the graph is rebuilt to represent these aggregated structures. This process is repeated until modularity converges, ensuring an optimal partitioning of the graph into communities.
3. **Evaluation**: To evaluate the effectiveness of the Louvain algorithm, the final modularity $Q$ is computed to assess the quality of the detected communities. The community assignments can then be used as features in machine learning models, where their impact on model performance is evaluated to determine the utility of the graph-based feature engineering process.

### 9.4 Advantages and Limitations

The Louvain algorithm is highly scalable, making it efficient for large graphs with a time complexity of $O(|V| \log |V|)$ for sparse graphs. It also offers interpretability, as the communities identified by the algorithm provide meaningful and understandable features that can be used effectively in machine learning tasks. Additionally, the algorithm is flexible and versatile, as it can be applied to both weighted and unweighted graphs, as well as directed and undirected graphs, making it suitable for a wide range of applications.

The Louvain algorithm has certain limitations. One notable issue is the **resolution limit**, where it may fail to detect small communities in large graphs [5]. Additionally, the algorithm exhibits **initialization sensitivity**, meaning that the results can depend on the initial configuration of communities. Furthermore, the algorithm may encounter **modularity saturation**, where the modularity metric does not always capture meaningful structures in certain types of graphs, potentially limiting its effectiveness in specific applications.

### 9.5 Analysis of the Louvain Algorithm for Graph-Based Feature Engineering

The Louvain algorithm is computationally efficient due to its greedy modularity optimization process, making it suitable for large-scale graphs. The time complexity can be analyzed as follows:

- **Phase 1: Local Modularity Optimization**: - Each node is visited and moved to the community that maximizes the modularity gain $\Delta Q$. For a graph with $|V|$ vertices and $|E|$ edges, the complexity of this step is $O(|V| + |E|)$ for each pass over the graph.
- **Phase 2: Community Aggregation**: - Communities are aggregated into super-nodes, and a new graph is built. This step requires traversing all edges to compute the new edge weights, resulting in a complexity of $O(|E|)$.
- **Iterative Optimization**: - The two phases are repeated until modularity converges. Let $k$

be the number of iterations. The overall complexity of the Louvain algorithm is:

$$O(k \cdot (|V| + |E|)).$$

In practice, $k$ is small and typically bounded by 10 for most graphs.

### Summary of Time Complexity Analysis

The Louvain algorithm has a time complexity of $O(k \cdot (|V| + |E|))$, which scales linearly with the number of nodes and edges for sparse graphs. This efficiency makes it ideal for large-scale applications in graph-based feature engineering.

## 9.6 Correctness Proof

The correctness of the Louvain algorithm relies on its ability to maximize modularity $Q$ in each iteration. The proof can be outlined as follows:

### Greedy Modularity Optimization

In **Phase 1**, each node $v$ is moved to the community that yields the largest increase in modularity $\Delta Q$. This step guarantees that modularity does not decrease in each iteration, as only moves that improve $Q$ are performed.

### Community Aggregation

In **Phase 2**, the graph is reduced by aggregating communities into super-nodes. This step preserves the modularity of the graph, as the edge weights between super-nodes represent the total weight of edges between the corresponding communities in the original graph.

### Convergence

The iterative process alternates between the two phases until no further improvement in modularity $Q$ is possible. Since modularity is bounded (it cannot exceed 1), the algorithm is guaranteed to converge after a finite number of iterations.

### Summary of Correctness Proof

The Louvain algorithm is correct because it iteratively maximizes modularity using a greedy optimization approach. The algorithm guarantees convergence, as each step either improves or preserves modularity, and modularity has an upper bound.

## 9.7 Summary - Louvain Algorithm for Graph-Based Feature Engineering

The Louvain algorithm is a scalable and effective method for community detection in graphs, making it particularly valuable for graph-based feature engineering in machine learning. By iteratively maximizing modularity, the algorithm identifies dense clusters of nodes that can serve as meaningful features. Its time complexity of $O(k \cdot (|V| + |E|))$ ensures efficiency even for large graphs, and its correctness is rooted in the modularity optimization process. The algorithm's ability to handle

weighted, unweighted, directed, and undirected graphs further enhances its applicability across a wide range of data science tasks [1, 5].

# 10 Planetoid for Semi-Supervised Learning on Graphs

The Planetoid algorithm is a semi-supervised learning framework designed for graph-based problems such as node classification and text classification. It leverages both node features and graph structure to improve classification performance. The algorithm combines supervised learning on labeled nodes with a graph-based regularization term that ensures connected nodes in the graph have similar predictions. By combining these two aspects, Planetoid provides a robust framework for problems where labeled data is scarce [12].

## 10.1 Mathematical Formulations

Given a graph $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of edges, each node $v \in V$ is associated with a feature vector $\mathbf{x}_v \in \mathbb{R}^d$. The goal is to predict the label $\mathbf{y}_v$ for each node using a neural network $f_\theta$, parameterized by $\theta$.

The algorithm optimizes the following loss function:

$$\mathcal{L} = \mathcal{L}_{\text{sup}} + \lambda \cdot \mathcal{L}_{\text{reg}},$$

where:

**Supervised Loss**:

$$\mathcal{L}_{\text{sup}} = - \sum_{v \in V_{\text{labeled}}} \mathbf{y}_v \cdot \log f_\theta(\mathbf{x}_v),$$

with $\mathbf{y}_v$ as the true one-hot encoded label of node $v$.

**Regularization Loss**:

$$\mathcal{L}_{\text{reg}} = \sum_{(u,v) \in E} \|f_\theta(\mathbf{x}_u) - f_\theta(\mathbf{x}_v)\|^2,$$

which encourages connected nodes to have similar predictions.

The parameters $\theta$ are updated using gradient descent:

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta \mathcal{L},$$

where $\alpha$ is the learning rate.

## 10.2 Planetoid Algorithm for Semi-Supervised Learning on Graphs

Given a graph $G = (V, E)$, where $V$ is the set of vertices (or nodes) representing entities with associated features, and $E$ is the set of edges indicating relationships between these entities, the following algorithm combines node features and graph structure to perform semi-supervised learning. Each node is associated with a feature vector, and the algorithm utilizes both labeled and unlabeled nodes to optimize the classification performance across the graph.

