Fundamentals of Accelerated Data Science

NVIDIA

Workshop Overview

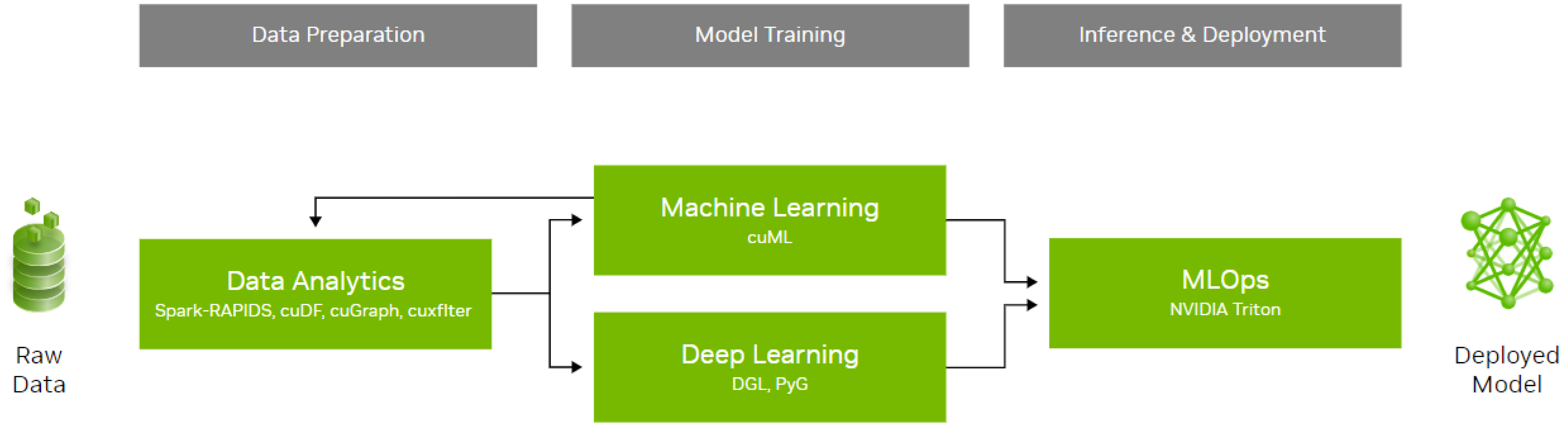
An introduction to data science with a focus on speed and efficiency

Learning objectives:

- Understand the fundamental concepts of data science and parallel computing
- Explore practical examples of accelerated data science pipelines
- Examine the methods used to achieve acceleration in data science and discuss their broader implications

This workshop will not cover statistical analysis, neural networks, and distributed computing

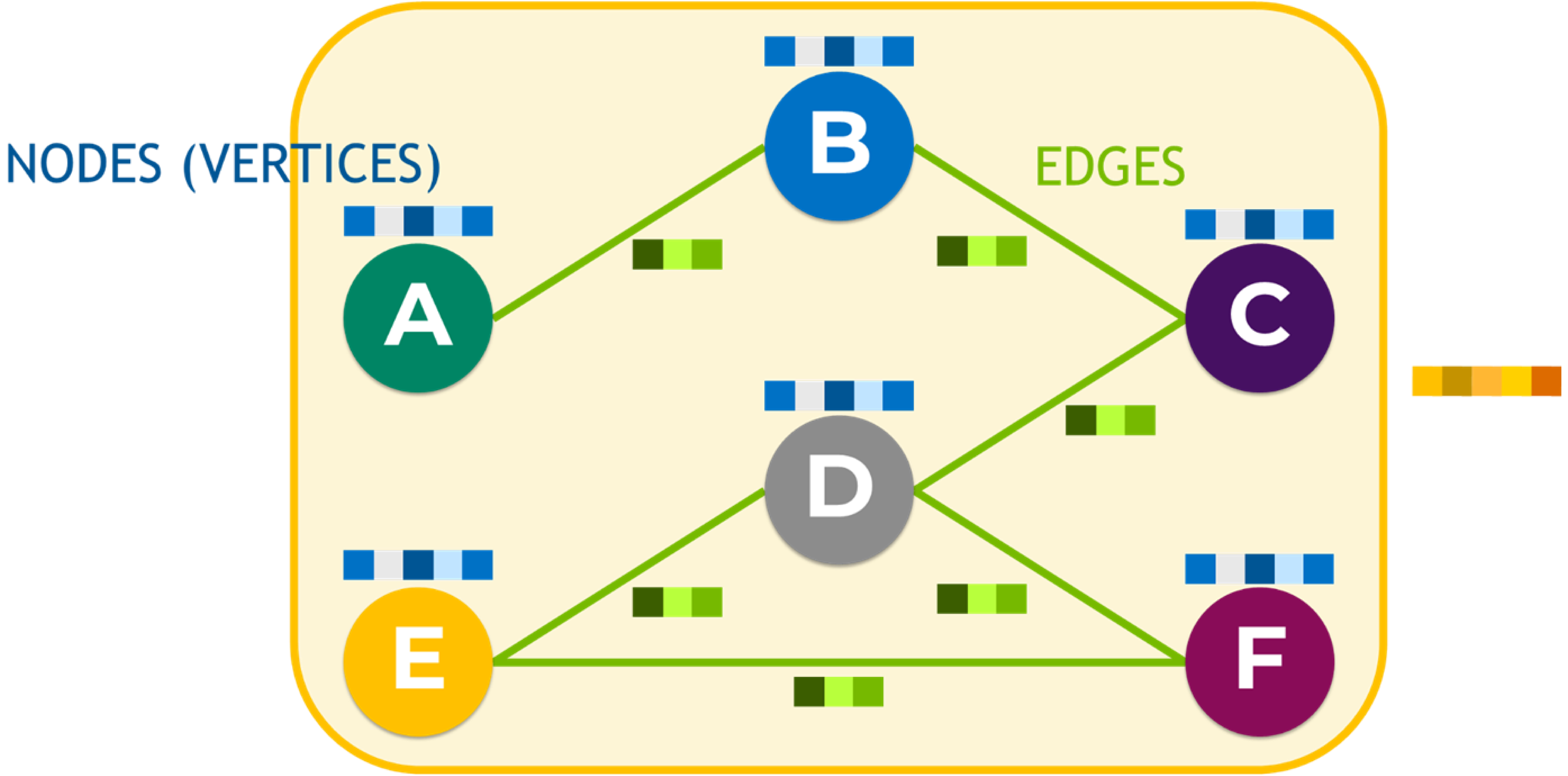
	Workshop Outline
Task 1	Data science overview Data manipulation
Task 2	Graph analytics
Task 3	Machine Learning
Coding Assessment	Biodefense



