DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Part I

Introduction and Fundamentals

Lecture 02

Graph Theory for GNNs

Discover the core elements of graph theory, covering graph properties, concepts, and essential algorithms like BFS and DFS. Learn about different graph types, properties, and their real-world applications. Gain a solid grasp of fundamental graph measures and algorithms using NetworkX library.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Unveiling Graph Theory

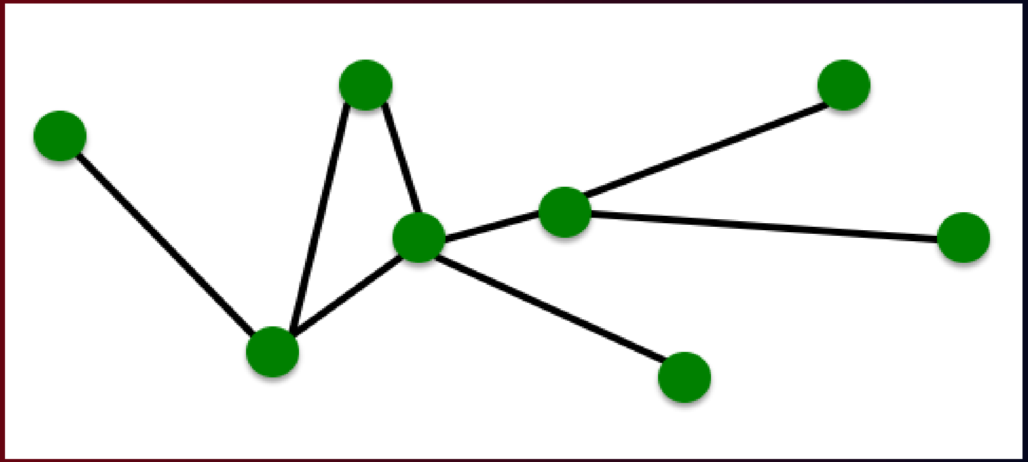
- **Graph theory** is fundamental in mathematics, focusing on the study of graphs and networks.
- Crucial in modeling and analyzing real-world challenges like transportation systems, social networks, and internet connectivity.

Definition & Elements of a Graph

- In **graph theory**, a **graph** comprises **vertices** (**nodes**) and **edges**, represented as:

$$G(V, E)$$

Graph — Vertices — Edges



- **Objects**: nodes, vertices V
- **Interactions**: links, edges E
- **System**: network, graph G(V,E)

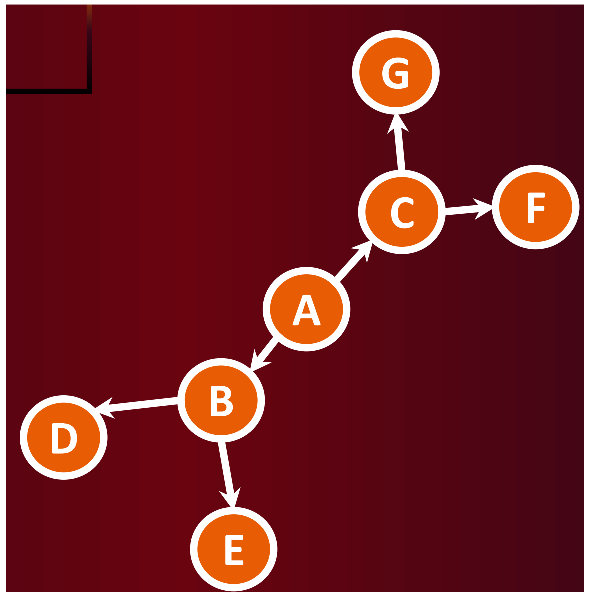
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Essential Graph Properties

01 **Directed Graphs**
Most basic properties of a graph is whether it is **directed** (**digraph**) or **undirected**.

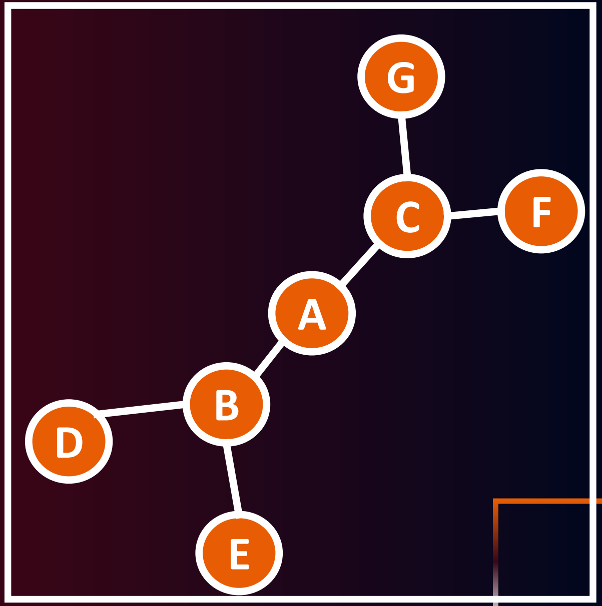
02 Weighted Graphs

03 Connected Graphs



Directed Graphs

- ❑ **Edges** possess a defined **direction** or **orientation**.
- ❑ **Edges** connect **nodes** with one as the **source** and the other as the **destination**.



Undirected Graphs

- ❑ **Edges** lack a specific **direction**.
- ❑ **Edges** allow traversal in both **directions**.
- ❑ The order of visiting **nodes** is insignificant.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Essential Graph Properties

01 **Directed Graphs**
Most basic properties of a graph is whether it is **directed** (**digraph**) or **undirected**.

02 Weighted Graphs

03 Connected Graphs

Directed Graphs

```
# Importing the networkx library as 'nx'
import networkx as nx
# Creating an instance of a directed graph
DG = nx.DiGraph()
# Adding edges to the graph
DG.add_edges_from([('A', 'B'), ('A', 'C'),
                   ('B', 'D'),('B', 'E'),
                   ('C', 'F'), ('C', 'G')])
```

Undirected Graphs

```
# Importing the networkx library as 'nx'
import networkx as nx
# Creating an instance of an undirected graph
UG = nx.Graph()
# Adding edges to the graph
UG.add_edges_from([('A', 'B'), ('A', 'C'),
                   ('B', 'D'),('B', 'E'),
                   ('C', 'F'), ('C', 'G')])
```

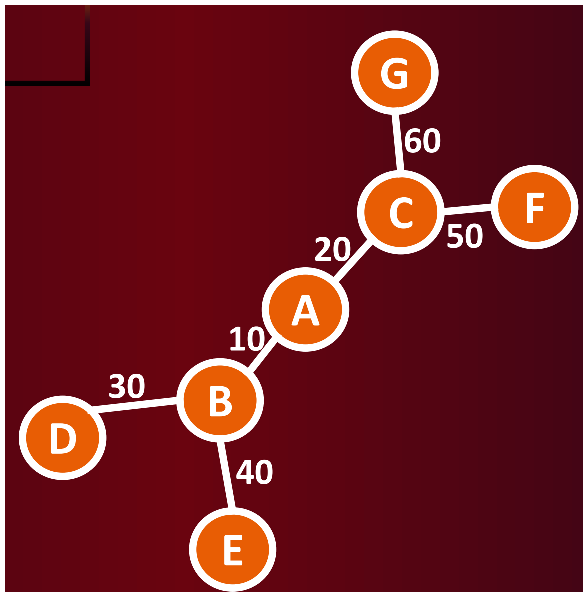
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Essential Graph Properties

01 Directed Graphs
Most basic properties of a graph is whether it is **directed** (**digraph**) or **undirected**.

02 Weighted Graphs
Another important property of graphs is whether the edges are **weighted** or **unweighted**.

03 Connected Graphs

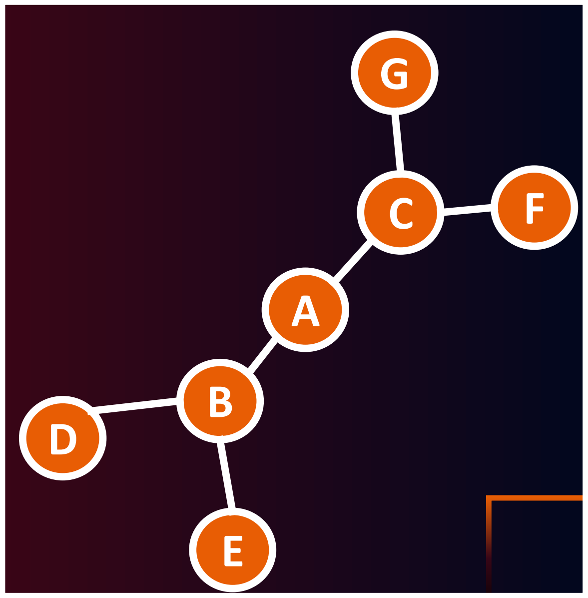


Weighted Graphs

- Each **edge** is associated with a **weight** or **cost**.
- **Weights** represent factors like distance, travel time, or cost.
- Useful in scenarios where precise measurements between **nodes** matter.