The Planetoid algorithm operates in four key steps:

1. **Model Initialization**: Initialize the neural network parameters $\theta$ randomly.
2. **Prediction**: For each node, compute the predicted label distribution $f_\theta(\mathbf{x}_v)$.

**Algorithm 7** Planetoid Algorithm for Node Classification

**Require:** $G = (V, E)$: A graph with nodes $V$ and edges $E$, $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$: Node feature matrix, $\mathbf{Y}$: Labels for a subset of nodes, $f_\theta$: Neural network model with parameters $\theta$, $\lambda$: Regularization coefficient, $\alpha$: Learning rate, $T$: Number of training iterations.

**Ensure:** Trained model $f_\theta$ for node classification.

**function** PLANETOID($G, \mathbf{X}, \mathbf{Y}, f_\theta, \lambda, \alpha, T$)

**Step 1: Initialize Model**

Initialize model parameters $\theta$ randomly.

**for** $t = 1$ to $T$ **do**

**Step 2: Compute Predictions**

For each node $v \in V$, compute $f_\theta(\mathbf{x}_v)$, the predicted label distribution.

**Step 3: Compute Loss Function**

Calculate supervised loss on labeled nodes:

$$\mathcal{L}_{\text{sup}} = -\sum_{v \in V_{\text{labeled}}} \mathbf{y}_v \cdot \log f_\theta(\mathbf{x}_v),$$

where $\mathbf{y}_v$ is the one-hot encoded true label for node $v$.

Calculate regularization loss on graph structure:

$$\mathcal{L}_{\text{reg}} = \sum_{(u,v) \in E} \|f_\theta(\mathbf{x}_u) - f_\theta(\mathbf{x}_v)\|^2.$$

Combine losses:

$$\mathcal{L} = \mathcal{L}_{\text{sup}} + \lambda \cdot \mathcal{L}_{\text{reg}}.$$

**Step 4: Update Parameters**

Compute gradients $\nabla_\theta \mathcal{L}$.

Update $\theta$ using gradient descent:

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta \mathcal{L}.$$

**end for**

**return** $f_\theta$

**end function**

3. **Loss Computation**: Combine the supervised loss on labeled nodes with the graph regularization loss.
4. **Parameter Update**: Update $\theta$ using gradient descent to minimize the combined loss.

These steps are repeated for $T$ training iterations until convergence.

### 10.3 Implementation Details

1. **Data Preprocessing**: To implement the Planetoid algorithm, the graph $G = (V, E)$ is represented using a feature matrix $\mathbf{X}$, where each row corresponds to the feature vector of a node, and an adjacency matrix $\mathbf{A}$, which encodes the graph's edge connections. Node features and edge weights are normalized if necessary to ensure numerical stability and consistency across the dataset. Finally, the nodes are split into training, validation, and test sets to facilitate model training, hyperparameter tuning, and performance evaluation.
2. **Model Processing**: The Planetoid algorithm requires defining the neural network model $f_\theta$, specifying its architecture, such as fully connected layers, to map input node features to output predictions. Once the architecture is defined, the model parameters $\theta$ are initialized, typically using random initialization or pre-trained weights, to prepare the model for training.
3. **Evaluation**: After training, the model is evaluated on the test set using performance metrics such as accuracy or F1-score to measure its effectiveness in classification tasks. Additionally, the contribution of graph regularization to the model's performance is analyzed to understand how the graph structure improves predictions and generalization.

### 10.4 Advantages and Limitations

The Planetoid algorithm offers several key advantages. It **combines node features and graph structure**, effectively leveraging both labeled data and graph relationships to achieve improved classification performance. The algorithm is **effective for semi-supervised learning**, as it performs well even when only a small portion of the nodes in the graph are labeled. Additionally, it demonstrates **scalability**, making it suitable for large, sparse graphs commonly encountered in real-world applications.

The Planetoid algorithm has several limitations. It exhibits **hyperparameter sensitivity**, requiring careful tuning of parameters such as $\lambda$ (regularization coefficient) and $\alpha$ (learning rate) to achieve optimal performance. Additionally, it incurs **computational overhead** when applied to densely connected graphs, as the regularization term scales with the number of edges. Finally, the algorithm's performance is highly **feature-dependent**, relying heavily on the quality of the input node features to make accurate predictions.

### 10.5 Analysis of the Planetoid Algorithm for Semi-Supervised Learning on Graphs

The time complexity of the Planetoid algorithm depends on the size of the graph $G = (V, E)$, the dimensionality of the feature vectors, and the number of training iterations $T$. The primary contributors to the time complexity are:

1. **Feature Propagation and Predictions**: For each node $v \in V$, the algorithm computes the predicted label distribution $f_\theta(\mathbf{x}_v)$, which requires evaluating the neural network. For a

graph with $|V| = n$ nodes and a neural network with $L$ layers and $d$ features, this step has complexity:

$$O(n \cdot L \cdot d^2).$$

2. **Loss Function Computation**: The supervised loss involves summing over $|V_{\text{labeled}}|$, the set of labeled nodes. The regularization loss involves summing over the edges $E$, requiring $O(|E|)$ operations.

3. **Gradient Updates**: Backpropagation through the neural network involves computing gradients with respect to the loss, with complexity proportional to the network size. This adds:

$$O(n \cdot L \cdot d^2).$$

4. **Overall Complexity**: For $T$ training iterations, the total time complexity is:

$$O(T \cdot (n \cdot L \cdot d^2 + |E|)).$$

**Summary of Time Complexity Analysis**

The Planetoid algorithm is computationally efficient for large, sparse graphs, as the regularization term scales with the number of edges $|E|$. The dominant factor is the neural network evaluation, which scales with the number of nodes $n$, the number of layers $L$, and the feature dimensionality $d$.