Section 2 Agenda

- Graph 101
- Graph Analytics
- Hands-On Lab

Anatomy of a Graph

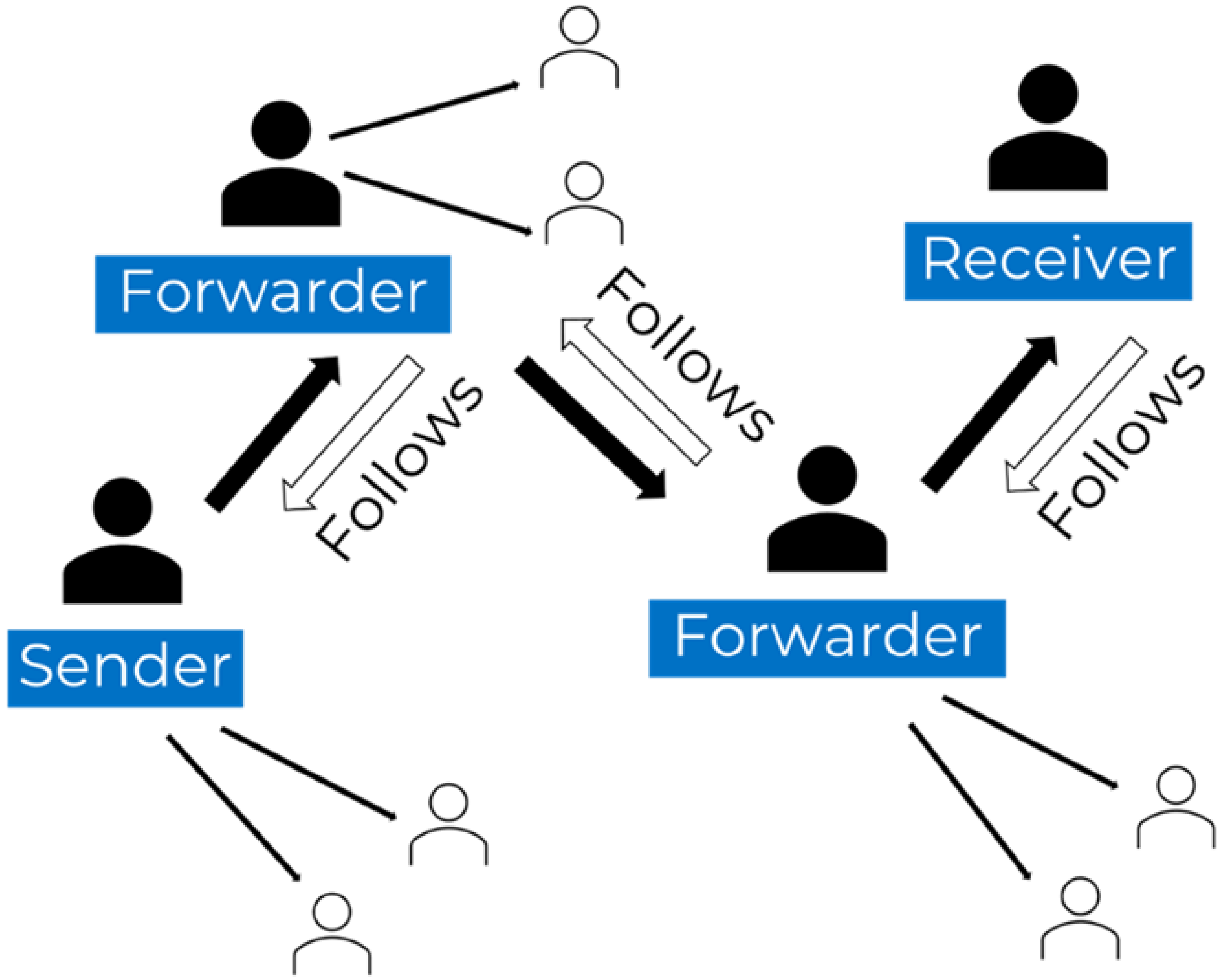


Edges

Directed, undirected, weighted, and unweighted

FACEBOOK

TWITTER



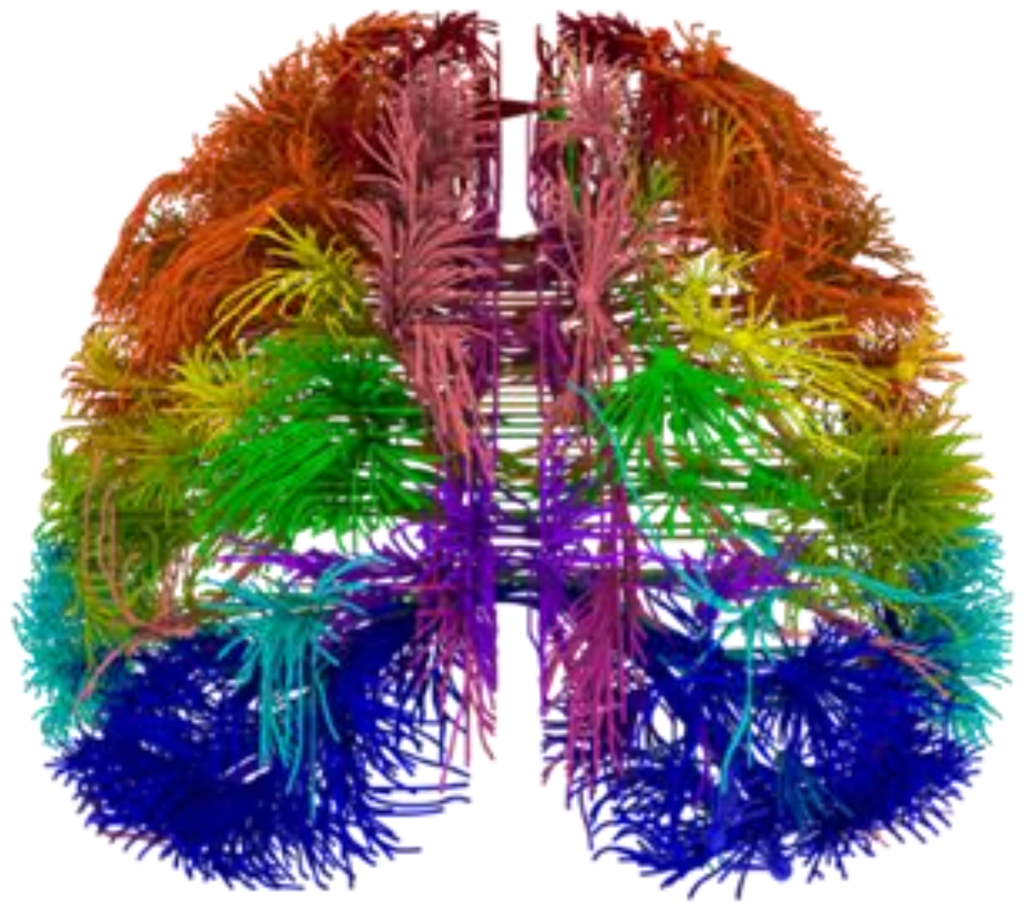
Degree and Neighborhood

In-degree, out-degree, 1-hop, and 2-hop

