ECE
ÉCOLE D'INGÉNIEURS
ENGINEERING SCHOOL

GRAPH THEORY

AUTHORS:

-Anir Agountaf

-Muhammad Hassaan Shafique

PROFESSOR:

M. Soufian Benamor

COURSE:

Mathematics for Data Science

PROGRAM:

Master of Science in Data Management

Academic Year 2025-2026

Project Report

7 December 2025

Abstract

Graph theory is a foundational area of discrete mathematics concerned with the study of networks formed by nodes (vertices) and the connections (edges) between them. This report provides an in-depth overview of core graph-theoretic concepts, including graph representations, connectivity, traversals, shortest-path algorithms, tree and cycle structures and properties of directed and weighted graphs. Beyond these fundamentals, the report emphasizes the growing relevance of graph theory in modern computational fields, particularly artificial intelligence (AI).

In AI, graphs serve as powerful modeling tools for capturing complex relationships within data. Knowledge graphs represent structured information that supports reasoning and semantic understanding in intelligent systems. Graph algorithms enable efficient navigation and inference over these structures, contributing to advancements in natural language processing, recommendation systems and search technologies. Additionally, the emergence of graph-based machine learning methods—such as Graph Neural Networks (GNNs)—has expanded the role of graph theory in pattern recognition, social network analysis, molecular prediction and other areas where relational dependencies are critical.

By examining both classical theory and contemporary AI applications, this project highlights how graph theory provides the mathematical framework necessary for representing, analyzing and learning from interconnected data. The report concludes that as AI systems increasingly rely on relational information, the importance of graph theory will continue to grow, bridging the gap between mathematical abstraction and real-world intelligent technologies.

Keywords: Graph Theory, Data Science, Artificial Intelligence, Graph Neural Networks, Network Analysis, Machine Learning, Algorithms

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

In the contemporary landscape of data-driven decision making, traditional tabular data structures often prove inadequate for capturing the complex, interconnected relationships that characterize real-world systems. From the intricate web of social interactions on digital platforms to the sophisticated networks of protein interactions in biological organisms, modern datasets are inherently relational. The proliferation of interconnected data sources—social networks, communication systems, biological pathways, transportation networks, and knowledge graphs—has created an urgent need for mathematical frameworks capable of modeling, analyzing, and extracting insights from such structured data.

Graph theory, with its elegant mathematical foundation, provides precisely such a framework. Originally developed to solve abstract mathematical problems, graph theory has evolved into an indispensable tool across numerous disciplines. In computer science, it underpins algorithms for network routing and database design; in sociology, it models social structures and influence propagation; in biology, it represents metabolic pathways and genetic interactions. However, it is in the realms of artificial intelligence and data science that graph theory has witnessed its most transformative applications in recent years.

The emergence of Graph Neural Networks (GNNs) and other graph-based machine learning techniques represents a paradigm shift in how we approach learning from relational data. These advancements have enabled breakthroughs in diverse applications: from predicting molecular properties in drug discovery to detecting anomalous patterns in financial transactions, from recommending products in e-commerce to understanding information diffusion in social media. This convergence of mathematical theory and practical application forms the central theme of this report.

This report aims to bridge the theoretical foundations of graph theory with their practical implementations in data science. By systematically exploring both classical algorithms and modern machine learning approaches, we seek to demonstrate how graph-theoretical concepts not only provide mathematical rigor but also offer powerful tools for solving real-world problems in an increasingly interconnected digital ecosystem.

1.2 What Is Graph Theory?

Graph theory is a branch of discrete mathematics that studies the properties and applications of graphs—mathematical structures consisting of vertices (also called nodes) connected by edges (also called links or arcs). Formally, a graph G is defined as an ordered pair $G = (V, E)$ where:

- V is a finite set of vertices (nodes)
- $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ is a set of edges connecting pairs of vertices

This deceptively simple abstraction possesses remarkable expressive power. The history of graph theory traces back to 1736 when Leonhard Euler solved the Königsberg bridge problem, proving that there was no walk through the city that would cross each of its seven bridges exactly once. Euler's solution laid the foundation for what would become a rich mathematical discipline, though the term "graph" wasn't coined until 1878 by James Joseph Sylvester.