### 10.6 Correctness Proof

The correctness of the Planetoid algorithm relies on its ability to optimize a loss function that combines supervised learning and graph-based regularization. The proof involves the following components:

**Supervised Loss Minimization**

The supervised loss $\mathcal{L}_{\text{sup}}$ ensures that the predictions $f_\theta(\mathbf{x}_v)$ for labeled nodes $v \in V_{\text{labeled}}$ match their true labels $\mathbf{y}_v$. By minimizing $\mathcal{L}_{\text{sup}}$, the algorithm achieves high accuracy on labeled nodes, assuming sufficient model capacity.

**Regularization Loss Minimization**

The regularization loss $\mathcal{L}_{\text{reg}}$ ensures that the predictions for connected nodes $u, v \in E$ are similar. This term leverages the graph structure to propagate label information from labeled to unlabeled nodes, effectively regularizing the model.

**Convergence**

The gradient descent updates:

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta \mathcal{L}$$

guarantee convergence to a local minimum of the combined loss $\mathcal{L}$, assuming a sufficiently small learning rate $\alpha$. Convergence ensures that the model balances supervised accuracy and graph-based regularization.

### Summary of Correctness Proof

The Planetoid algorithm correctly learns node representations by minimizing a loss function that integrates supervised learning and graph-based regularization. The use of labeled data ensures predictive accuracy, while the regularization term leverages the graph structure to generalize to unlabeled nodes.

## 10.7 Summary - Planetoid Algorithm for Semi-Supervised Learning on Graphs

The Planetoid algorithm combines node features and graph structure to perform semi-supervised learning on graphs. By integrating supervised learning on labeled nodes with graph-based regularization, the algorithm achieves high performance in tasks such as node classification and text classification. The time complexity is efficient for large, sparse graphs, and the correctness of the algorithm is guaranteed by its optimization of a well-defined loss function. Applications include citation networks, social networks, and biological networks, where graph structure provides critical information for accurate predictions [12].

# 11 Spectral Clustering Algorithm for Dimensionality Reduction and Clustering

The Spectral Clustering algorithm is a graph-based technique for clustering and dimensionality reduction. It leverages the eigenvalues and eigenvectors of the graph Laplacian to embed nodes into a lower-dimensional space that preserves structural information. This embedding is then used to cluster the nodes into groups, capturing the underlying relationships encoded in the graph. Spectral clustering is particularly well-suited for applications in graph embeddings and clustering data points in graph structures, where relationships among data points are critical [9].

## 11.1 Mathematical Formulations

Given a graph $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of edges, the algorithm constructs the graph Laplacian and uses its spectral properties for clustering.

**Graph Laplacian:** The graph Laplacian $L$ is defined as:

$$L = D - A,$$

where $D$ is the degree matrix (a diagonal matrix with $D_{ii} = \sum_j A_{ij}$) and $A$ is the adjacency matrix.

The normalized graph Laplacian is defined as:

$$L_{\text{norm}} = D^{-1/2} L D^{-1/2}.$$

**Eigenvalue Decomposition:** The eigenvalues and eigenvectors of $L$ or $L_{\text{norm}}$ are computed. The $k$ smallest eigenvectors, corresponding to the $k$ smallest eigenvalues, form a matrix $U \in \mathbb{R}^{|V| \times k}$, which provides a low-dimensional embedding of the graph.

**Clustering:** The rows of $U$ are normalized to unit length:

$$U'_{ij} = \frac{U_{ij}}{\sqrt{\sum_j U_{ij}^2}}.$$

The normalized rows are clustered using $k$-means to assign each node to one of $k$ clusters.

## 11.2 Spectral Clustering Algorithm for Dimensionality Reduction and Clustering

Given a graph $G = (V, E)$, where $V$ is the set of vertices (or nodes) and $E$ is the set of edges representing relationships between these nodes, the Spectral Clustering algorithm uses the graph structure to partition the nodes into meaningful clusters. By constructing and analyzing the eigenvalues and eigenvectors of the graph Laplacian, the algorithm embeds the nodes into a lower-dimensional space that preserves structural information. This representation is then used to group nodes into clusters, leveraging the global connectivity of the graph.

The Spectral Clustering algorithm consists of the following steps:

1. **Graph Representation:** Construct the degree matrix $D$ and adjacency matrix $A$. Compute the graph Laplacian $L$ or normalized Laplacian $L_{\text{norm}}$.

**Algorithm 8** Spectral Clustering Algorithm

**Require:** $G = (V, E)$: A graph with nodes $V$ and edges $E$, $k$: Number of clusters
**Ensure:** Cluster assignments for the nodes $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$

**function** SpectralClustering($G, k$)
  **Step 1: Construct the Laplacian Matrix**
  Compute the degree matrix $D$, where $D_{ii} = \sum_j A_{ij}$
  Compute the adjacency matrix $A$, where $A_{ij}$ is the weight of the edge between $v_i$ and $v_j$
  Compute the graph Laplacian $L$:

$$L = D - A$$

  Optionally, use the normalized Laplacian:

$$L_{\text{norm}} = D^{-1/2} L D^{-1/2}$$

  **Step 2: Compute Eigenvalues and Eigenvectors**
  Compute the eigenvalues and eigenvectors of $L$ or $L_{\text{norm}}$
  Select the $k$ smallest eigenvectors to form the matrix $U \in \mathbb{R}^{|V| \times k}$
  **Step 3: Normalize the Rows of $U$**
  Normalize each row of $U$ to have unit length:

$$U'_{ij} = \frac{U_{ij}}{\sqrt{\sum_j U_{ij}^2}}$$

  **Step 4: Apply k-Means Clustering**
  Apply the $k$-means algorithm to the rows of $U'$ to partition the nodes into $k$ clusters
  **return** Cluster assignments $\mathbf{X}$
**end function**

2. **Spectral Decomposition:** Compute the eigenvalues and eigenvectors of $L$ or $L_{\text{norm}}$, and select the $k$ smallest eigenvectors to form the matrix $U$.
3. **Normalization:** Normalize each row of $U$ to unit length.
4. **Clustering:** Apply $k$-means clustering to the normalized rows of $U$ to assign nodes to clusters.