Unweighted Graphs

- **Edges** lack associated **weights**.
- Suitable for scenarios where **node relationships** are **binary**.
- **Edges** indicate the presence or absence of a connection between nodes.



DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Essential Graph Properties

Weighted Graphs

01 **Directed Graphs**
Most basic properties of a graph is whether it is **directed** (**digraph**) or **undirected**.

02 **Weighted Graphs**
Another important property of graphs is whether the edges are **weighted** or **unweighted**.

03 **Connected Graphs**

```python
# Importing the networkx library as 'nx'
import networkx as nx

# Creating an instance of an undirected graph
WG = nx.Graph()

# Adding edges to the graph along with their weights
WG.add_edges_from([('A', 'B', {"weight": 10}),
                   ('A', 'C', {"weight": 20}),
                   ('B', 'D', {"weight": 30}),
                   ('B', 'E', {"weight": 40}),
                   ('C', 'F', {"weight": 50}),
                   ('C', 'G', {"weight": 60})])

# Retrieving the weights associated with each edge
labels = nx.get_edge_attributes(WG, "weight")
print(labels)
```

```
{('A', 'B'): 10, ('A', 'C'): 20, ('B', 'D'): 30, ('B',
'E'): 40, ('C', 'F'): 50, ('C', 'G'): 60}
```

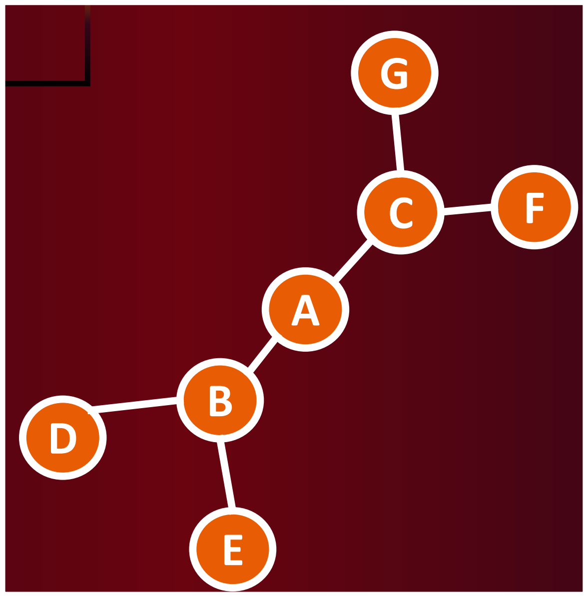
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Essential Graph Properties

01 **Directed Graphs**
Most basic properties of a graph is whether it is **directed** (**digraph**) or **undirected**.

02 **Weighted Graphs**
Another important property of graphs is whether the edges are **weighted** or **unweighted**.

03 **Connected Graphs**
A Graph can be either **connected** or **disconnected.**

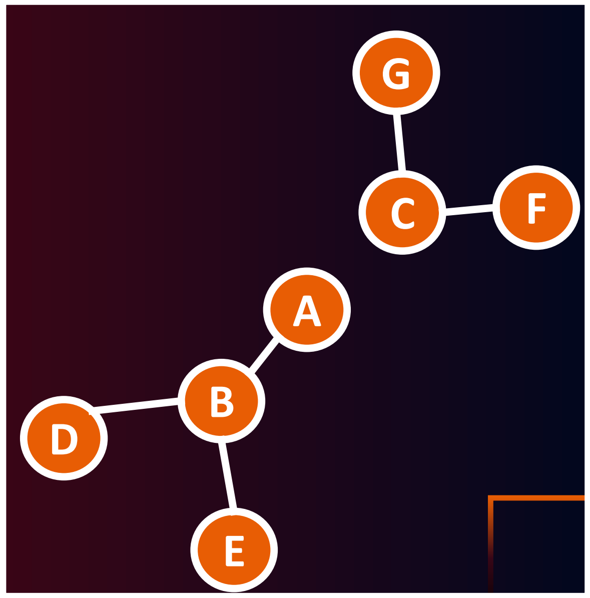


Connected Graphs

- Describe a graph that is composed of nodes or vertices that are all connected to each other.
- For every pair of nodes in the graph, there exists a **path** (or sequence of nodes) that connects them.

Disconnected Graphs

- Contain **isolated nodes** that cannot communicate with other **nodes directly**.
- Pose challenges in designing efficient routing algorithms due to fragmented communication.



DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz


Essential Graph Properties

Connected/Disconnected Graphs

01 **Directed Graphs**

Most basic properties of a graph is whether it is **directed** (**digraph**) or **undirected**.

02 **Weighted Graphs**

Another important property of graphs is whether the edges are **weighted** or **unweighted**.

03 **Connected Graphs**

A Graph can be either **connected** or **disconnected.**

```python
# Importing the networkx library as 'nx'
import networkx as nx

# Creating an instance of an undirected graph
G1 = nx.Graph()
# Adding edges to the graph for G1
G1.add_edges_from([(1, 2), (2, 3), (3, 1), (4, 5)])
# Check Connectivity of G1 and Print Results
print(f"Is graph G1 connected? {nx.is_connected(G1)}")

# Creating another instance of an undirected graph
G2 = nx.Graph()
# Adding edges to the graph for G2
G2.add_edges_from([(1, 2), (2, 3), (3, 1), (1, 4)])
# Check Connectivity of G2 and Print Results
print(f"Is graph G2 connected? {nx.is_connected(G2)}")
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Essential Graph Properties

01 **Directed Graphs**

Most basic properties of a graph is whether it is **directed** (**digraph**) or **undirected**.

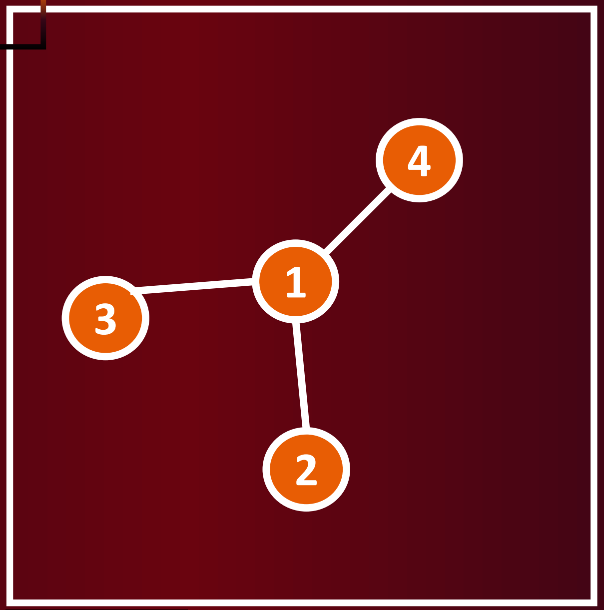
02 **Weighted Graphs**

Another important property of graphs is whether the edges are **weighted** or **unweighted**.

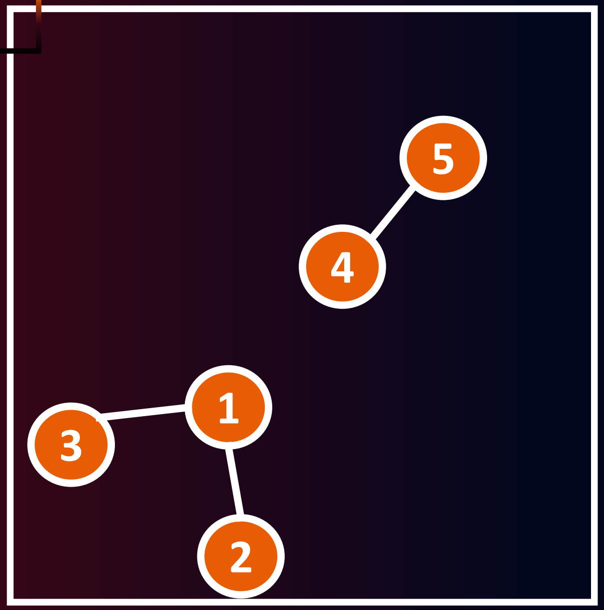
03 **Connected Graphs**

A Graph can be either **connected** or **disconnected.**

Connected/Disconnected Graphs



Graph G2

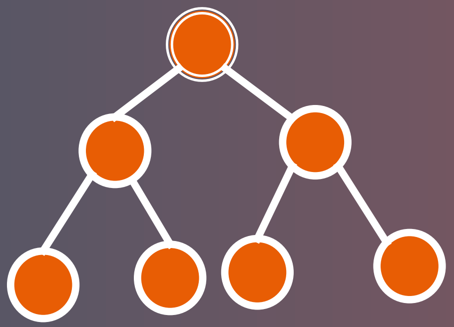


Graph G1