1.2.1 Historical Development

The evolution of graph theory can be traced through several key milestones:

- **1736:** Euler's solution to the Königsberg bridge problem
- **1847:** Kirchhoff's circuit laws for electrical networks
- **1852:** Francis Guthrie poses the Four Color Problem
- **1926:** Dénes Kőnig publishes the first graph theory textbook
- **1959:** Erdős and Rényi introduce random graph theory
- **1976:** Appel and Haken prove the Four Color Theorem
- **1998-1999:** Watts-Strogatz and Barabási-Albert introduce small-world and scale-free network models

1.2.2 Fundamental Concepts

At its core, graph theory provides a language for describing relationships. The vertices represent entities, while edges represent relationships between those entities. This simple representation enables the modeling of remarkably complex systems:

- **Social Networks:** Vertices represent individuals; edges represent friendships, following relationships, or communications

- **Transportation Systems:** Vertices represent locations (stations, airports); edges represent routes or connections
- **Biological Networks:** Vertices represent proteins or genes; edges represent interactions or regulatory relationships
- **Knowledge Graphs:** Vertices represent entities (people, places, concepts); edges represent semantic relationships
- **Computer Networks:** Vertices represent devices; edges represent physical or logical connections

The power of this abstraction lies in its ability to separate the essential relational structure from domain-specific details, enabling the transfer of insights and algorithms across diverse application domains.

1.3 Importance in AI and Data Science

Graph theory's significance in artificial intelligence and data science has grown exponentially in recent years, driven by several converging trends: the increasing availability of network-structured data, advances in computational power, and breakthroughs in machine learning methodologies.

1.3.1 Natural Representation of Relational Data

Many real-world problems involve entities and their relationships, which graphs naturally capture. This representation enables:

- **Modeling Complex Systems:** Graphs can represent systems with multiple types of entities and relationships through heterogeneous graphs and multi-relational graphs
- **Analysis of Network Properties:** Centrality measures, community structure, and connectivity properties provide insights into system behavior
- **Algorithm Development:** Graph algorithms leverage relational information to solve problems more efficiently than approaches that ignore structure

1.3.2 Foundation for Graph-Based Machine Learning

Recent advances in machine learning have increasingly incorporated graph structures:

- **Graph Neural Networks (GNNs):** Extend neural network architectures to operate directly on graph-structured data, enabling learning from relational information

- **Graph Embeddings:** Techniques like Node2Vec and DeepWalk learn low-dimensional vector representations of nodes or entire graphs, preserving structural properties
- **Spectral Methods:** Utilize the spectral properties of graph Laplacians for clustering, classification, and dimensionality reduction
- **Graph Kernels:** Define similarity measures between graphs for classification and regression tasks

1.3.3 Scalability and Efficiency

Graph algorithms often provide efficient solutions to complex problems:

- **Shortest Path Algorithms:** Dijkstra's and Floyd-Warshall algorithms find optimal routes in polynomial time
- **Community Detection:** Modularity optimization and spectral clustering identify cohesive subgroups in large networks
- **Distributed Processing:** Frameworks like Pregel and GraphX enable processing of billion-scale graphs on distributed systems

1.3.4 Interpretability and Explainability

Unlike some "black-box" machine learning models, graph-based approaches often provide more interpretable results:

- **Structural Explanations:** Important nodes, edges, or subgraphs can be identified and visualized
- **Causal Inference:** Graph structures naturally support causal reasoning and inference
- **Knowledge Integration:** Domain knowledge can be incorporated through predefined graph structures

1.3.5 Emerging Applications

The application of graph theory in AI and data science continues to expand into new domains:

- **Autonomous Systems:** Road networks for self-driving vehicles, sensor networks for IoT systems
- **Healthcare:** Disease progression networks, drug interaction graphs, patient similarity networks

- **Cybersecurity:** Attack graphs, malware propagation networks, anomaly detection in network traffic
- **Finance:** Transaction networks for fraud detection, correlation networks for portfolio optimization

Chapter 2

Fundamentals of Graph Theory

2.1 Basic Definitions

2.1.1 Graphs, Vertices, Edges

Definition 2.1 (Graph)**.** A **graph** G is an ordered pair (V, E) where:

- V is a set of **vertices** (or nodes)
- $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ is a set of **edges** (or links)

For a graph with n vertices and m edges, we denote $|V| = n$ and $|E| = m$.