### 11.3 Implementation Details

1. **Data Preprocessing:** To begin, construct the graph $G = (V, E)$ using the adjacency matrix $A$, where the relationships between data points define the edges. Normalize edge weights if necessary to ensure consistency across the graph, which helps maintain the integrity of the graph structure and improves the algorithm's performance.
2. **Model Processing:** The next step involves computing the graph Laplacian $L$ or the normalized Laplacian $L_{\text{norm}}$, which captures the structure of the graph. Following this, perform spectral decomposition to obtain the eigenvectors, which are essential for embedding the graph into a lower-dimensional space.
3. **Evaluation:** To evaluate the results, analyze the quality of the clusters using metrics such as the silhouette score or modularity, which provide insights into the effectiveness of the clustering. Additionally, visualize the embedding and clustering results to interpret the structure of the graph and gain a better understanding of the relationships among the nodes.

### 11.4 Advantages and Limitations

Spectral clustering offers several notable advantages. It can **identify non-convex clusters**, making it effective for detecting complex cluster shapes that other methods might miss. The algorithm leverages the **global structure of the graph** through the Laplacian matrix, ensuring that the clustering process captures comprehensive relationships within the data. Additionally, it provides **dimensionality reduction**, embedding the graph into a lower-dimensional space while preserving critical structural information.

**Limitations:** Despite its advantages, spectral clustering has some limitations. The **computational complexity** of the eigenvalue decomposition step can make it computationally expensive for large graphs, limiting its scalability. Additionally, the algorithm is **sensitive to the number of clusters** $k$, which must be specified beforehand and may not always be evident. Furthermore, the **quality of the input graph** plays a critical role, as the performance heavily depends on the accuracy and relevance of the graph representation, including the adjacency matrix and edge weights.

### 11.5 Analysis of the Spectral Clustering Algorithm

The Spectral Clustering algorithm is a powerful graph-based technique for clustering and dimensionality reduction. By analyzing the eigenvalues and eigenvectors of the graph Laplacian, the algorithm embeds nodes into a lower-dimensional space that preserves the structural properties of the graph. This representation enables meaningful partitioning of nodes into clusters. Applications of Spectral Clustering include graph embeddings, community detection, and clustering data points embedded in graph structures [9].

### 11.6 Time Complexity Analysis

The time complexity of the Spectral Clustering algorithm is determined by the following major steps:

1. **Graph Laplacian Construction:** Constructing the degree matrix $D$ and adjacency matrix $A$, and computing the graph Laplacian $L$, requires $O(|E|)$, where $|E|$ is the number of edges in the graph.
2. **Eigenvalue Decomposition:** Computing the eigenvalues and eigenvectors of the Laplacian $L$ or normalized Laplacian $L_{\text{norm}}$ involves spectral decomposition. This step is computationally expensive, requiring $O(|V|^3)$ in general, where $|V| = n$ is the number of nodes. For sparse graphs, this can be reduced to $O(kn^2)$, where $k$ is the number of eigenvectors computed.
3. **Row Normalization and $k$-Means Clustering:** Normalizing the rows of the eigenvector matrix and applying $k$-means clustering takes $O(nk^2)$, where $k$ is the number of clusters.

**Overall Complexity:** For large, sparse graphs, the overall complexity is dominated by the eigenvalue decomposition and is approximately:

$$O(|E| + kn^2).$$

### Summary of Time Complexity Analysis

Spectral Clustering is computationally feasible for small to medium-sized graphs but can become expensive for very large graphs due to the eigenvalue decomposition step. Its complexity depends on the size of the graph and the number of clusters.

## 11.7 Correctness Proof

The correctness of the Spectral Clustering algorithm is rooted in spectral graph theory and relies on the properties of the graph Laplacian and the spectral embedding.

**Graph Partitioning and Laplacian Properties:** The eigenvalues and eigenvectors of the Laplacian provide a natural way to partition the graph. The smallest eigenvector of $L$ (corresponding to the eigenvalue 0) represents the connected components of the graph. By selecting the $k$ smallest eigenvectors, the algorithm embeds the graph into a $k$-dimensional space that preserves the structure of the graph.

**Normalized Cuts and Clustering:** The normalized Laplacian minimizes the normalized cut, a metric that balances the cut size and cluster size. This ensures that the resulting clusters are both meaningful and balanced.

**Convergence:** The use of $k$-means clustering on the embedded space guarantees that the nodes are partitioned into $k$ clusters. While $k$-means converges to a local minimum of the clustering objective, the spectral embedding ensures that this local minimum corresponds to a meaningful graph partition.

### Summary of Correctness Proof

Spectral Clustering is guaranteed to produce clusters that optimize the normalized cut criterion. The use of the Laplacian's eigenvectors ensures that the embedding reflects the global structure of the graph, leading to meaningful clustering results.

## 11.8 Summary - Spectral Clustering Algorithm

The Spectral Clustering algorithm effectively combines graph structure and spectral properties to partition nodes into meaningful clusters. By embedding the graph into a lower-dimensional space using the eigenvectors of the graph Laplacian, it preserves the structural properties of the graph. Its applications include graph-based dimensionality reduction, community detection, and clustering in data science and machine learning. The algorithm's strengths lie in its ability to detect non-convex clusters and leverage global graph structure, while its limitations include computational complexity and sensitivity to the number of clusters $k$.

# 12 Evaluation Metrics for Graph Algorithms

The evaluation of graph algorithms is essential to assess their performance across various tasks, including traversal, clustering, classification, and community detection. The following metrics provide quantitative measures tailored to specific graph algorithms, enabling practitioners to compare and optimize methods for real-world applications.

### Purpose of Evaluation Metrics

Evaluation metrics help quantify the performance of graph algorithms in terms of efficiency, accuracy, and effectiveness. For example, traversal algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) are evaluated based on their ability to explore all vertices or edges, while community detection and clustering algorithms like the Louvain Algorithm and Spectral Clustering are assessed using modularity or silhouette scores.