Common Graph Data Use Cases

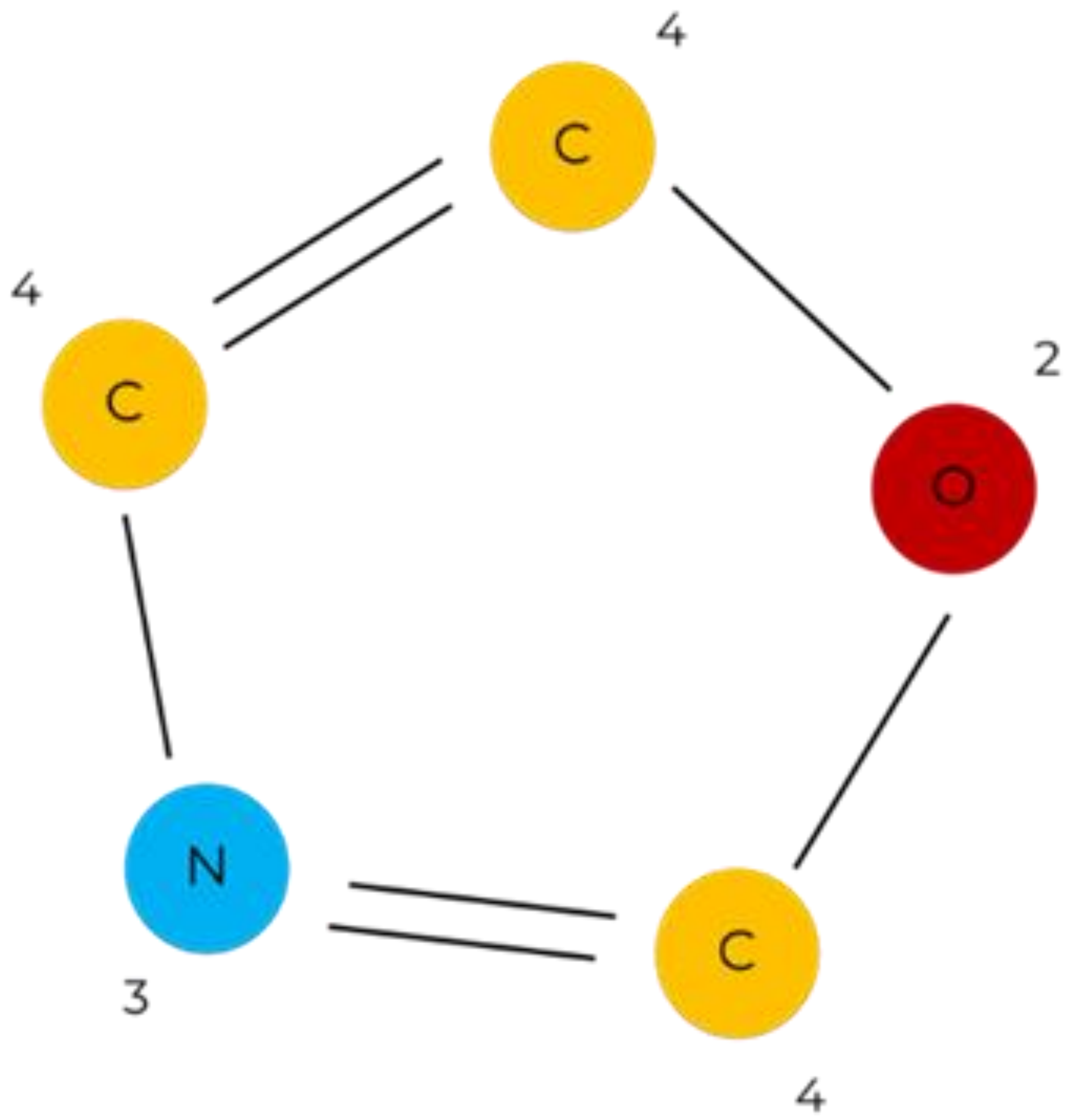
Usage extends across industries

Healthcare



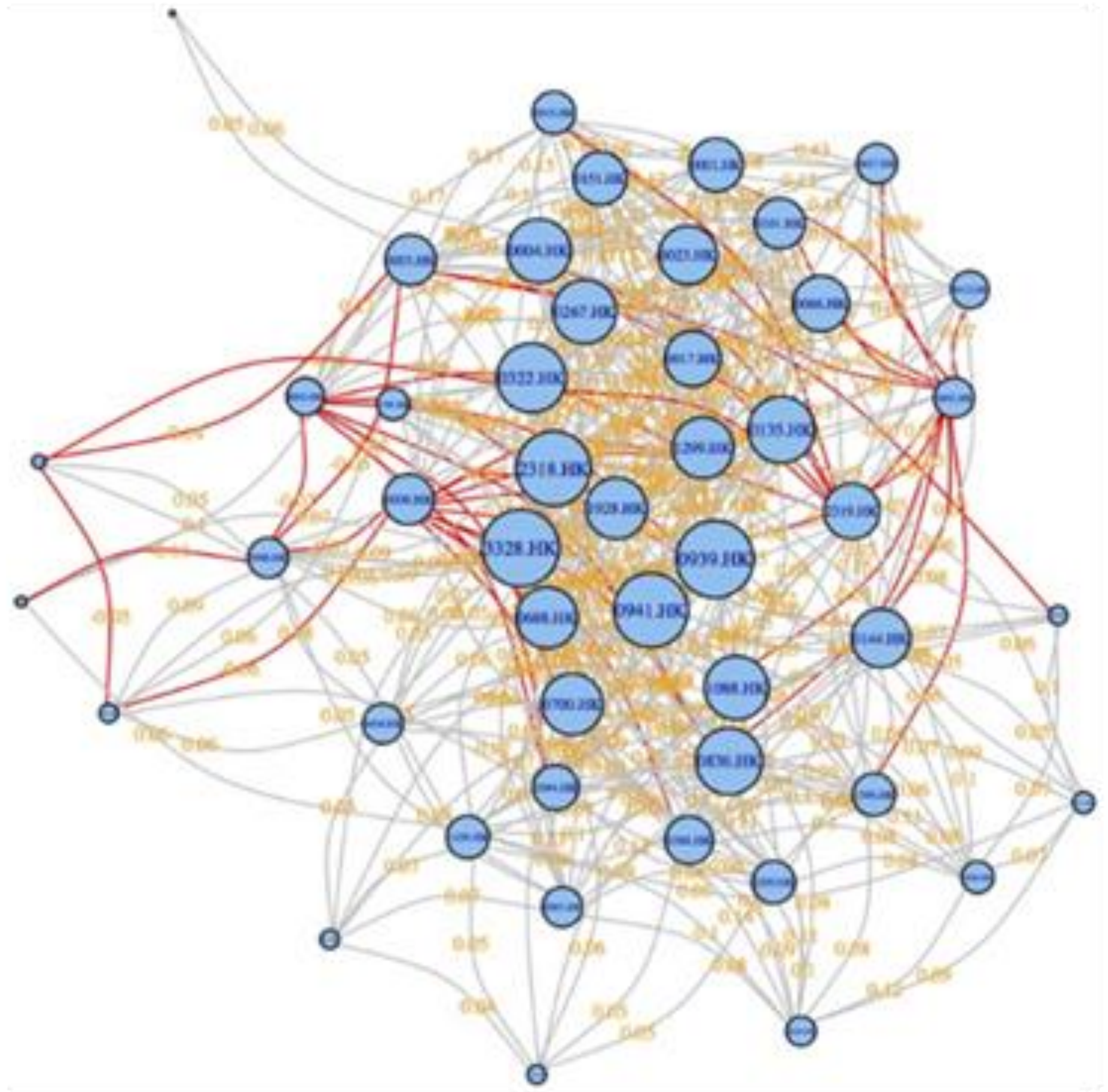
Connectomes
Brian fMRI/DTI

Chemistry & Pharmacy



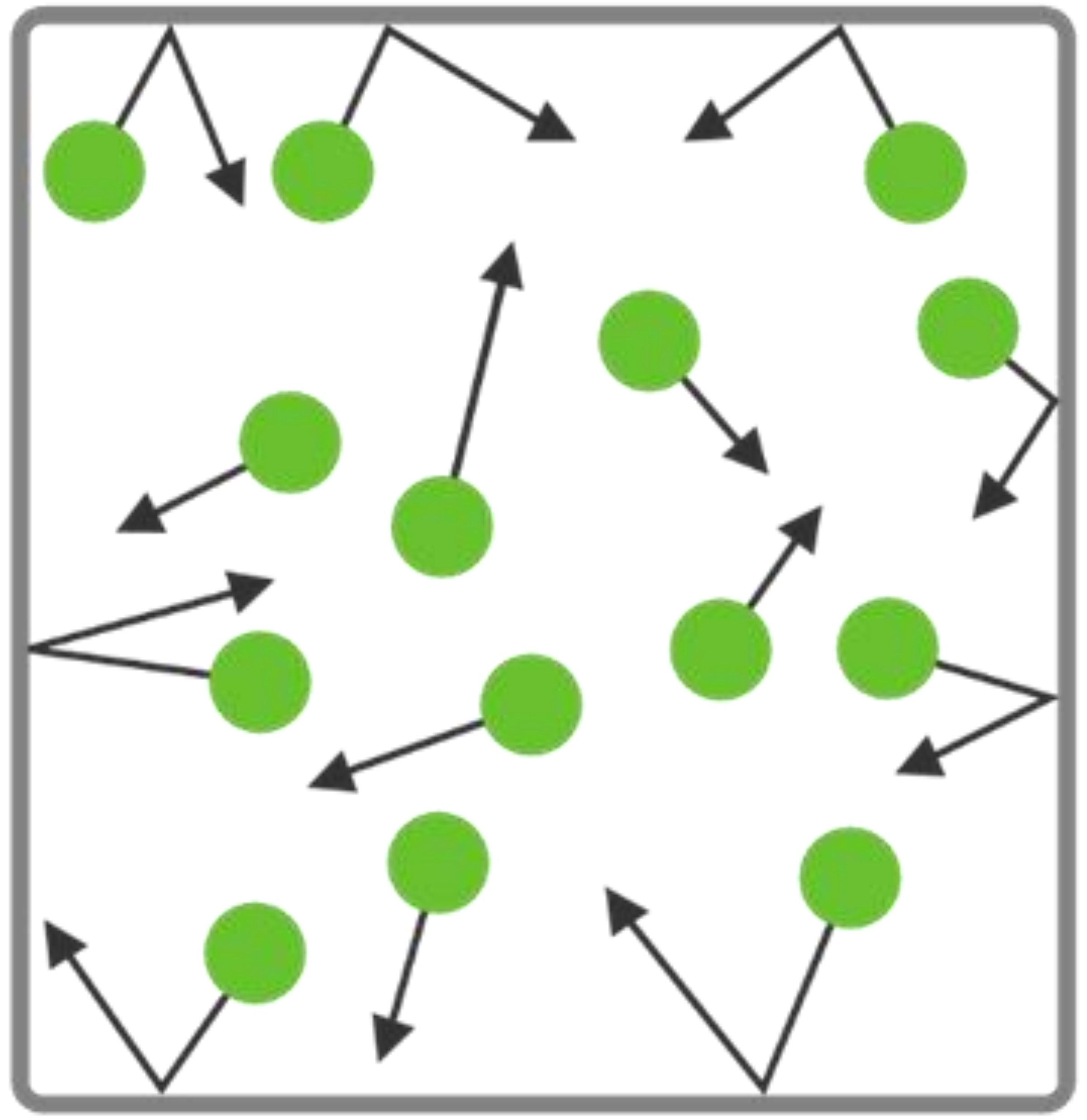
Molecular Analysis
Drug Discovery
Drug Repurposing