```
Is graph G1 connected? False
Is graph G2 connected? True
```

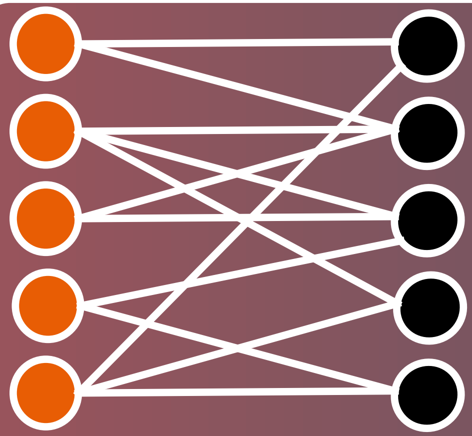
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz


Advanced Types of Graphs



Rooted Tree

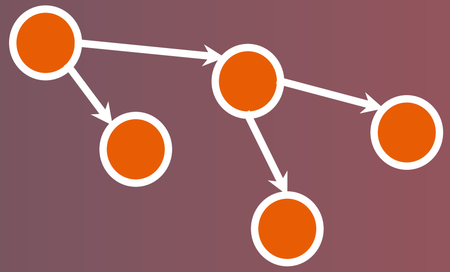
- A tree structure with a designated root node.
- Widely used in computer science to represent hierarchical data structures.
- Commonly utilized for modeling file systems and the structure of XML documents.



Bipartite Graph

- Vertices are partitioned into two sets, with edges only connecting vertices from different sets.
- Frequently used in mathematics and computer science.
- Represents relationships between distinct types of objects, like buyers and sellers, or employees and projects.

- A directed graph with no cycles; edges only traverse in a specific direction.
- Commonly used to model dependencies between tasks or events.
- Vital in project management or computing the critical path of a job.



Directed Acyclic Graph (DAG)

- A fully connected graph.
- Widely used in combinatorics to model problems involving all possible pairwise connections.
- Commonly employed in computer networks to model fully connected networks.

Complete Graph

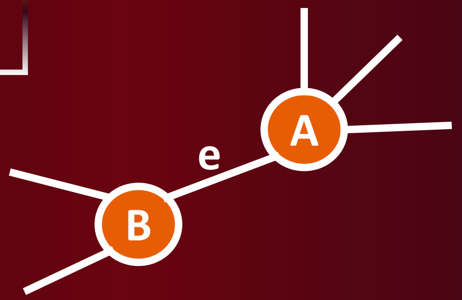
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz


Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation



- e = {A, B} is <u>incident</u> to **A** and **B**, or joins **A** and **B**.
- $\boldsymbol{Deg}(A) = 4$
- $\boldsymbol{Deg}(B) = 3$

Degree of a Node

- Represents the number of edges <u>**incident**</u> on a node v_i.
  - Edge is **incident** if the node is one of its endpoints.
- Denoted by $\boldsymbol{Deg}(v_i)$.
- Applicable to both **directed** and **undirected** graphs

For Undirected Graphs

- **Node degree** is the <u>count</u> of **edges** connected to the **node**.
- If a **node** has a **self-loop**, it adds **two** to the **degree**.

For Directed Graphs

- **Degree** is divided into **indegree** and **outdegree**.
- **Indegree** $(\boldsymbol{Deg}^{-}(v_i))$ is **edges** <u>pointing</u> towards the **node**.
- **Outdegree** $(\boldsymbol{Deg}^{+}(v_i))$ is **edges** <u>starting</u> from the **node**.
- **Self-loops** add **one** to both **indegree** and **outdegree**.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation

Degree of a Node: Undirected Graph

```
import networkx as nx

UG = nx.Graph()
UG.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'),
                   ('B','E'), ('C', 'F'), ('C', 'G')])
print(f"deg(A) = {UG.degree['A']}")
```

```
deg(A) = 2
```

Degree of a Node: Directed Graph

```
DG = nx.DiGraph()
DG.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'),
                   ('B','E'), ('C', 'F'), ('C', 'G')])
print(f"deg^-(A) = {DG.in_degree['A']}")
print(f"deg^+(A) = {DG.out_degree['A']}")
```

```
deg^-(A) = 0
deg^+(A) = 2
```

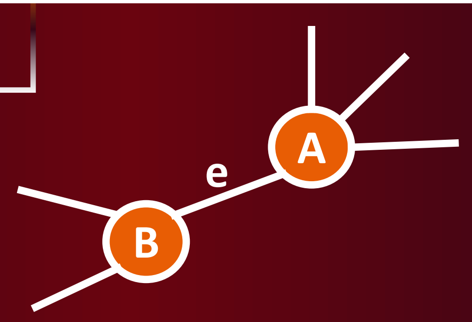
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation



- **A** and **B** are neighbors

Neighbors

- **Neighbors** are **nodes** directly connected to a particular **node** via an edge.
- **Adjacency**: Two nodes are **adjacent** if they share at least one common neighbor.
- **Neighbors & Adjacency** are crucial for **Path Searching** and **Cluster Identification**.

Node Importance:

- **Degrees** and **paths** help determine a node's importance in a network.

Paths

- A **path** is a sequence of **edges** connecting two (or more) **nodes** in a graph.
- **Path length** is the count of **edges** along the **path**.
- **Types of paths:** Simple path (no repeated nodes except start and end), Cycle (first and last vertices are the same).

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

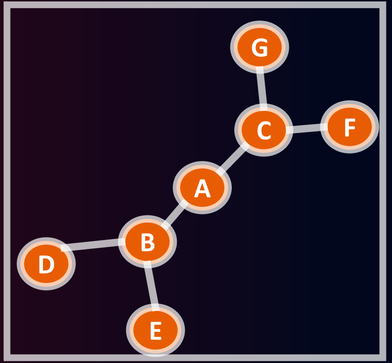
02 Graph Measures

03 Graph Representation

Neighbors in Undirected Graphs

```
# Get neighbor of node ‘C’ in the unDirected Graph UG
for neighbor in UG.neighbors("C"):
  print("Node {} has neighbor {}".format("C", neighbor))
```

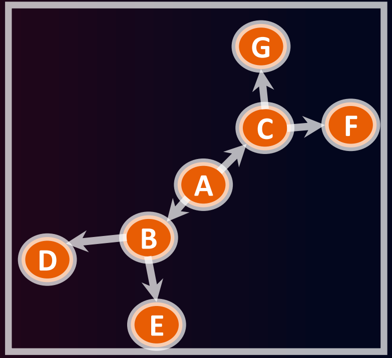
```
Node C has neighbor A
Node C has neighbor F
Node C has neighbor G
```



Neighbors in Directed Graphs