### Traversal Algorithms

For algorithms like **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**, the following metrics are relevant:

- **Traversal Completeness:** Verifies whether all vertices or edges are explored during the traversal.
- **Time Complexity:** Measures the computational efficiency, typically $O(|V| + |E|)$ for BFS and DFS.
- **Space Complexity:** Evaluates memory usage, including queue (BFS) or stack (DFS) sizes.

### Minimum Spanning Trees (MST)

For MST algorithms, such as **Kruskal's Algorithm** and **Prim's Algorithm**, evaluation focuses on:

- **Total Weight:** The sum of edge weights in the spanning tree, which should be minimized.
- **Correctness:** Ensures the solution is a valid spanning tree with $|V| - 1$ edges and no cycles.
- **Time Complexity:** Depends on graph representation; typically $O(|E| \log |E|)$ for Kruskal's Algorithm.

### Decision Trees

For **Decision Trees**, the evaluation aligns with standard supervised learning metrics:

- **Accuracy:** Measures the proportion of correctly classified instances.
- **Precision, Recall, and F1-Score:** Assess the model's performance on imbalanced datasets.
- **Tree Depth:** Captures model complexity and its potential to overfit or underfit the data.

### Bayesian Networks

For **Bayesian Networks**, key evaluation metrics include:

- **Log-Likelihood:** Measures how well the model fits the observed data.
- **Posterior Accuracy:** Evaluates the accuracy of probabilistic inferences.
- **Computational Complexity:** Assesses the efficiency of probabilistic inference, which can grow exponentially with the number of variables.

### Louvain Algorithm for Graph-Based Feature Engineering

For the **Louvain Algorithm**, metrics for community detection and feature engineering include:

- **Modularity:** Quantifies the strength of community structure in the graph. Higher modularity indicates better-defined communities.
- **Execution Time:** Evaluates the algorithm's scalability for large graphs.
- **Feature Utility:** Assesses the quality of extracted graph-based features when used in downstream tasks, such as classification.

### Planetoid for Semi-Supervised Learning on Graphs

For **Planetoid**, the evaluation focuses on node classification performance:

- **Classification Metrics:** Accuracy, F1-score, precision, and recall on labeled nodes.
- **Graph Regularization Effectiveness:** Measures the impact of the graph structure on performance.
- **Training Time:** Assesses the algorithm's scalability, particularly for large, sparse graphs.

### Spectral Clustering Algorithm for Dimensionality Reduction and Clustering

For **Spectral Clustering**, metrics for clustering performance include:

- **Silhouette Score:** Evaluates cluster compactness and separation.
- **Modularity:** Measures the quality of partitions in community detection tasks.
- **Computational Efficiency:** Assesses the time complexity of spectral decomposition, which is $O(|V|^3)$ for large graphs.

### Summary of Evaluation Metrics

The selection of evaluation metrics for graph algorithms depends on the specific task and algorithm being assessed. Traversal algorithms prioritize completeness and efficiency, while clustering and community detection algorithms emphasize modularity and partition quality. Supervised learning approaches, such as decision trees and Planetoid, rely on standard classification metrics, while probabilistic models like Bayesian Networks focus on inference accuracy and log-likelihood. These metrics provide comprehensive insights into the effectiveness and efficiency of graph algorithms across various applications.

# 13 Recent Advances in Graph Algorithms

Graph algorithms have witnessed significant advancements, driven by the increasing complexity and diversity of real-world problems that can be modeled as graphs. Traditional approaches such as Breadth-First Search (BFS), Depth-First Search (DFS), and Minimum Spanning Trees (MST) have established solid foundations. However, recent innovations have introduced advanced methods that leverage machine learning, probabilistic reasoning, and spectral properties to address challenges in graph-based tasks. This section explores modern developments in graph algorithms, emphasizing graph embeddings, community detection, semi-supervised learning, and scalable clustering techniques. These advancements enable effective solutions for problems in social networks, biological systems, and recommendation engines.

## 13.1 Graph Embeddings and Spectral Methods

Graph embedding techniques, such as those employed in spectral clustering and node representation learning, transform high-dimensional graph structures into lower-dimensional vector spaces. These embeddings preserve the structural and relational properties of the graph, enabling efficient downstream tasks like clustering and classification.

### Spectral Clustering

Spectral clustering algorithms utilize the eigenvalues and eigenvectors of the graph Laplacian to partition nodes into clusters. This approach is particularly effective for identifying non-convex clusters and leveraging the global structure of the graph. By embedding the nodes into a lower-dimensional space, spectral clustering enables the application of traditional clustering methods, such as $k$-means, to uncover community structures in graphs.

### Applications

Spectral methods have been applied extensively in:

- **Social Networks:** Detecting communities or groups within social graphs.
- **Biological Systems:** Identifying functional modules in protein-protein interaction networks.
- **Recommender Systems:** Grouping similar users or items to enhance recommendation accuracy.

### Challenges

Despite their utility, spectral methods face challenges related to computational complexity, particularly for large graphs, where eigenvalue decomposition can become infeasible. Recent advancements in scalable approximations have addressed these limitations, enabling the application of spectral clustering to massive datasets.

## 13.2 Community Detection and Feature Engineering

Community detection algorithms, such as the Louvain method, identify densely connected subgraphs, providing insights into the underlying structure of complex networks. These methods are

crucial for feature engineering in machine learning tasks, where the extracted communities serve as meaningful features.

### Louvain Algorithm

The Louvain algorithm employs modularity optimization to detect communities in large-scale graphs. Its greedy optimization approach ensures scalability, making it suitable for real-world applications in social network analysis and biological systems.

### Feature Engineering

Community detection enables the transformation of graph data into meaningful features, which can be incorporated into machine learning models for tasks like classification and anomaly detection.

## 13.3 Semi-Supervised Learning on Graphs

Semi-supervised learning techniques, such as the Planetoid algorithm, combine graph structure with node features to improve classification performance. These methods leverage both labeled and unlabeled data, addressing the challenge of limited labeled samples in graph-based tasks.