Financial Soundness Indicators



Stock Market Prediction

Physics

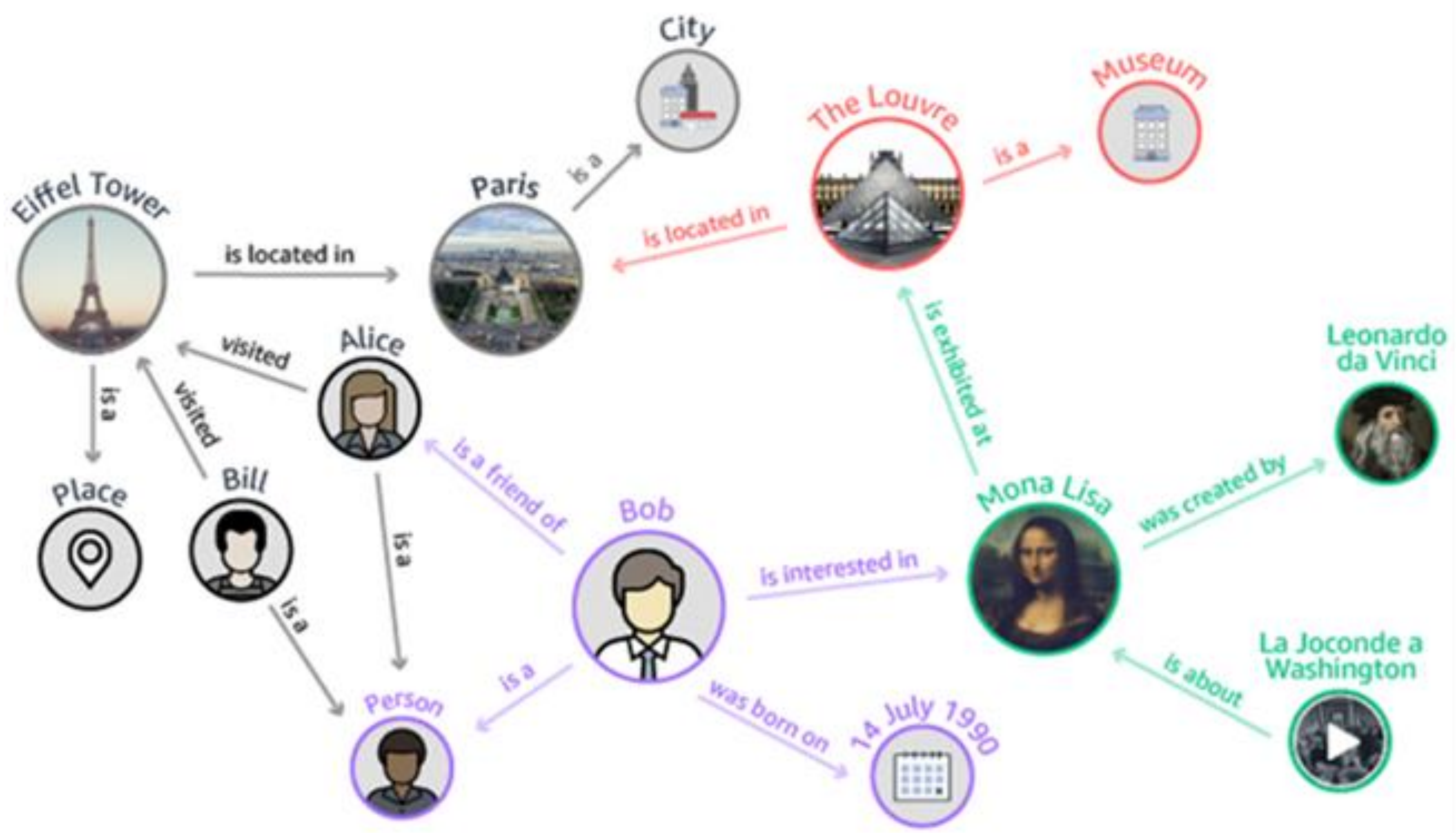


Particle Systems
Thermodynamics

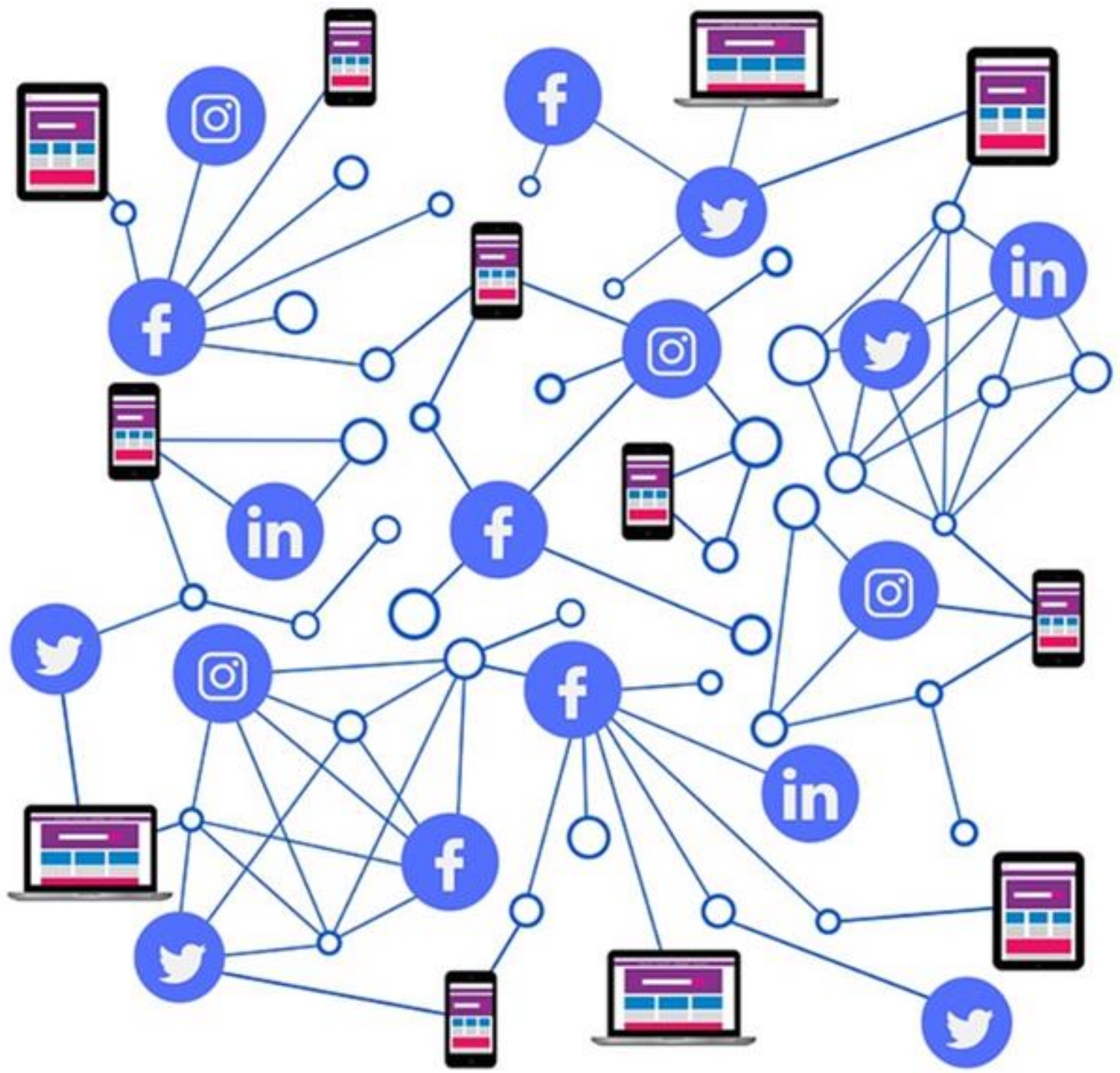
Advance Graph Data Use Cases

Used to gather information about relationships between objects

KNOWLEDGE GRAPHS



SOCIAL NETWORKS & RECOMMENDER SYSTEMS

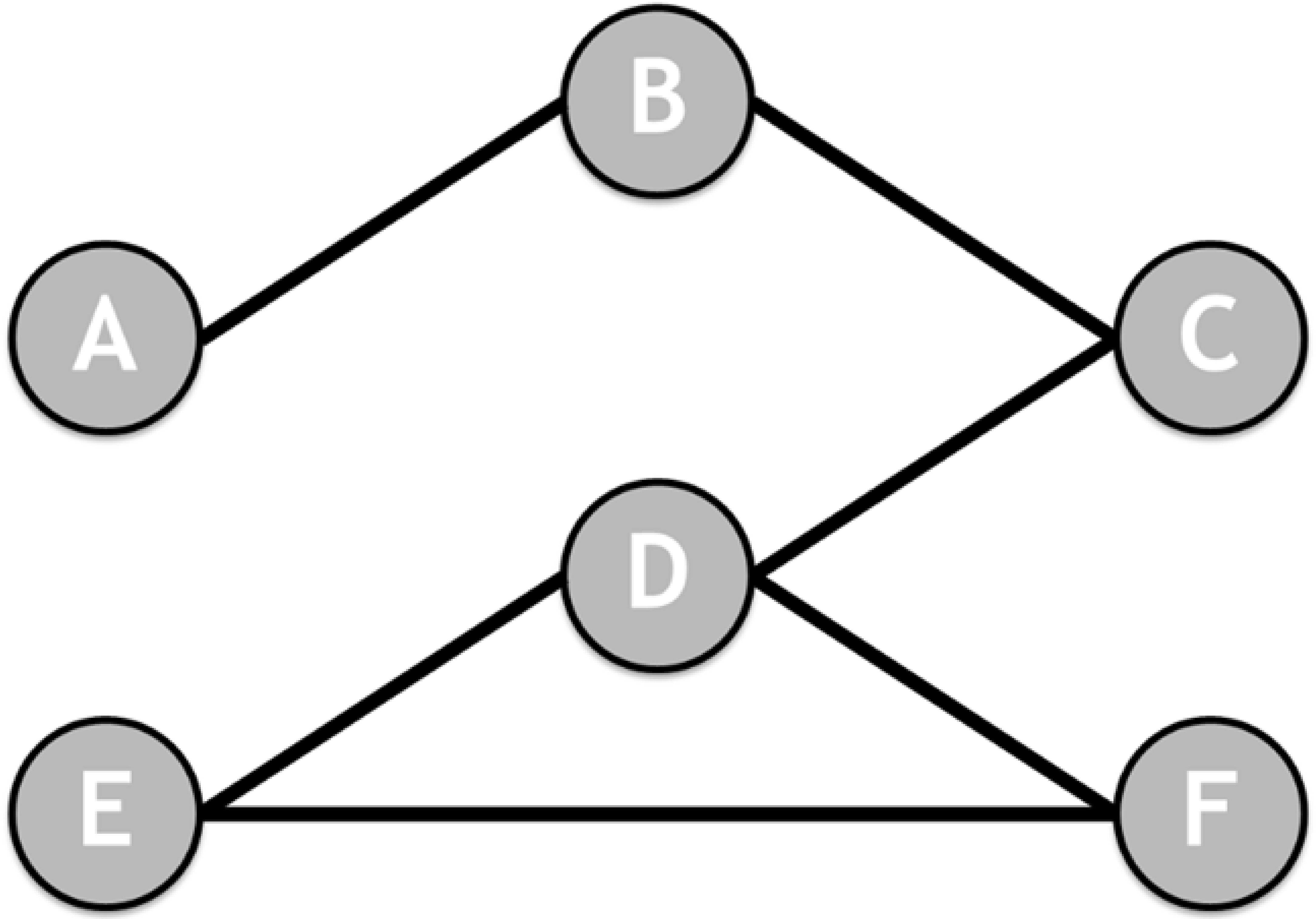


Representing Graphs

Adjacency matrix shows the neighborhood of each node

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	0	0
C	0	1	0	1	0	0
D	0	0	1	0	1	1
E	0	0	0	1	0	1
F	0	0	0	1	1	0