```
# Get neighbor of node ‘C’ in the Directed Graph DG
for neighbor in DG.neighbors("C"):
  print("Node {} has neighbor {}".format("C", neighbor))
```

```
Node C has neighbor F
Node C has neighbor G
```



DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz


Discovering Graph Concepts

01 Degrees & Neighbors

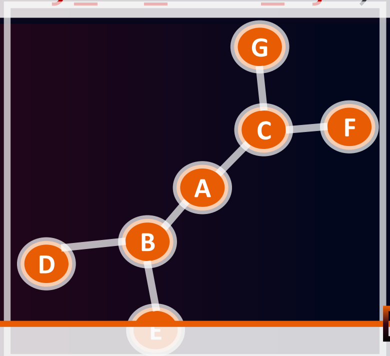
02 Graph Measures

03 Graph Representation

Adjacency of a Node in Graphs

```
# Node for which you want to get the adjacency
source_node = 'C'
# Get the adjacency of the specified node
adjacency_of_node_c = list(UG.adj[source_node])
# Print the adjacency of node "C"
print(f"Adjacency of node {node}: {adjacency_of_node_c}")
```

```
Adjacency of node C: ['A', 'F', 'G']
```



Path Length in Graphs

```
target_node = 'G'
# Get the path length from node "C" to node "G"
path_length = nx.shortest_path_length(G,
                                      source=source_node,
                                      target= target_node)
# Print the path length
print(f"Path length from node {source_node} to node
{target_node}: {path_length}")
```

```
Path length from node C to node G: 1
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

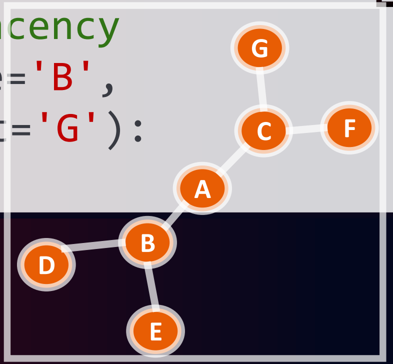
02 Graph Measures

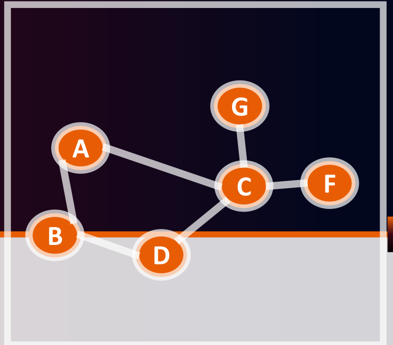
03 Graph Representation

Simple Paths in Graphs

```
# Node for which you want to get the adjacency
for path in nx.all_simple_paths(G, source='B',
                                   target='G'):
    print(path)
```

```
['B', 'A', 'C', 'G']
```





Cycle Paths in Graphs

```
# Create a graph with a cycle
CG = nx.Graph()
CG.add_edges_from([('A', 'B'), ('B', 'D'), ('D', 'C'),
                   ('C', 'A'), ('C', 'G'), ('C', 'E')])
# Node for which you want to get the adjacency
for path in nx.all_simple_paths(G, source='B',
                                   target='G'):
    print(path)
```

```
['A', 'B', 'D', 'C']
```

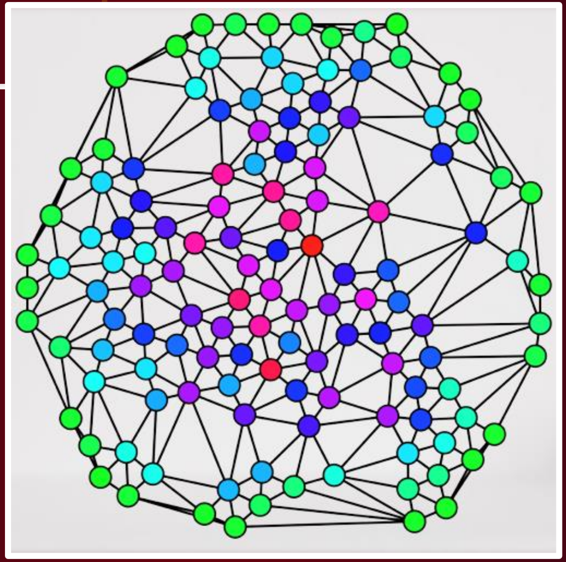
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz


Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

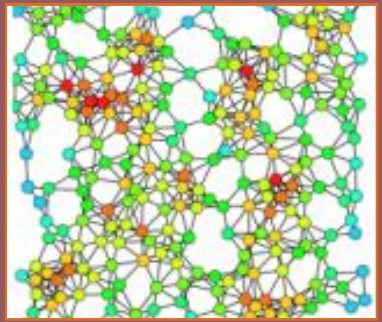
03 Graph Representation



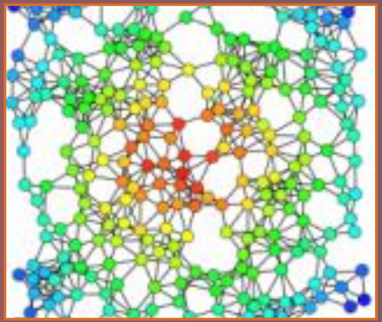
Centrality Measure

- Quantifies node importance based on connectivity and influence in a network.
- Helps identify key nodes impacting information or interactions flow.
- Several measures of centrality, each providing a different perspective on the importance of a node

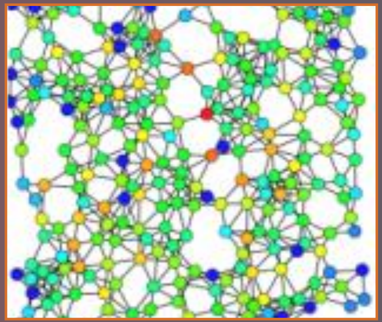
Most used Centrality Measures:



Degree Centrality



Closeness Centrality



Betweenness Centrality

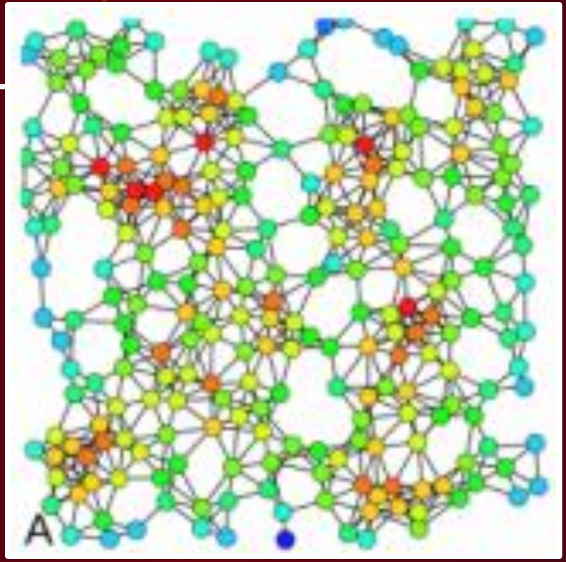
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz


Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation



Degree Centrality

- Simple and common centrality measure.
- Defined as the number of edges incident to a node.
- High degree centrality indicates strong connections and influence in the network.