Definition 2.2 (Directed Graph)**.** A **directed graph** (or digraph) $D = (V, A)$ consists of:

- V: set of vertices
- $A \subseteq \{(u, v) \mid u, v \in V, u \neq v\}$: set of **arcs** (directed edges)

Each arc (u, v) has a direction from u (tail) to v (head).

2.1.2 Degree, Paths, Cycles

Definition 2.3 (Degree)**.** The **degree** of a vertex v, denoted $\deg(v)$, is the number of edges incident to v. For directed graphs:

- $\text{indeg}(v)$: number of arcs entering v
- $\text{outdeg}(v)$: number of arcs leaving v

Definition 2.4 (Path)**.** A **path** is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, k-1$. The **length** of the path is k (number of edges).

Definition 2.5 (Cycle)**.** A **cycle** is a path $v_0, v_1, \ldots, v_k$ where $v_0 = v_k$ and all other vertices are distinct.

2.1.3 Connectedness

Definition 2.6 (Connected Graph)**.** An undirected graph is **connected** if there exists a path between every pair of vertices.

Definition 2.7 (Connected Components)**.** A **connected component** of a graph is a maximal connected subgraph.

2.2 Types of Graphs

2.2.1 Simple Graphs

Definition 2.8 (Simple Graph)**.** A **simple graph** is an undirected graph without loops (edges connecting a vertex to itself) and without multiple edges between the same pair of vertices.

2.2.2 Multigraphs

Definition 2.9 (Multigraph)**.** A **multigraph** allows multiple edges between the same pair of vertices.

2.2.3 Bipartite Graphs

Definition 2.10 (Bipartite Graph)**.** A graph $G = (V, E)$ is **bipartite** if V can be partitioned into two disjoint sets U and W such that every edge connects a vertex in U to a vertex in W.

Theorem 2.1 (Characterization of Bipartite Graphs)**.** A graph is bipartite if and only if it contains no odd cycles.

2.2.4 Complete Graphs

Definition 2.11 (Complete Graph)**.** A **complete graph** K_n is a simple graph with n vertices where every pair of distinct vertices is connected by an edge.

2.2.5 Trees and Forests

Definition 2.12 (Tree)**.** A **tree** is a connected graph with no cycles.

Theorem 2.2 (Tree Properties)**.** For a tree $T = (V, E)$ with n vertices:

1. $|E| = n - 1$
2. There is exactly one path between any two vertices

3. Adding any edge creates exactly one cycle
4. Removing any edge disconnects the graph

2.2.6 Planar Graphs

Definition 2.13 (Planar Graph)**.** A graph is **planar** if it can be drawn in the plane without edge crossings.

Theorem 2.3 (Euler's Formula)**.** For a connected planar graph with n vertices, m edges, and f faces:

$$n - m + f = 2$$

2.3 Graph Representations

2.3.1 Adjacency Matrix

The **adjacency matrix** A of a graph $G = (V, E)$ with n vertices is an $n \times n$ matrix where:

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

For weighted graphs, $A_{ij} = w(i,j)$ where $w(i,j)$ is the weight of edge (i,j).