Note: The adjacency matrix is symmetric for undirected graphs

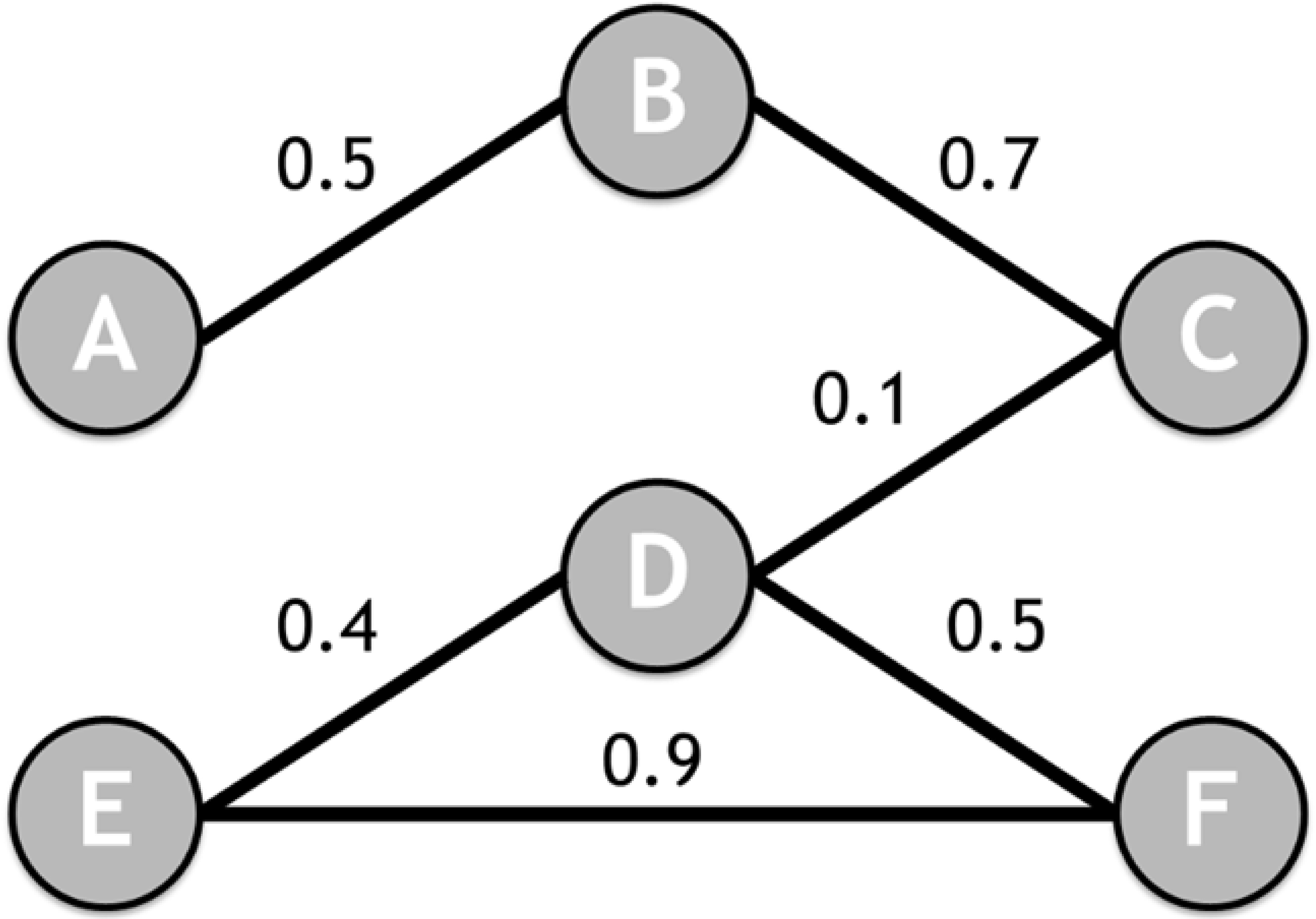


Representing Graphs

Adjacency matrix can also be used for a weighted graph

	A	B	C	D	E	F
A	0	0.5	0	0	0	0
B	0.5	0	0.7	0	0	0
C	0	0.7	0	0.1	0	0
D	0	0	0.1	0	0.4	0.5
E	0	0	0	0.4	0	0.9
F	0	0	0	0.5	0.9	0

Note: The adjacency matrix is symmetric for undirected graphs

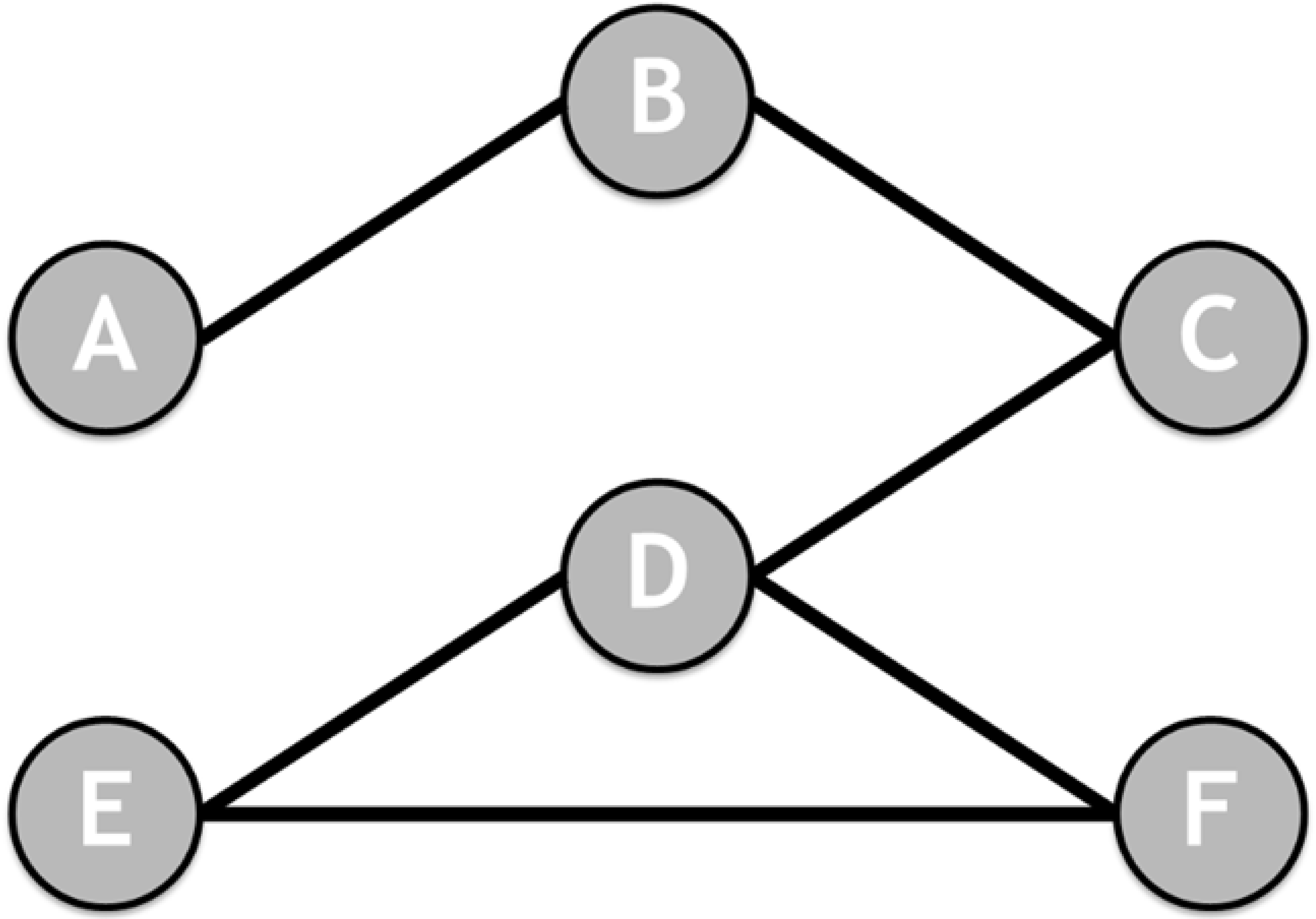


Representing Graphs