Note: The **Normalized Degree Centrality** is : $C_D^*(v_i) = \frac{\deg(v_i)}{|v|-1}$

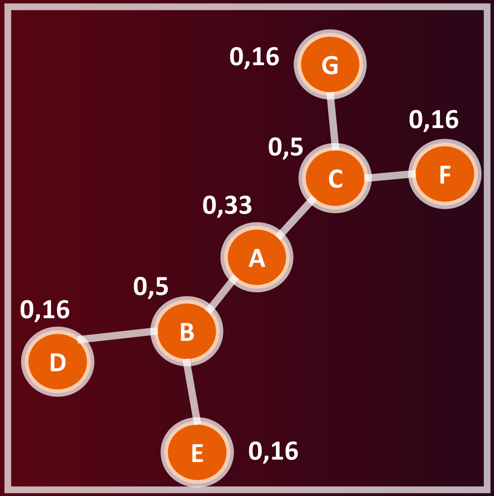
Note: For Directed Graphs, extra two degrees centralities are existed:

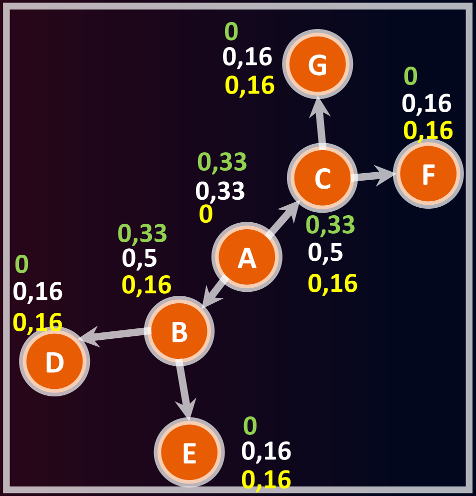
In Degree Centrality:

$$C_D^{*-}(v_i) = \frac{\deg^-(v_i)}{|v|-1}$$

Out Degree Centrality:

$$C_D^{*+}(v_i) = \frac{\deg^+(v_i)}{|v|-1}$$





DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation

Degree Centrality for Undirected Graphs

```
print(nx.degree_centrality(UG))                    #deg(v_i)
```

$C_D^*(v_i)$

```
{'A': 0.3333333333333333, 'B': 0.5, 'C': 0.5,
 'D': 0.16666666666666666, 'E': 0.16666666666666666,
 'F': 0.16666666666666666, 'G': 0.16666666666666666}
```

Degrees Centralities for Directed Graphs

```
print(nx.degree_centrality(DG))                    #deg(v_i)
print(nx.in_degree_centrality(DG))                 #deg^(-)(v_i)
print(nx.out_degree_centrality(DG))                #deg^(+)(v_i)
```

$C_D^*(v_i)$

```
{'A': 0.3333333333333333, 'B': 0.5, 'C': 0.5, 'D':
0.16666666666666666, 'E': 0.16666666666666666, 'F':
0.16666666666666666, 'G': 0.16666666666666666}
```

$C_D^{*-}(v_i)$

```
{'A': 0.0, 'B': 0.16666666666666666, 'C': 0.16666666666666666,
 'D': 0.16666666666666666, 'E': 0.16666666666666666,
 'F': 0.16666666666666666, 'G': 0.16666666666666666}
```

$C_D^{*+}(v_i)$

```
{'A': 0.3333333333333333, 'B': 0.3333333333333333,
 'C': 0.3333333333333333, 'D': 0.0, 'E': 0.0,
 'F': 0.0, 'G': 0.0}
```

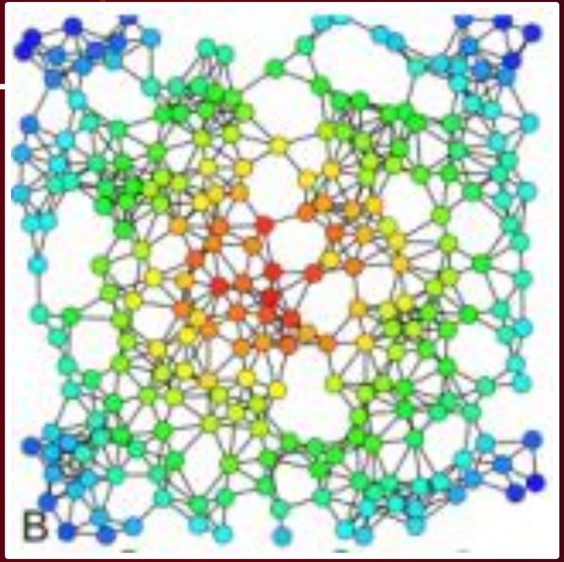
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation



Closeness Centrality

- Measures node proximity to all other nodes in the graph.
- Corresponds to the reciprocal of the average shortest path distance to v_i over all n reachable nodes from v_i .
- High closeness centrality implies efficient reachability to all nodes in the network.

Note: The **Normalized Closeness Centrality** is:

$$C_C^*(v_i) = \frac{n}{|v|-1}\frac{n}{\sum_{u_j\neq v_i} S(v_i,u_i)}$$

Note: For Directed Graphs, same equation. But distances with directed edges only.

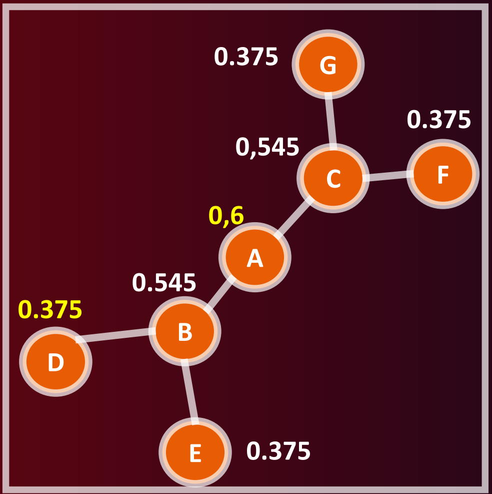
Examples:

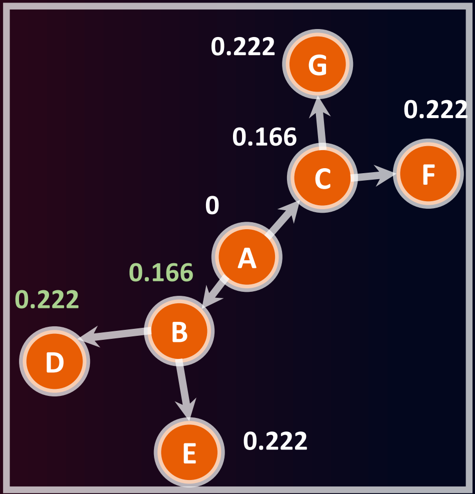
$$C_C^*(A) = \frac{6}{6}\frac{6}{(1+2+2)*2}=0,6$$

$$C_C^*(D) = \frac{6}{6}\frac{6}{(1+2*2+3+4*2)}=0,375$$

$$C_C^*(B) = \frac{1}{6}\frac{1}{(1)}=0,166$$

$$C_C^*(D) = \frac{2}{6}\frac{2}{(1+2)}=0,222$$





DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

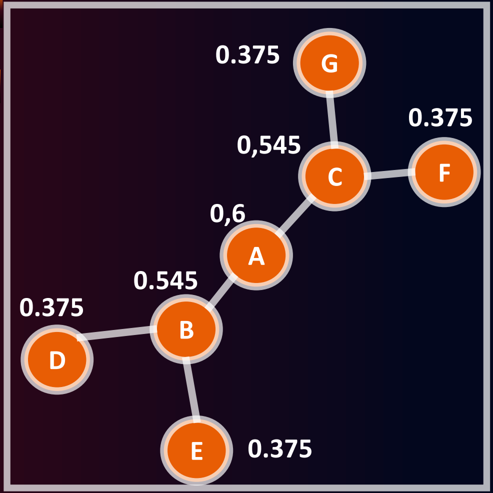
03 Graph Representation

Closeness Centrality for Undirected Graphs

```
print(nx.closeness_centrality(UG))
```

$C_c^*(v_i)$

```
{'A': 0.6,
 'B': 0.5454545454545454,
 'C': 0.5454545454545454,
 'D': 0.375,
 'E': 0.375,
 'F': 0.375,
 'G': 0.375}
```

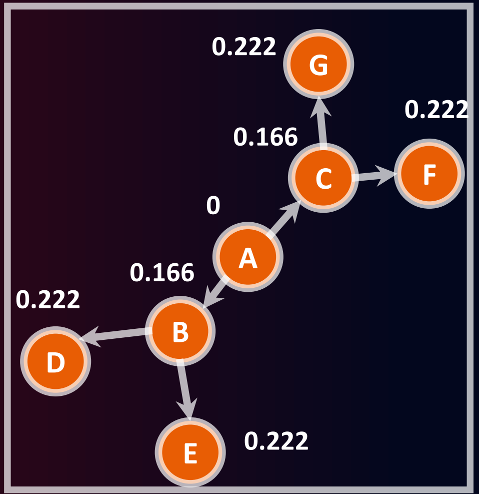


Closeness Centralities for Directed Graphs

```
print(nx.closeness_centrality(DG))
```