Example 2.1. For the graph with vertices $\{1,2,3\}$ and edges $\{(1,2),(2,3),(3,1)\}$:

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

2.3.2 Adjacency List

An **adjacency list** stores for each vertex a list of its neighbors:

- Space complexity: $O(n + m)$
- Efficient for sparse graphs
- Easy to iterate over neighbors

2.3.3 Edge List

An **edge list** simply stores all edges as pairs of vertices:

- Space complexity: $O(m)$
- Simple implementation
- Inefficient for neighbor queries

2.3.4 Incidence Matrix

The **incidence matrix** B of a graph $G = (V, E)$ is an $n \times m$ matrix where:

$$B_{ij} = \begin{cases} 1 & \text{if vertex } i \text{ is incident to edge } j \\ 0 & \text{otherwise} \end{cases}$$

2.4 Weighted Graphs

Definition 2.14 (Weighted Graph)**.** A **weighted graph** $G = (V, E, w)$ assigns a weight $w(e) \in \mathbb{R}$ to each edge $e \in E$.

Applications include:

- Transportation networks (distances/travel times)
- Communication networks (bandwidth/cost)
- Social networks (strength of relationships)

2.5 Connectivity and Components

2.5.1 Vertex Connectivity

Definition 2.15 (k-Connected)**.** A graph is **k-connected** if removing fewer than k vertices does not disconnect it.

2.5.2 Edge Connectivity

Definition 2.16 (k-Edge-Connected)**.** A graph is **k-edge-connected** if removing fewer than k edges does not disconnect it.

2.6 Special Graph Families

2.6.1 Regular Graphs

Definition 2.17 (Regular Graph)**.** A graph is **k-regular** if every vertex has degree k.

2.6.2 Cayley Graphs

Definition 2.18 (Cayley Graph)**.** Given a group G and a generating set S, the **Cayley graph** has vertices corresponding to group elements and edges (g, gs) for $g \in G, s \in S$.

2.7 Planar Graphs

2.7.1 Kuratowski's Theorem

Theorem 2.4 (Kuratowski, 1930)**.** A graph is planar if and only if it does not contain a subdivision of K_5 or $K_{3,3}$.

2.7.2 Four Color Theorem

Theorem 2.5 (Four Color Theorem)**.** Every planar graph can be colored with at most 4 colors such that no adjacent vertices share the same color.

2.8 Cliques

Definition 2.19 (Clique)**.** A **clique** is a subset of vertices such that every two distinct vertices are adjacent.

Definition 2.20 (Clique Number)**.** The **clique number** $\omega(G)$ is the size of the largest clique in G.

2.9 Independent Sets

Definition 2.21 (Independent Set)**.** An **independent set** is a subset of vertices such that no two vertices are adjacent.

Definition 2.22 (Independence Number)**.** The **independence number** $\alpha(G)$ is the size of the largest independent set in G.

Theorem 2.6 (Relationship)**.** For any graph G with n vertices:

$$\omega(G) \cdot \alpha(G) \geq n$$

Chapter 3

Classical Graph Algorithms

3.1 Traversal Algorithms

Graph traversal algorithms form the foundation for many other graph algorithms, enabling systematic exploration of graph structures.

3.1.1 Depth-First Search (DFS)

Depth-First Search explores as far as possible along each branch before backtracking.

Algorithm 1 Depth-First Search (Recursive)

```
0: function DFS(G, v)
0:     Mark v as visited
0:     for each neighbor u of v in G do
0:         if u is not visited then
0:             DFS(G, u)
0:         end if
0:     end for
0: end function=0
```

Properties:

- Time Complexity: $O(|V| + |E|)$
- Space Complexity: $O(|V|)$ for recursion stack
- Applications: Topological sorting, connected components, cycle detection

3.1.2 Breadth-First Search (BFS)

Breadth-First Search explores all vertices at the present depth level before moving to vertices at the next depth level.

Algorithm 2 Breadth-First Search

```
0: function BFS(G, s)
0:     Create queue Q
0:     Mark s as visited
0:     Q.enqueue(s)
0:     while Q is not empty do
0:         v ← Q.dequeue()
0:         for each neighbor u of v in G do
0:             if u is not visited then
0:                 Mark u as visited
0:                 Q.enqueue(u)
0:             end if
0:         end for
0:     end while
0: end function=0
```

Properties:

- Time Complexity: $O(|V| + |E|)$
- Space Complexity: $O(|V|)$ for queue
- Applications: Shortest path in unweighted graphs, peer-to-peer networks

3.2 Shortest Path Algorithms

Shortest path algorithms find the minimum-weight path between vertices in a graph.