Degree matrix counts number of connections to each node

	A	B	C	D	E	F
A	1	0	0	0	0	0
B	0	2	0	0	0	0
C	0	0	2	0	0	0
D	0	0	0	3	0	0
E	0	0	0	0	2	0
F	0	0	0	0	0	2

Note: Illustration assumes unweighted and undirected graph



Representing Graphs

Laplacian matrix shows the smoothness of the graph

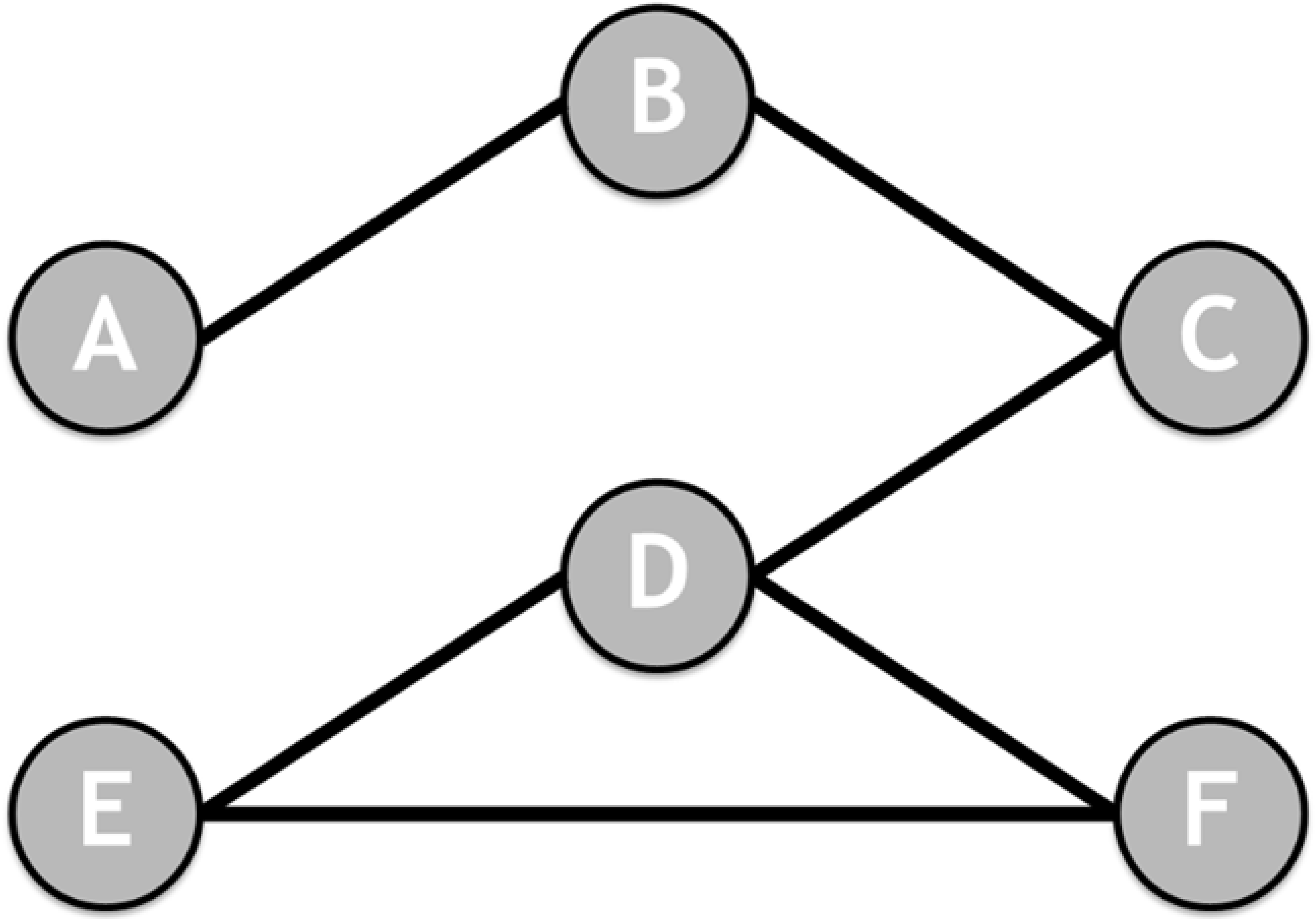
	A	B	C	D	E	F
A	1	-1	0	0	0	0
B	-1	2	-1	0	0	0
C	0	-1	2	-1	0	0
D	0	0	-1	3	-1	-1
E	0	0	0	-1	2	-1
F	0	0	0	-1	-1	2