$C_c^*(v_i)$

```
{'A': 0.0,
 'B': 0.16666666666666666,
 'C': 0.16666666666666666,
 'D': 0.2222222222222222,
 'E': 0.2222222222222222,
 'F': 0.2222222222222222,
 'G': 0.2222222222222222}
```



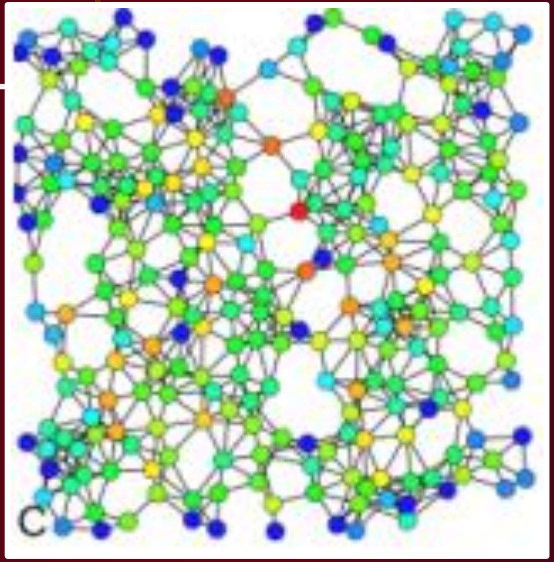
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation



Betweenness Centrality

- Quantifies how often a node lies on shortest paths between other nodes.
- Indicates a node's role as a **bridge** in the network.
- High **betweenness** centrality signifies influence over **information flow** between parts of the graph.

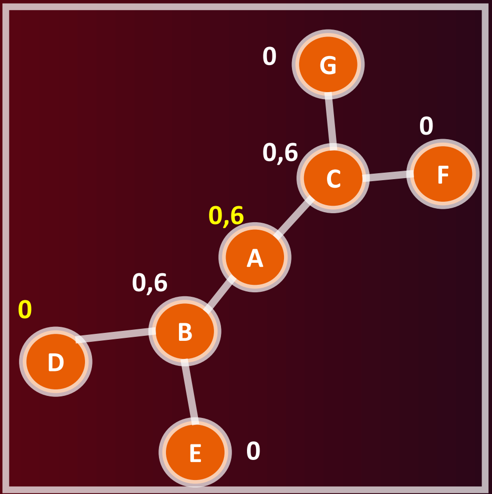
Note: The **Normalized Betweeness Centrality** is: $C_B^*(v_i) = \frac{2}{(|v|-1)(|v|-2)} \sum_{s \neq t \neq v_i} \frac{\sigma_{s,t}(v_i)}{\sigma_{s,t}}$

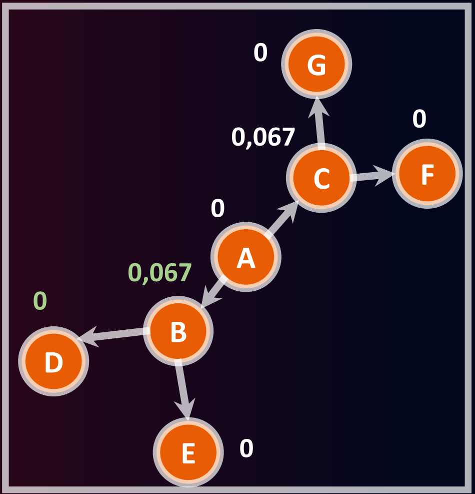
Note: For Directed Graphs, same equation. But with a normalized factor: $\frac{1}{(|v|-1)(|v|-2)}$

Examples:

$$C_B^*(A) = \frac{2}{6*5} * 9 = 0{,}6 \qquad C_B^*(D) = \frac{2}{6*5} * 0 = 0$$

$$C_B^*(B) = \frac{1}{6*5} * 2 = 0{,}067 \qquad C_B^*(D) = \frac{1}{6*5} * 0 = 0$$





DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

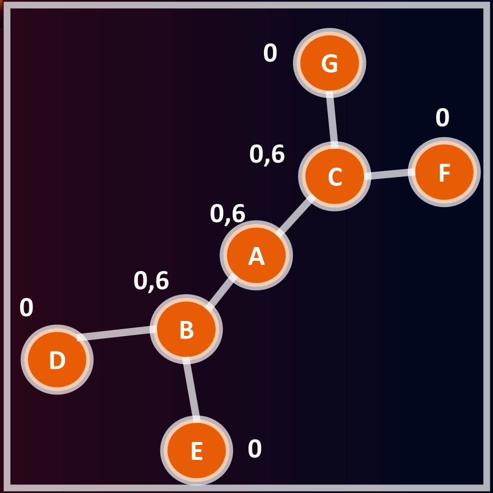
03 Graph Representation

Betweenness Centralities for UnDirected Graphs

```
print(nx.betweenness_centrality(UG))
```

$C_B^*(v_i)$

```
{'A': 0.6,
 'B': 0.6,
 'C': 0.6,
 'D': 0.0,
 'E': 0.0,
 'F': 0.0,
 'G': 0.0}
```

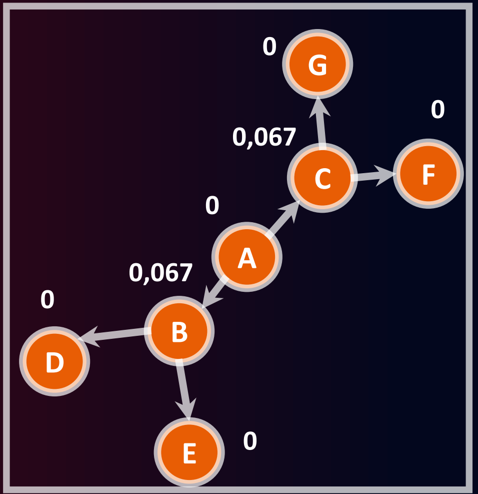


Betweenness Centralities for Directed Graphs

```
print(nx.betweenness_centrality(DG))
```

$C_B^*(v_i)$

```
{'A': 0.0,
 'B': 0.06666666666666667,
 'C': 0.06666666666666667,
 'D': 0.0,
 'E': 0.0,
 'F': 0.0,
 'G': 0.0}
```



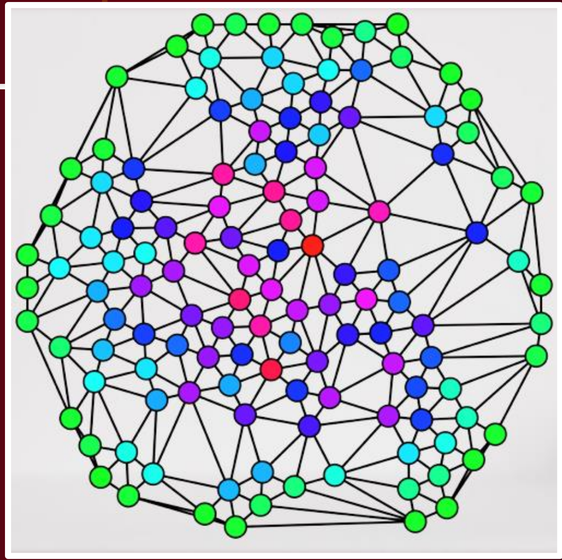
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation



Graph Representation

- **Structured Encoding:** Utilizes mathematical or data structures to represent graph relationships and structure.
- **Effective Processing:** Enables computer algorithms to efficiently work with abstract graph concepts..

Common ways to represent a Graph:

Adjacency Edges

Adjacency Matrix

Adjacency Nodes

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation

Adjacency Edges

Representation Type	• List-based.
Space Complexity	• $O(\|e\|)$, where $\|e\|$ is the number of edges.
Space-Efficient for Sparse Graphs	• Efficient for graphs with significantly fewer edges compared to nodes.
Connectivity Checking	• Requires iterating through the entire list for connectivity checks.