3.2.1 Dijkstra's Algorithm

Dijkstra's algorithm finds shortest paths from a source vertex in graphs with non-negative edge weights.

Algorithm 3 Dijkstra's Algorithm

```
0: function DIJKSTRA(G, s)
0:     Initialize distances d[s] ← 0, d[v] ← ∞ for v ≠ s
0:     Initialize priority queue Q with all vertices
0:     while Q is not empty do
0:         u ← Q.extract_min()
0:         for each neighbor v of u do
0:             alt ← d[u] + w(u, v)
0:             if alt < d[v] then
0:                 d[v] ← alt
0:                 prev[v] ← u
0:                 Q.decrease_key(v, alt)
0:             end if
0:         end for
0:     end while
0:     return d, prev
0: end function=0
```

Complexity: $O((|V| + |E|) \log |V|)$ with binary heap

3.2.2 Bellman-Ford Algorithm

Bellman-Ford handles graphs with negative edge weights and detects negative-weight cycles.

Algorithm 4 Bellman-Ford Algorithm

```
0: function BellmanFord(G, s)
0:     Initialize distances d[s] ← 0, d[v] ← ∞ for v ≠ s
0:     for i ← 1 to |V| − 1 do
0:         for each edge (u, v) in E with weight w do
0:             if d[u] + w < d[v] then
0:                 d[v] ← d[u] + w
0:                 prev[v] ← u
0:             end if
0:         end for
0:     end for
0:     for each edge (u, v) in E with weight w do
0:         if d[u] + w < d[v] then
0:             return "Graph contains negative weight cycle"
0:         end if
0:     end for
0:     return d, prev
0: end function=0
```

Complexity: $O(|V||E|)$

3.2.3 Floyd-Warshall Algorithm

Floyd-Warshall computes all-pairs shortest paths in a weighted graph.

Algorithm 5 Floyd-Warshall Algorithm

```
0: function FloydWarshall(W)
0:     D ← W
0:     for k ← 1 to n do
0:         for i ← 1 to n do
0:             for j ← 1 to n do
0:                 if D[i][k] + D[k][j] < D[i][j] then
0:                     D[i][j] ← D[i][k] + D[k][j]
0:                 end if
0:             end for
0:         end for
0:     end for
0:     return D
0: end function=0
```

Complexity: $O(|V|^3)$ time, $O(|V|^2)$ space

3.2.4 A* Search Algorithm

A* is an informed search algorithm that uses heuristics to improve performance.

Algorithm 6 A* Search Algorithm

```
0: function ASTAR(G, start, goal)
0:     Initialize open set with start, g[start] ← 0, f[start] ← h(start)
0:     while open set is not empty do
0:         current ← node in open set with lowest f
0:         if current = goal then
0:             return reconstruct path
0:         end if
0:         Remove current from open set
0:         for each neighbor v of current do
0:             tentative_g ← g[current] + w(current, v)
0:             if tentative_g < g[v] then
0:                 g[v] ← tentative_g
0:                 f[v] ← g[v] + h(v)
0:                 prev[v] ← current
0:                 if v not in open set then
0:                     Add v to open set
0:                 end if
0:             end if
0:         end for
0:     end while
0:     return failure
0: end function=0
```

3.3 Minimum Spanning Tree Algorithms

MST algorithms find a subset of edges that connects all vertices with minimum total edge weight.

3.3.1 Kruskal's Algorithm

Kruskal's algorithm builds MST by sorting edges and adding them in increasing weight order.

Algorithm 7 Kruskal's Algorithm

```
0: function KRUSKAL(G)
0:     Sort edges by weight in non-decreasing order
0:     Initialize disjoint-set data structure for vertices
0:     mst ← ∅
0:     for each edge (u, v) in sorted order do
0:         if find(u) ≠ find(v) then
0:             Add (u, v) to mst
0:             union(u, v)
0:         end if
0:     end for
0:     return mst
0: end function=0
```

Complexity: $O(|E| \log |E|)$ for sorting edges

3.3.2 Prim's Algorithm

Prim's algorithm grows MST from an arbitrary starting vertex.