Note: Illustration assumes unweighted and undirected graph

Represented as L, where

L = { D - A }

D = Degree Matrix and
A = Adjacency Matrix

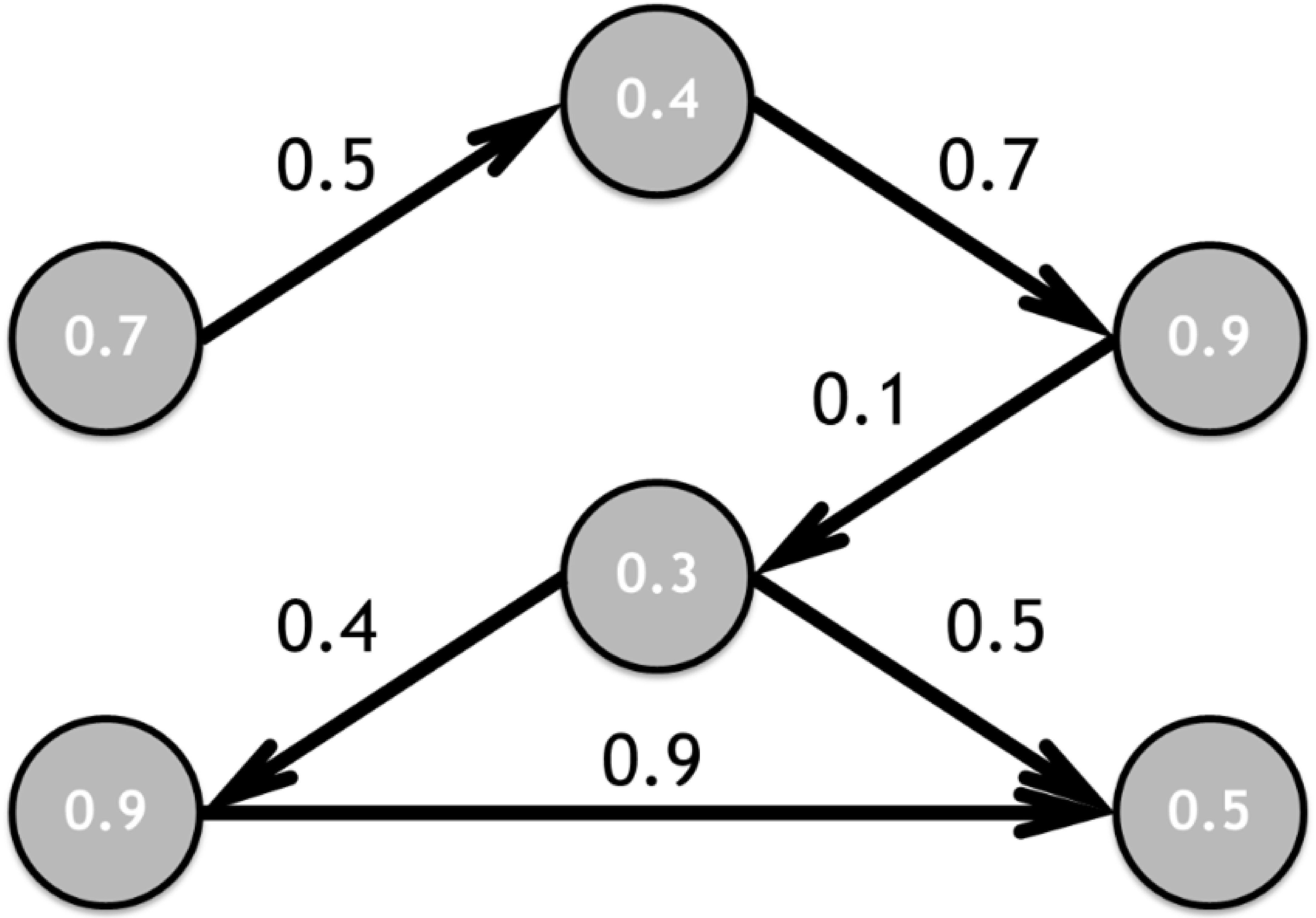


Representing Graphs

Graphs can also be represented by an adjacency list

Adjacency	Edges	Nodes
[[1, 2], [2, 3], [3, 4], [4, 5], [4, 6], [5, 6]]	[0.5, 0.7, 0.1, 0.4, 0.5, 0.9]	[0.7, 0.4, 0.9, 0.3, 0.9, 0.5]

Note: Representing graph structure as adjacency lists can be more efficient

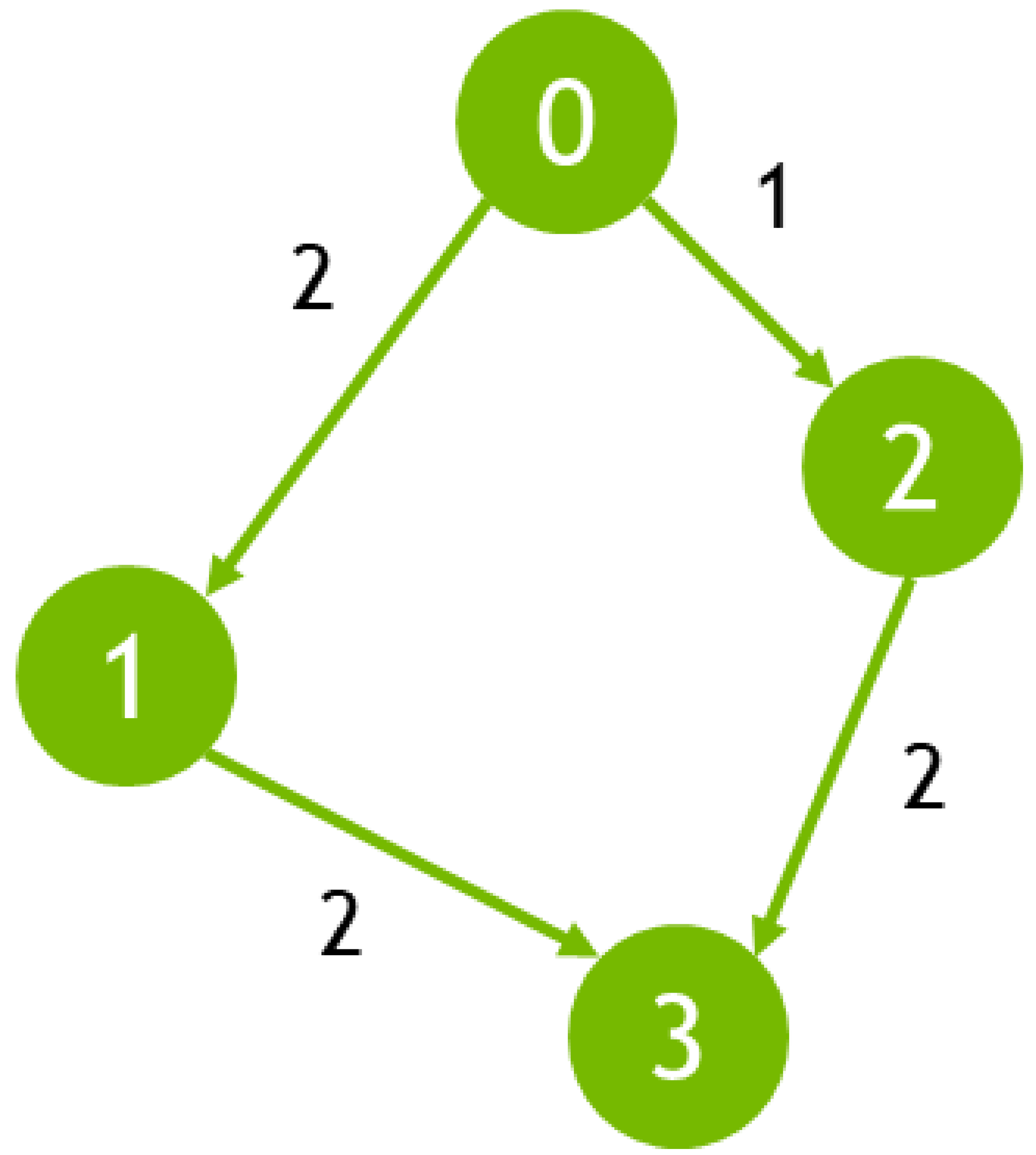


Single-Source Shortest Path

Computes the shortest path from a designated source node to all other reachable nodes in a weighted graph