### Planetoid Algorithm

Planetoid integrates graph regularization and feature learning by minimizing a loss function that combines supervised and unsupervised components. This approach has proven effective in applications like citation networks and social graph analysis.

### Applications

Planetoid and related methods are widely used in:

- **Text Classification:** Classifying documents in citation networks.
- **Biological Networks:** Predicting gene or protein functions based on graph connectivity.
- **Recommendation Systems:** Leveraging graph structure to improve user-item recommendations.

## 13.4 Probabilistic Graph Models

Bayesian Networks represent a probabilistic approach to modeling relationships and dependencies within graphs. These directed acyclic graphs (DAGs) enable robust reasoning under uncertainty, making them indispensable in domains such as medical diagnosis and risk assessment.

### Bayesian Inference

Bayesian Networks utilize conditional probability distributions (CPDs) to quantify the relationships between nodes. Probabilistic inference techniques allow for predictions and causal reasoning in complex systems.

### Applications

Bayesian Networks are widely used in:

- **Medical Diagnosis:** Inferring disease probabilities based on symptoms.
- **Risk Assessment:** Modeling dependencies in financial systems or cybersecurity.
- **Causal Inference:** Understanding cause-effect relationships in experimental data.

## 13.5 Scalability and Advanced Clustering Techniques

Recent advances in graph clustering and traversal algorithms emphasize scalability to handle massive graphs. These techniques include parallelized implementations of BFS and DFS, as well as approximate clustering methods for large-scale networks.

### Challenges and Future Directions

While these advancements address many real-world challenges, ongoing research focuses on:

- Developing more interpretable and explainable graph algorithms.
- Enhancing scalability for distributed and dynamic graphs.
- Integrating graph-based learning with deep neural networks for end-to-end optimization.

## 13.6 Summary of Recent Advances in Graph Algorithms

Recent advances in graph algorithms have revolutionized the field, enabling efficient solutions for clustering, classification, and community detection in complex networks. By integrating graph structure, node features, and probabilistic reasoning, modern techniques provide robust frameworks for tackling real-world challenges. Applications span diverse domains, from social network analysis and medical diagnosis to recommendation systems and anomaly detection. These developments underscore the importance of graph algorithms as a cornerstone of modern data science and machine learning.

# 14 Graph Algorithms using Python Packages

Graph algorithms are fundamental to analyzing and understanding graph-structured data, which is ubiquitous in various domains such as social networks, biological systems, and machine learning applications. Python provides several packages tailored to specific graph algorithms and their use cases. This section outlines packages suitable for key graph algorithms, including traversal, clustering, community detection, and graph-based machine learning.

## 14.1 Packages for Graph Algorithms

The following Python packages are categorized based on their suitability for different types of graph algorithms:

#### General-Purpose Graph Analysis

- **NetworkX:** A versatile and user-friendly library for creating, manipulating, and analyzing graphs. It supports a wide range of algorithms for traversal (e.g., Breadth First Search, Depth First Search), minimum spanning trees, and basic graph statistics.
- **igraph:** Designed for high-performance graph computations, igraph is efficient for large-scale graph operations such as clustering, community detection, and traversal algorithms.

#### Machine Learning on Graphs

- **PyTorch Geometric (PyG):** Built on PyTorch, PyG specializes in graph neural networks, enabling advanced machine learning tasks such as node classification, link prediction, and graph classification. It is ideal for algorithms like the Planetoid algorithm for semi-supervised learning on graphs.
- **DGL (Deep Graph Library):** A scalable library supporting multiple backends (e.g., PyTorch, TensorFlow) for building graph neural network models. It is particularly effective for integrating node features with graph structure.

#### Large-Scale Graph Processing

- **Graph-tool:** A performance-optimized library for computationally intensive graph operations, such as spectral clustering and minimum spanning tree calculations, leveraging C++ for speed.
- **SNAP (Stanford Network Analysis Project):** Designed for processing massive graphs, SNAP provides algorithms for clustering, centrality, and temporal graphs.
- **CuGraph:** Part of NVIDIA's RAPIDS ecosystem, CuGraph utilizes GPU acceleration to perform graph algorithms like Louvain community detection and shortest path computation at scale.

#### Visualization of Graphs

- **PyGraphviz:** A Python interface to Graphviz, it provides tools for creating and rendering graph visualizations in various formats.

- **NetworkX:** Alongside its analysis capabilities, NetworkX also supports basic graph visualization for smaller graphs.

#### Kernel Methods for Graphs

- **Grakel:** A specialized library for graph kernels, enabling kernel-based machine learning approaches for graph classification and similarity measures.

### 14.2 Application of Packages to Specific Algorithms

The following packages are best suited for specific graph algorithms:

- **Breadth First Search (BFS) and Depth First Search (DFS):** - NetworkX and igraph are ideal for implementing traversal algorithms, offering efficient methods for BFS and DFS.
- **Minimum Spanning Trees:** - NetworkX, igraph, and Graph-tool provide built-in functions for computing minimum spanning trees using Kruskal's or Prim's algorithms.
- **Decision Trees:** - While decision trees are not graph algorithms in the strict sense, packages like scikit-learn can integrate with graph-based features extracted using other libraries (e.g., PyG or DGL).
- **Bayesian Networks:** - Libraries such as pgmpy (Probabilistic Graphical Models in Python) and NetworkX support the representation and inference of Bayesian networks.
- **Louvain Algorithm for Community Detection:** - CuGraph, igraph, and Graph-tool provide optimized implementations of the Louvain algorithm for community detection.
- **Planetoid for Semi-Supervised Learning:** - PyTorch Geometric (PyG) and DGL are best suited for implementing the Planetoid algorithm, leveraging their support for graph neural networks and semi-supervised learning.
- **Spectral Clustering for Dimensionality Reduction and Clustering:** - Graph-tool, igraph, and NetworkX offer tools for constructing graph Laplacians and performing spectral decomposition.