Algorithm 8 Prim's Algorithm

```
0: function PRIM(G, r)
0:     for each vertex v in G do
0:         key[v] ← ∞
0:         parent[v] ← null
0:     end for
0:     key[r] ← 0
0:     Initialize priority queue Q with all vertices
0:     while Q is not empty do
0:         u ← Q.extract_min()
0:         for each neighbor v of u do
0:             if v ∈ Q and w(u, v) < key[v] then
0:                 parent[v] ← u
0:                 key[v] ← w(u, v)
0:                 Q.decrease_key(v, key[v])
0:             end if
0:         end for
0:     end while
0:     return parent
0: end function=0
```

Complexity: $O(|E| \log |V|)$ with binary heap

3.4 Network Flow Algorithms

Network flow algorithms solve problems involving flow of resources through networks.

3.4.1 Max-Flow / Min-Cut Theorem

The maximum flow through a network equals the capacity of the minimum cut.

3.4.2 Ford-Fulkerson Method

Ford-Fulkerson finds maximum flow by repeatedly finding augmenting paths.

Algorithm 9 Ford-Fulkerson Method

```
0: function FordFulkerson(G, s, t)
0:     Initialize flow f to 0 on all edges
0:     while there exists augmenting path p from s to t in G_f do
0:         c_f(p) ← min{c_f(u, v) : (u, v) ∈ p}
0:         for each edge (u, v) in p do
0:             if (u, v) is in original graph then
0:                 f(u, v) ← f(u, v) + c_f(p)
0:             else
0:                 f(v, u) ← f(v, u) − c_f(p)
0:             end if
0:         end for
0:     end while
0:     return f
0: end function=0
```

3.4.3 Edmonds-Karp Algorithm

Edmonds-Karp uses BFS to find shortest augmenting paths.

Algorithm 10 Edmonds-Karp Algorithm

```
0: function EdmondsKarp(G, s, t)
0:     flow ← 0
0:     Initialize residual network G_f
0:     while true do
0:         p ← BFS(G_f, s, t)
0:         if no such path exists then
0:             break
0:         end if
0:         c_f(p) ← min{c_f(u, v) : (u, v) ∈ p}
0:         Update flow along p
0:         Update G_f
0:         flow ← flow + c_f(p)
0:     end while
0:     return flow
0: end function=0
```

Complexity: $O(|V||E|^2)$

3.5 Graph Matching Algorithms

Graph matching finds sets of edges without common vertices.

3.5.1 Maximum Matching

A maximum matching contains the largest possible number of edges.

3.5.2 Bipartite Matching

Bipartite matching finds maximum matchings in bipartite graphs.

3.5.3 Stable Matching (Gale-Shapley Algorithm)

Stable matching finds a matching where no unmatched pair would both prefer each other.

Algorithm 11 Gale-Shapley Algorithm

```
0: function GaleShapley(M, W)
0:     Initialize all men and women as free
0:     while there exists free man m with woman w to propose to do
0:         w ← first woman on m's list not yet proposed to
0:         if w is free then
0:             m and w become engaged
0:         else if w prefers m to current partner m′ then
0:             m′ becomes free
0:             m and w become engaged
0:         end if
0:     end while
0:     return matching of engaged pairs
0: end function=0
[0]
```

Properties:

- Always produces stable matching
- Man-optimal matching
- Time Complexity: $O(n^2)$ where n is number of men/women

3.6 Summary

This chapter presented foundational classical graph algorithms. From basic traversal to sophisticated optimization, these methods provide efficient solutions to practical problems.