Example

```
edge_list = [('A', 'B'), ('A', 'C'),
             ('B', 'A'), ('B', 'D'), ('B', 'E'),
             ('C', 'A'), ('C', 'F'), ('C', 'G'),
             ('D', 'B'),
             ('E', 'B'),
             ('F', 'C'),
             ('G', 'C')]
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

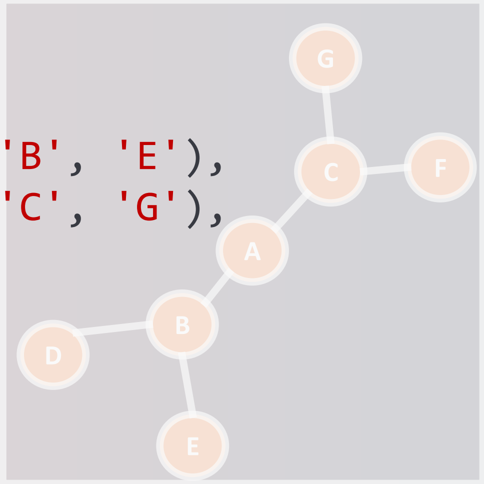
03 Graph Representation

Creating a Graph from Adjacency Edges

```
import networkx as nx
# Given edge list
edge_list = [('A', 'B'), ('A', 'C'),
             ('B', 'A'), ('B', 'D'), ('B', 'E'),
             ('C', 'A'), ('C', 'F'), ('C', 'G'),
             ('D', 'B'),
             ('E', 'B'),
             ('F', 'C'),
             ('G', 'C')]

# Create a graph from the edge list
G = nx.Graph(edge_list)

# Draw the graph
nx.draw(G, with_labels=True)
```



DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation

Adjacency Matrix

Representation Type	• Matrix-based.
Space Complexity	• $O(\|v\|^2)$, where $\|v\|$ is the number of nodes.
Efficient Edge Existence Check	• Constant time operation for edge existence check.
Inefficient for Sparse Graphs	• Space-consuming for sparse graphs.

Example

```
adj = [[0,1,1,0,0,0,0],
       [1,0,0,1,1,0,0],
       [1,0,0,0,0,1,1],
       [0,1,0,0,0,0,0],
       [0,1,0,0,0,0,0],
       [0,0,1,0,0,0,0],
       [0,0,1,0,0,0,0]]
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation

Creating a Graph from Adjacency Matrix

```
import networkx as nx
import numpy as np

# Given adjacency matrix
adj = np.array([[0,1,1,0,0,0,0],
                [1,0,0,1,1,0,0],
                [1,0,0,0,0,1,1],
                [0,1,0,0,0,0,0],
                [0,1,0,0,0,0,0],
                [0,0,1,0,0,0,0],
                [0,0,1,0,0,0,0]])
# Desired node labels
node_labels = {0:'A', 1:'B', 2:'C', 3:'D', 4:'E', 5:'F',
               6:'G'}
# Create a graph from the adjacency matrix
G = nx.from_numpy_matrix(adj)
# Relabel nodes
G = nx.relabel_nodes(G, node_labels)
# Draw the graph with specified labels
nx.draw(G, with_labels=True)
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation

Adjacency Nodes

Representation Type	• List-based.
Space Complexity	• $O(\|v\| + \|e\|)$, $\|v\|$: the number of nodes, $\|e\|$: the number of edges.
Efficient Iteration	• Allows efficient iteration through adjacent vertices.
Efficient Additions:	• Adding a node or an edge can be done in constant time.

Example

```
adj_list = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B'],
    'E': ['B'],
    'F': ['C'],
    'G': ['C']
}
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Discovering Graph Concepts

01 Degrees & Neighbors

02 Graph Measures

03 Graph Representation

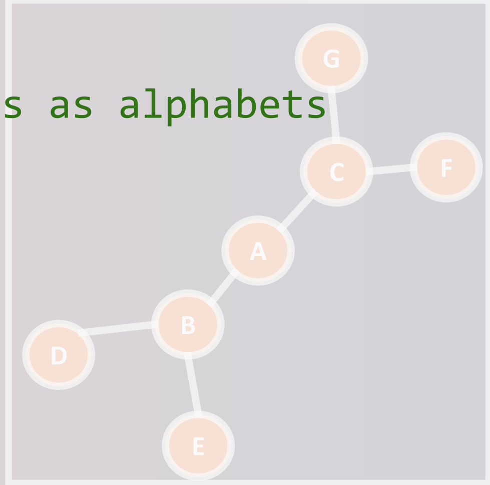
Creating a Graph from Adjacency Nodes

```python
import networkx as nx

# Given adjacency list with node labels as alphabets
adj_nodes = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B'],
    'E': ['B'],
    'F': ['C'],
    'G': ['C']
}

# Create a graph from the adjacency nodes
G = nx.Graph(adj_nodes)

# Print the nodes and edges
print("Nodes:", G.nodes())
print("Edges:", G.edges())
nx.draw(G, with_labels= True)
```



DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Exploring Graph Algorithms

- **Importance of Graph Algorithms**: Essential for solving graph-related problems like **shortest paths** and **cycle detection**.
- **Graph Traversal Algorithms:** Focus on algorithms:

01 BFS (Breadth-First Search)

02 DFS (Depth-First Search))

Importance of BFS & DFS in GNNs

Pre-processing and Graph Construction

- Can be used to explore and preprocess the graph structure, organizing it into a suitable format for GNNs

Node Embedding Initialization

- Can initialize node embeddings by traversing the graph, capturing initial representations based on the traversal order.

Negative Sampling

- Can help in generating negative samples for training GNNs, enhancing the model's ability to distinguish between positive and negative connections.

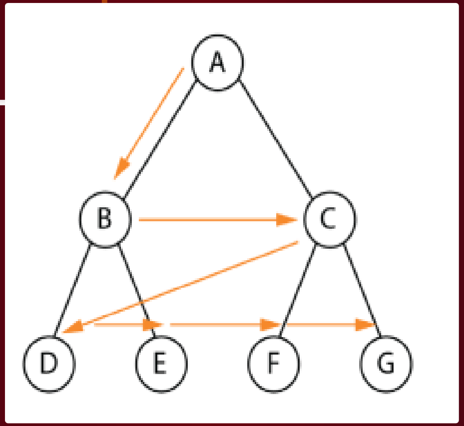
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Exploring Graph Algorithms

- **Importance of Graph Algorithms**: Essential for solving graph-related problems like **shortest paths** and **cycle detection**.
- **Graph Traversal Algorithms:** Focus on algorithms:

01 BFS (Breadth-First Search)

02 DFS (Depth-First Search))



BFS Overview

- A popular algorithm for traversing and searching unweighted graphs.

Key Points of BFS Algorithm

Traversal Algorithm	• Queue-based traversal.
Exploration Strategy	• Explores all neighbors at the current level before moving to the next level.
Applications	• Shortest paths, connectivity, web crawling, social networks, and routing.
Cycle Detection	• Does not efficiently detect cycles.

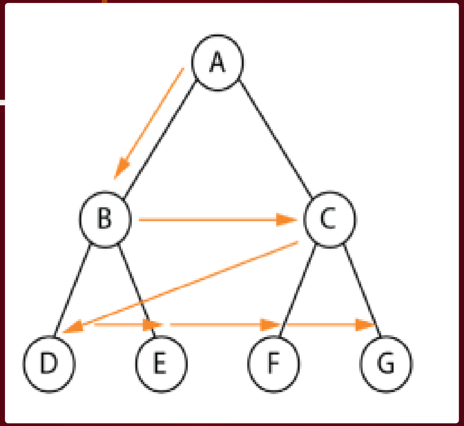
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Exploring Graph Algorithms

- **Importance of Graph Algorithms**: Essential for solving graph-related problems like **shortest paths** and **cycle detection**.
- **Graph Traversal Algorithms:** Focus on algorithms:

01 BFS (Breadth-First Search)

02 DFS (Depth-First Search)



BFS Overview

- A popular algorithm for traversing and searching unweighted graphs

Key Points of BFS Algorithm (Continued)

Time Complexity	• $O(\lvert v\rvert + \lvert e\rvert)$, $\lvert v\rvert$: the number of nodes, $\lvert e\rvert$: the number of edges.
Memory Usage	• Requires more memory due to the queue.
Path Solution	• Provides the shortest path.
Loop Trapping	• Does not lead to infinite loops.