Input: Graph (w/o negative-weight cycles), Node ID

Output: Vertex, Distance, Predecessor columns



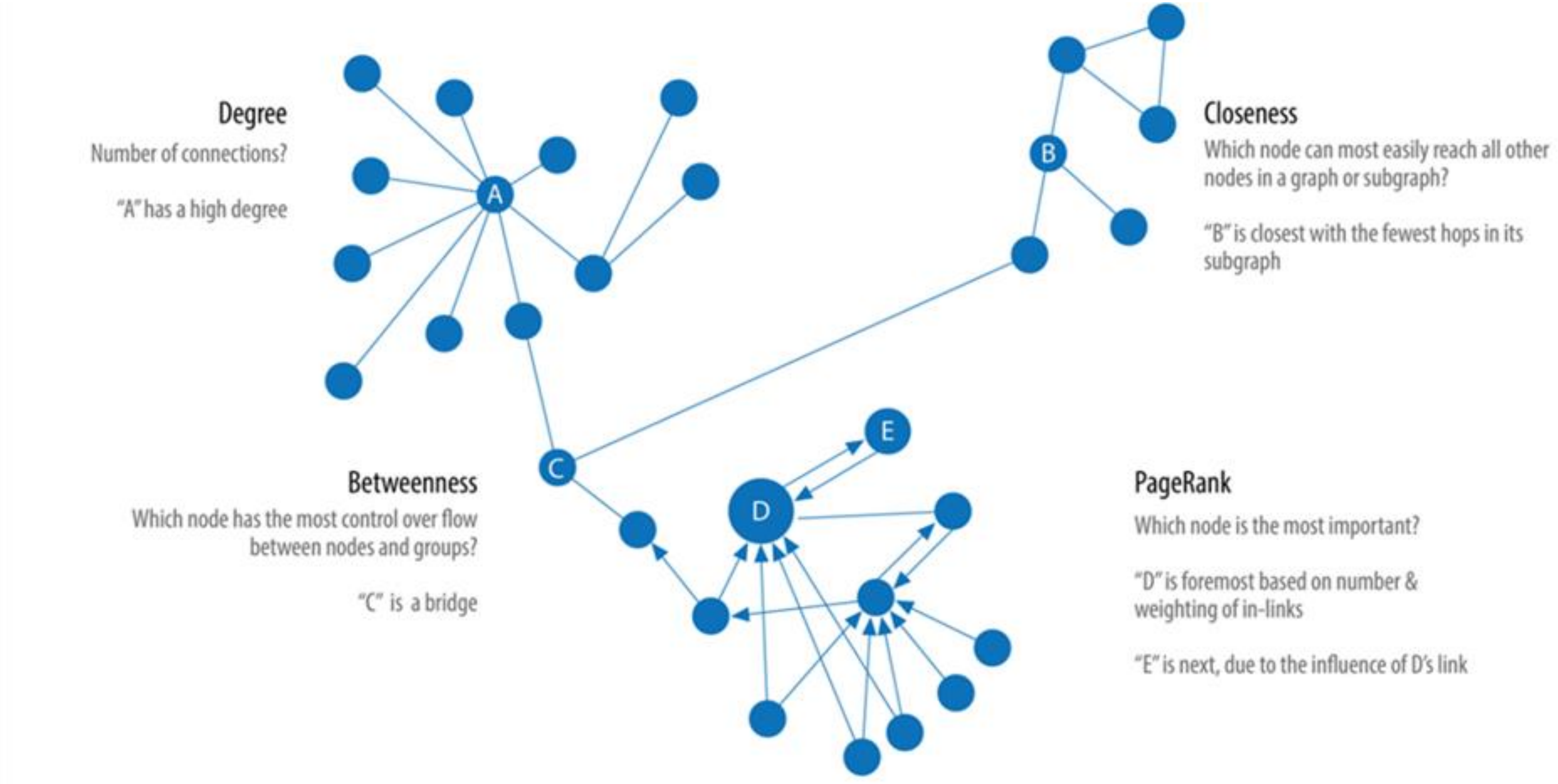
`cg.sssp(G, 0)` →

Vertex	Distance	Predecessor
0	0	-1
1	2	0
2	1	0
3	3	2

Use cases: road networks, logistics, communications, and network analysis.

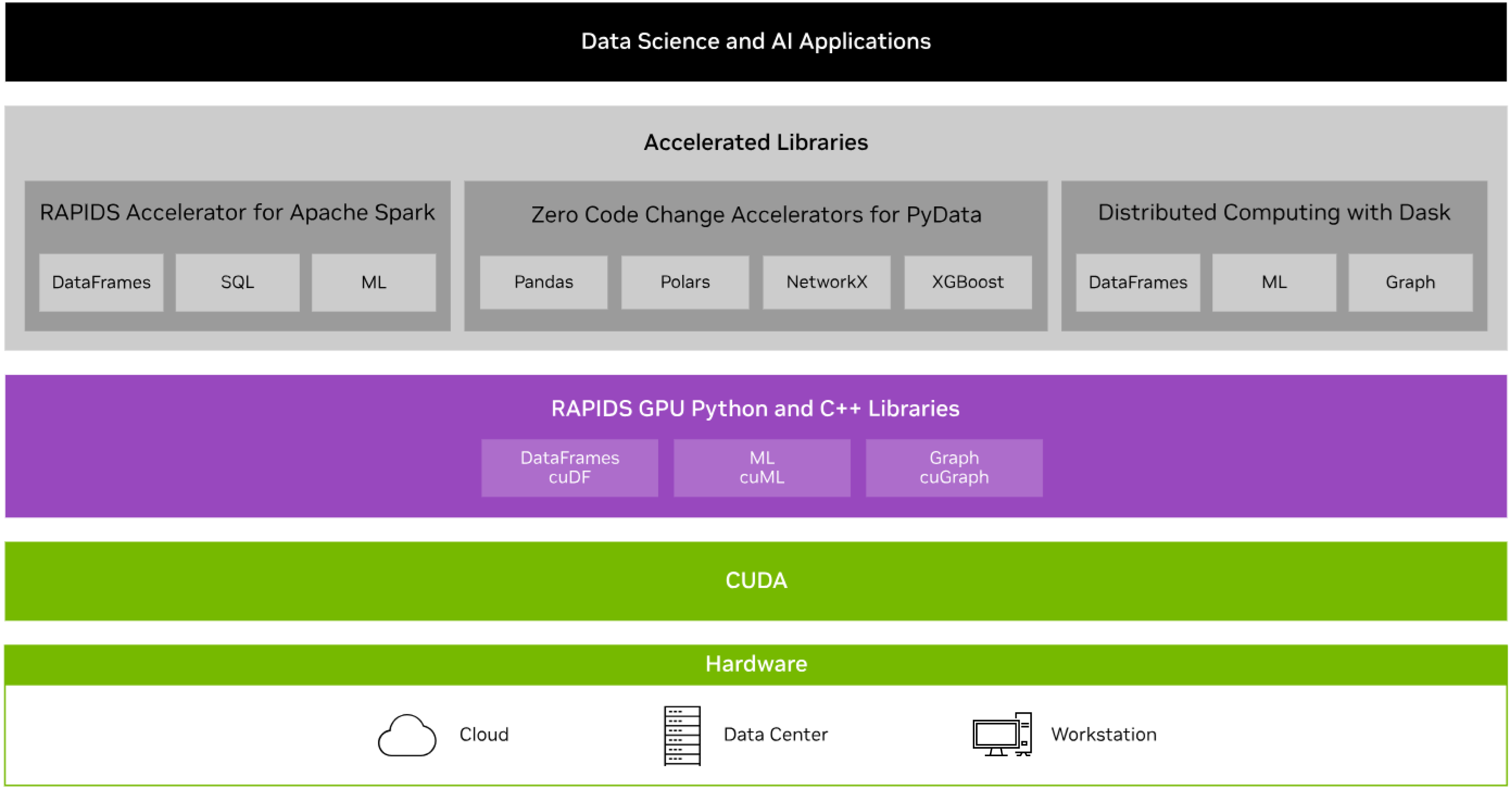
Centrality

Measuring different aspects of importance of nodes within a network



RAPIDS Transforms Data Science

An optimized hardware-to-software stack for the entire data science pipeline



nx-cugraph

NetworkX backend that provides GPU acceleration to many popular graph analytics algorithms

- Aims to bridge the gap between the ease of use of NetworkX and the high-performance capabilities of GPU-accelerated graph analytics

- Works by leverages GPUs to perform graph computations in parallel
  - Leads to faster processing, especially for large graphs and complex algorithms
  - Uses more efficient data structures than NetworkX's dictionary-based approach
  - Minimizes data movement between CPU and GPU
  - Integrates with other RAPIDS libraries like cuDF for efficient data loading and preprocessing on GPUs

- There are 3 ways to utilize nx-cugraph
  - Environment variable at runtime
  - Backend keyword argument
  - Type-based dispatching

nx-cugraph

Enable no-code acceleration using an environment variable

The **NX_CUGRAPH_AUTOCONFIG** environment variable can be used to have NetworkX automatically dispatch to specified backends. This also works in Jupyter Notebooks by using the %env magic to set the variable.

```
user@machine:/# ipython demo.ipy
CPU times: user 7min 36s, sys: 5.22 s, total: 7min 41s
Wall time: 7min 41s

user@machine:/# NX_CUGRAPH_AUTOCONFIG=True ipython demo.ipy
CPU times: user 4.14 s, sys: 1.13 s, total: 5.27 s
Wall time: 5.32 s
```

nx-cugraph

Explicitly specify the cugraph backend

Backend keyword argument: NetworkX also supports explicitly specifying a particular backend for supported APIs with the **backend** keyword argument

```
nx.betweenness_centrality(cit_patents_graph, k=k, backend="cugraph")
```

nx-cugraph

Type-based dispatching

Type-based dispatching: for users wanting to ensure a particular behavior, without the potential for runtime conversions, NetworkX offers type-based dispatching. To utilize this method, users must import the desired backend and create a Graph instance for it.

```
import networkx as nx
import nx_cugraph as nxcg

G = nx.Graph()

# populate the graph
#  ...

nxcg_G = nxcg.from_networkx(G)             # conversion happens once here
nx.betweenness_centrality(nxcg_G, k=1000)  # nxcg Graph type causes cugraph backend
                                           # to be used, no conversion necessary
```

Hands-On Lab