### 14.3 Advantages and Limitations of Python Graph Libraries

#### Advantages:

- **Comprehensive Ecosystem:** Python's graph libraries cover a wide range of algorithms, from basic traversal to advanced graph machine learning.
- **Ease of Use:** Libraries like NetworkX offer intuitive APIs, making them accessible to both beginners and experts.
- **Scalability:** High-performance libraries such as CuGraph and SNAP can handle massive graphs efficiently.

#### Limitations:

- **Performance Trade-offs:** Libraries like NetworkX may struggle with large-scale graphs due to their Python-based implementation.

- **Complexity of Advanced Methods:** Implementing machine learning algorithms (e.g., Planetoid) requires familiarity with frameworks like PyTorch or DGL.

### 14.4 Conclusion

Python's extensive ecosystem of graph algorithm packages enables researchers and practitioners to address diverse challenges in graph theory, machine learning, and network science. By selecting the appropriate package for a given algorithm or task, users can effectively leverage the power of graph algorithms to extract insights from structured data.

# 15 Summary

Graph algorithms have evolved into versatile tools for solving complex problems across various domains. This document has covered a wide range of algorithms, each tailored to specific graph-related tasks. BFS and DFS form the foundation of graph traversal techniques, enabling the exploration of connected components, cycle detection, and shortest path discovery in unweighted graphs. Minimum Spanning Tree algorithms, such as Kruskal's and Prim's, provide efficient solutions for optimization problems in network design and clustering.

Advanced methods like Decision Trees and Bayesian Networks extend graph algorithms into decision-making and probabilistic inference, supporting applications in machine learning, risk assessment, and causal reasoning. The Louvain Algorithm and Spectral Clustering enable community detection and dimensionality reduction, enhancing graph-based feature engineering for machine learning models. Additionally, Planetoid exemplifies the integration of graph algorithms with deep learning, addressing semi-supervised learning challenges in large-scale networks.

The module concludes with an overview of recent advances, including graph embeddings, scalability enhancements, and innovations in machine learning on graphs. These developments demonstrate the increasing importance of graph algorithms in addressing real-world challenges in data science, artificial intelligence, and beyond.

# 16 Module Questions

### Breadth First Search Traversal Algorithms

- **Question:** What is the purpose of the Breadth-First Search (BFS) algorithm in graph traversal?
- **Question:** How does BFS ensure all vertices are visited in the correct order?
- **Question:** What is the time complexity of BFS?

### Depth First Search Traversal Algorithms

- **Question:** How does Depth-First Search (DFS) traverse a graph?
- **Question:** What are common applications of DFS?
- **Question:** What is the time complexity of DFS?

### Minimum Spanning Trees

- **Question:** What is a Minimum Spanning Tree (MST)?
- **Question:** Which algorithms are commonly used to compute an MST?
- **Question:** What is the time complexity of Kruskal's algorithm?

### Decision Trees

- **Question:** What is the structure of a decision tree?
- **Question:** How does a decision tree make predictions?
- **Question:** What are the advantages of decision trees?

### Bayesian Networks

- **Question:** What is a Bayesian Network?
- **Question:** What is the goal of inference in a Bayesian Network?
- **Question:** What are common applications of Bayesian Networks?

### Louvain Algorithm for Graph-Based Feature Engineering

- **Question:** What is the Louvain algorithm used for?
- **Question:** How does the Louvain algorithm optimize modularity?
- **Question:** What are some applications of the Louvain algorithm?

**Planetoid for Semi-Supervised Learning on Graphs**

- **Question:** What is the purpose of Planetoid?
- **Question:** How does Planetoid utilize graph structure?
- **Question:** What types of graphs can Planetoid handle?

**Spectral Clustering Algorithm for Dimensionality Reduction and Clustering**

- **Question:** What is the principle behind spectral clustering?
- **Question:** How is the graph Laplacian constructed?
- **Question:** What are typical applications of spectral clustering?

# 17 Recommended Kaggle Datasets for Graph Algorithms

Below are Kaggle datasets associated with each graph algorithm, along with a brief explanation of their use.

## 17.1 Breadth First Search Traversal Algorithms

**Dataset:** Facebook Page-Page Network (`https://www.kaggle.com/rohanrao/facebook-page-page-netwo`
**Use:** This dataset represents a graph of Facebook pages connected based on mutual interests. BFS can be used to explore connected components and analyze shortest paths in this network.

## 17.2 Depth First Search Traversal Algorithms

**Dataset:** Github Social Network (`https://www.kaggle.com/snap/github`)
**Use:** This dataset represents developer connections on GitHub. DFS can traverse the graph to explore clusters of developers or detect cycles.

## 17.3 Minimum Spanning Trees

**Dataset:** Road Networks of the USA (`https://www.kaggle.com/jehanbhathena/usa-roads-dataset`)
**Use:** This dataset contains the road network of the United States. Algorithms like Kruskal's or Prim's can compute MSTs to design cost-efficient road networks.

## 17.4 Decision Trees

**Dataset:** Heart Disease UCI (`https://www.kaggle.com/ronitf/heart-disease-uci`)
**Use:** This dataset predicts heart disease based on patient attributes. Decision trees can be applied to build interpretable models identifying key features contributing to heart disease.

## 17.5 Bayesian Networks

**Dataset:** Titanic - Machine Learning from Disaster (`https://www.kaggle.com/c/titanic`)
**Use:** This dataset predicts survival on the Titanic. Bayesian Networks can model the relationships between demographic features and survival probabilities.

## 17.6 Louvain Algorithm for Graph-Based Feature Engineering

**Dataset:** Yelp Open Dataset (`https://www.kaggle.com/yelp-dataset/yelp-dataset`)
**Use:** This dataset includes a graph of businesses, users, and reviews. The Louvain algorithm can detect communities of similar businesses based on user reviews, aiding in feature engineering.

## 17.7 Planetoid for Semi-Supervised Learning on Graphs

**Dataset:** Cora Citation Network (`https://www.kaggle.com/c/planetoid-dataset`)
**Use:** This dataset contains a citation network of research papers. Planetoid can classify papers into topics using both node features and graph structure.