Algorithm	Time Complexity	Space	Key Application
DFS	$O(\|V\| + \|E\|)$	$O(\|V\|)$	Topological sort
BFS	$O(\|V\| + \|E\|)$	$O(\|V\|)$	Shortest path (unweighted)
Dijkstra	$O((\|V\| + \|E\|) \log \|V\|)$	$O(\|V\|)$	Shortest path
Bellman-Ford	$O(\|V\|\|E\|)$	$O(\|V\|)$	Negative weights
Floyd-Warshall	$O(\|V\|^3)$	$O(\|V\|^2)$	All-pairs shortest
Kruskal	$O(\|E\| \log \|E\|)$	$O(\|V\|)$	MST
Prim	$O(\|E\| \log \|V\|)$	$O(\|V\|)$	MST
Ford-Fulkerson	$O(\|E\| \cdot f^*)$	$O(\|V\| + \|E\|)$	Max flow
Edmonds-Karp	$O(\|V\|\|E\|^2)$	$O(\|V\| + \|E\|)$	Max flow
Gale-Shapley	$O(n^2)$	$O(n^2)$	Stable matching

Table 3.1: Summary of classical graph algorithms. f^* is maximum flow value.

Chapter 4

Advanced Graph Theory Concepts

4.1 Graph Connectivity

4.1.1 Vertex and Edge Connectivity

4.1.2 Strong and Weak Connectivity

4.2 Graph Coloring

4.2.1 Chromatic Number

4.2.2 Applications of Coloring

4.3 Centrality Measures

4.3.1 Degree Centrality

4.3.2 Betweenness Centrality

4.3.3 Closeness Centrality

4.3.4 Eigenvector and PageRank

4.4 Spectral Graph Theory

4.4.1 Graph Laplacian

4.4.2 Eigenvalues and Eigenvectors

4.4.3 Spectral Clustering

4.5 Random Graphs

4.5.1 Erdős–Rényi Model

4.5.2 Small-World Networks

4.5.3 Scale-Free Networks

Chapter 5

Graph Theory in Machine Learning

5.1 Graph-Based Learning Paradigms

5.1.1 Transductive Learning

5.1.2 Semi-Supervised Learning

5.1.3 Graph Regularization

5.2 Feature Engineering with Graphs

5.2.1 Graph Embeddings

5.2.2 Node2Vec

5.2.3 DeepWalk

5.2.4 Graph Kernels

5.3 Graph Neural Networks (GNNs)

5.3.1 Message Passing Neural Networks

5.3.2 Graph Convolutional Networks (GCN)

5.3.3 Graph Attention Networks (GAT)

5.3.4 GraphSAGE

5.3.5 Training Challenges and Limitations

5.4 Explainability in GNNs

5.4.1 Subgraph Explanations

5.4.2 Feature Attribution

5.4.3 Global vs Local Explainability

Chapter 6

Applications of Graph Theory in Data Science

6.1 Social Network Analysis

6.1.1 Community Detection

6.1.2 Influence Propagation

6.1.3 Centrality for Influence Ranking

6.2 Recommendation Systems

6.2.1 Graph-Based Collaborative Filtering

6.2.2 Heterogeneous Information Networks

6.2.3 Knowledge Graphs

6.3 Natural Language Processing

6.3.1 Text as Graphs

6.3.2 Dependency Parsing Graphs

6.3.3 Graph Embeddings for NLP

6.4 Computer Vision

6.4.1 Scene Graphs

6.4.2 Graph-Based Image Segmentation

6.5 Fraud Detection

6.5.1 Transaction Networks

6.5.2 Anomaly Detection on Graphs

6.6 Biological and Chemical Graphs

Chapter 7

Graph Databases and Large-Scale Graph Processing

7.1 Graph Database Models

7.1.1 Property Graph Model

7.1.2 RDF Model

7.2 Graph Query Languages

7.2.1 Cypher

7.2.2 SPARQL

7.2.3 Gremlin

7.3 Distributed Graph Systems

7.3.1 Apache Spark GraphX

7.3.2 Google Pregel Model

7.3.3 Graph Processing Frameworks

Chapter 8

Case Studies

8.1 Social Media User Classification

8.2 Traffic Prediction Using GNNs

8.3 Fraud Detection in Financial Networks

8.4 Drug Discovery Using Molecular Graph Learning

Chapter 9

Conclusion

9.1 Summary of Findings

9.2 Future Directions in Graph-Based AI

9.3 Limitations of Graph Methods

Bibliography