### 17.8 Spectral Clustering Algorithm for Dimensionality Reduction and Clustering

**Dataset:** Mall Customer Segmentation Data (`https://www.kaggle.com/vjchoudhary7/customer-segment`
**Use:** This dataset includes customer data for segmentation. Spectral clustering can cluster customers based on purchasing behaviors, revealing patterns in consumer groups.

## 18 Module Questions and Answers

### Breadth First Search Traversal Algorithms

- **Question:** What is the purpose of the Breadth-First Search (BFS) algorithm in graph traversal?
- **Answer:** The BFS algorithm explores vertices level-by-level starting from a source vertex. It is used to find the shortest path in unweighted graphs, detect connected components, and analyze graph structures.
- **Question:** How does BFS ensure all vertices are visited in the correct order?
- **Answer:** BFS uses a queue to maintain vertices to be explored and a visited array to prevent revisiting. It processes all vertices at distance $k$ before moving to vertices at distance $k+1$.
- **Question:** What is the time complexity of BFS?
- **Answer:** The time complexity is $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph.

### Depth First Search Traversal Algorithms

- **Question:** How does Depth-First Search (DFS) traverse a graph?
- **Answer:** DFS explores as deeply as possible along each branch before backtracking. It uses a stack (either explicit or via recursion) to manage traversal.
- **Question:** What are common applications of DFS?
- **Answer:** DFS is used for cycle detection, topological sorting, connected component identification, and solving maze problems.
- **Question:** What is the time complexity of DFS?
- **Answer:** The time complexity is $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

### Minimum Spanning Trees

- **Question:** What is a Minimum Spanning Tree (MST)?
- **Answer:** An MST is a subset of edges in a connected, weighted graph that connects all vertices with the minimum total edge weight and without forming cycles.
- **Question:** Which algorithms are commonly used to compute an MST?
- **Answer:** Kruskal's and Prim's algorithms are commonly used for computing MSTs. Kruskal's algorithm sorts edges by weight, while Prim's algorithm grows the MST by adding the lowest-weight edge connecting the tree to a new vertex.
- **Question:** What is the time complexity of Kruskal's algorithm?
- **Answer:** The time complexity is $O(|E| \log |E|)$, where $|E|$ is the number of edges.

### Decision Trees

- **Question:** What is the structure of a decision tree?
- **Answer:** A decision tree is a hierarchical model represented as a directed graph where internal nodes are decision points based on features, and leaf nodes represent outcomes or class labels.
- **Question:** How does a decision tree make predictions?
- **Answer:** The input features are evaluated at each decision node, and the traversal proceeds to the left or right child based on a threshold until a leaf node is reached.
- **Question:** What are the advantages of decision trees?
- **Answer:** Decision trees are interpretable, handle both categorical and numerical data, and make no assumptions about the data distribution.

### Bayesian Networks

- **Question:** What is a Bayesian Network?
- **Answer:** A Bayesian Network is a directed acyclic graph where nodes represent random variables, and edges represent conditional dependencies. Each node has an associated conditional probability distribution (CPD).
- **Question:** What is the goal of inference in a Bayesian Network?
- **Answer:** The goal is to compute the posterior probability of a query variable given evidence by marginalizing out hidden variables and normalizing the result.
- **Question:** What are common applications of Bayesian Networks?
- **Answer:** Applications include medical diagnosis, risk assessment, causal inference, and decision-making under uncertainty.

### Louvain Algorithm for Graph-Based Feature Engineering

- **Question:** What is the Louvain algorithm used for?
- **Answer:** The Louvain algorithm is used for community detection in large graphs by maximizing modularity, a measure of the quality of partitioning.
- **Question:** How does the Louvain algorithm optimize modularity?
- **Answer:** It iteratively assigns nodes to communities to maximize modularity locally, then aggregates communities into a new graph and repeats the process.
- **Question:** What are some applications of the Louvain algorithm?
- **Answer:** Applications include social network analysis, feature engineering, and clustering tasks in large datasets.

### Planetoid for Semi-Supervised Learning on Graphs

- **Question:** What is the purpose of Planetoid?

- **Answer:** Planetoid is designed for semi-supervised learning on graphs, leveraging node features and graph structure to classify nodes with limited labels.
- **Question:** How does Planetoid utilize graph structure?
- **Answer:** Planetoid uses a loss function that incorporates neighborhood relationships, enabling the model to propagate label information through the graph.
- **Question:** What types of graphs can Planetoid handle?
- **Answer:** Planetoid supports both transductive settings (fixed graph) and inductive settings (generalizing to new graphs).

**Spectral Clustering Algorithm for Dimensionality Reduction and Clustering**

- **Question:** What is the principle behind spectral clustering?
- **Answer:** Spectral clustering uses the eigenvalues of the graph Laplacian matrix to reduce dimensionality and identify clusters.
- **Question:** How is the graph Laplacian constructed?
- **Answer:** The graph Laplacian is derived from the adjacency matrix and degree matrix as $L = D - A$, where $A$ is the adjacency matrix and $D$ is the degree matrix.
- **Question:** What are typical applications of spectral clustering?
- **Answer:** Applications include image segmentation, social network analysis, and clustering high-dimensional datasets.

# References


[1] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (2008), P10008.

[2] John Adrian Bondy and Uday S R Murty. *Graph Theory*. Springer, 2008.

[3] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009.

[4] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th. MIT Press, 2022.

[5] Santo Fortunato. "Community detection in graphs". In: *Physics Reports* 486.3-5 (2010), pp. 75–174.

[6] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd. Springer, 2009. DOI: 10.1007/978-0-387-84858-7.

[7] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT Press, 2009.

[8] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[9] Andrew Y Ng, Michael I Jordan, and Yair Weiss. "Spectral clustering: Analysis and an algorithm". In: *Advances in Neural Information Processing Systems* 14 (2002), pp. 849–856.

[10] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[11] Douglas B West. *Introduction to Graph Theory*. Prentice Hall, 2001.

[12] Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. "Revisiting semi-supervised learning with graph embeddings". In: *International Conference on Machine Learning (ICML)*. 2016, pp. 40–48.