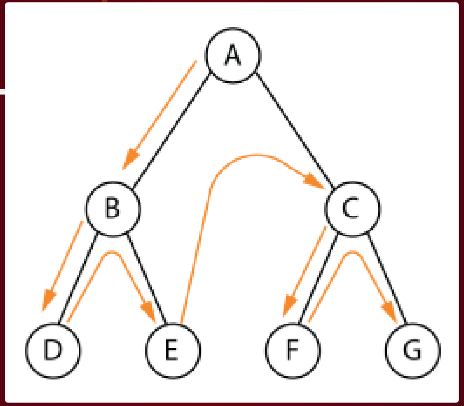
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Exploring Graph Algorithms

- **Importance of Graph Algorithms**: Essential for solving graph-related problems like **shortest paths** and **cycle detection**.
- **Graph Traversal Algorithms:** Focus on algorithms:

01 BFS (Breadth-First Search)

02 DFS (Depth-First Search)



DFS Overview

- A popular algorithm for traversing and searching through a graph or tree data structure.

Key Points of BFS Algorithm

Traversal Algorithm	• Recursive traversal.
Exploration Strategy	• Explores as far as possible along each branch before backtracking.
Applications	• Connected components, topological sorting, mazes, cycles.
Cycle Detection	• Efficiently detects cycles as it traverses the graph in a depth-first order.

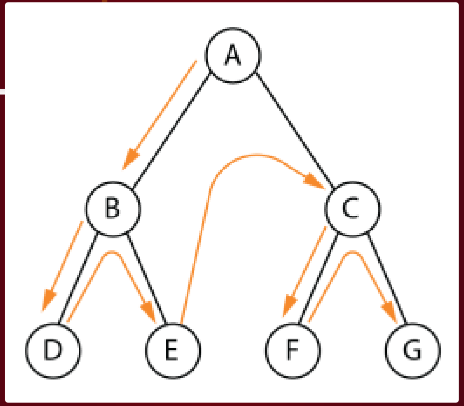
DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Exploring Graph Algorithms

- **Importance of Graph Algorithms**: Essential for solving graph-related problems like **shortest paths** and **cycle detection**.
- **Graph Traversal Algorithms:** Focus on algorithms:

01 BFS (Breadth-First Search)

02 DFS (Depth-First Search)



DFS Overview

- A popular algorithm for traversing and searching through a graph or tree data structure.

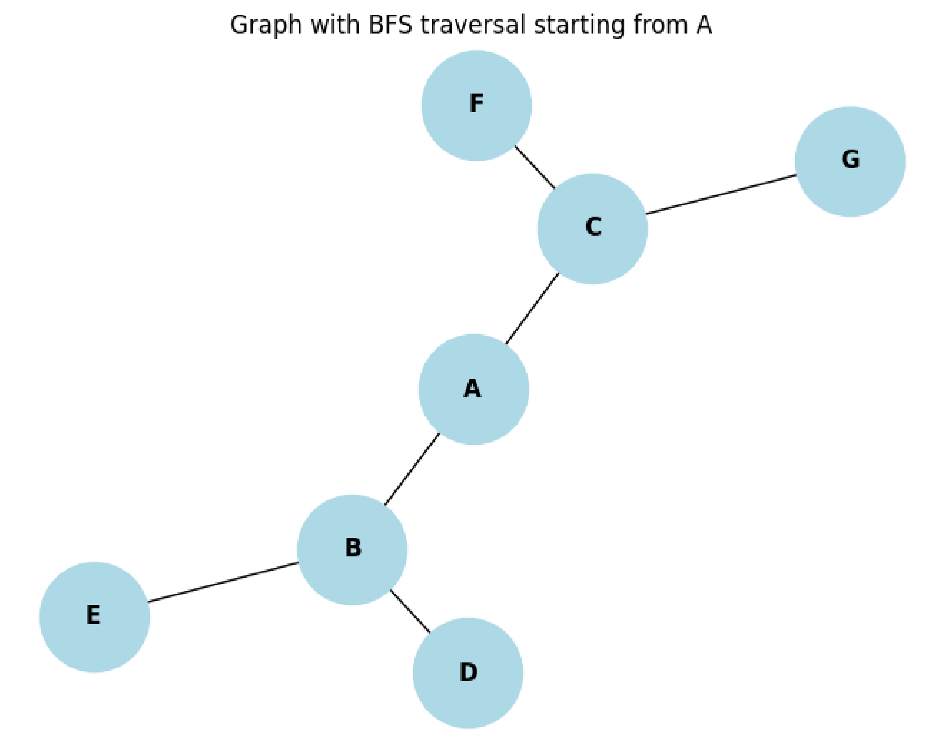
Key Points of DFS Algorithm (Continued)

Time Complexity	• $O(\|v\| + \|e\|)$, $\|v\|$: the number of nodes, $\|e\|$: the number of edges.
Memory Usage	• Requires less memory.
Path Solution	• Does not guarantee the shortest path.
Loop Trapping	• Can lead to infinite loops.

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Exploring Graph Algorithms

01 BFS (Breadth-First Search)



BFS with NetworkX

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

# Create a graph
G = nx.Graph()
G.add_edges_from(edge_nodes)

# Perform BFS starting from node 'A'
bfs_result = list(nx.bfs_edges(G, source='A'))

# Print the BFS result
print("BFS Result:", bfs_result)

# Draw the graph
nx.draw(G, with_labels=True, node_color='lightblue',
node_size=3000, font_size=12, font_weight='bold')
plt.title('Graph with BFS traversal starting from A')
plt.show()
```

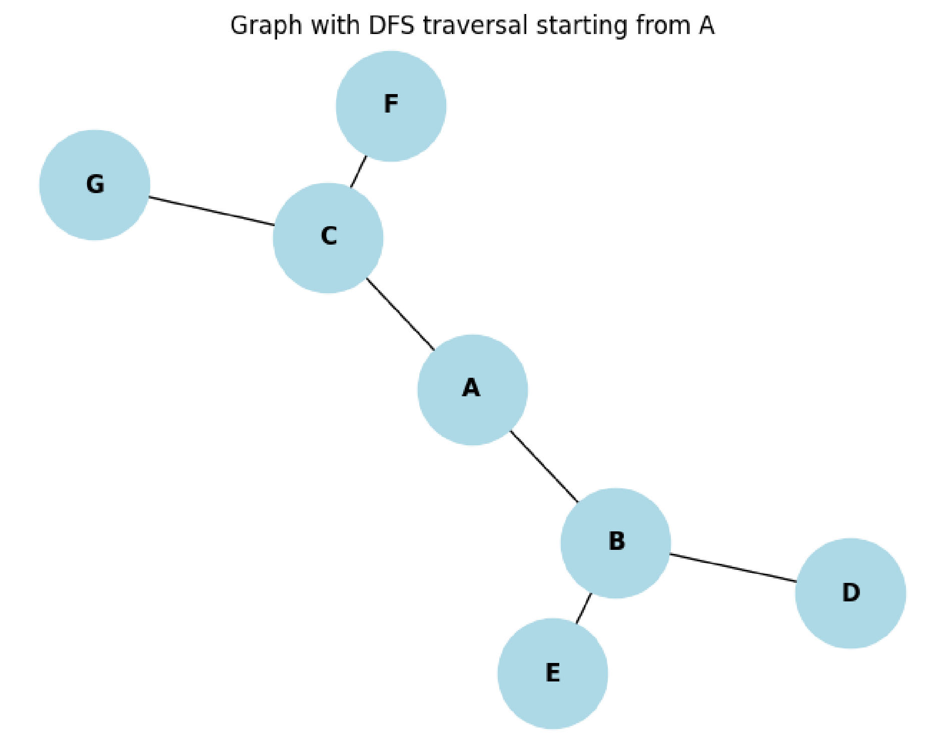
```
BFS Result: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B',
'E'), ('C', 'F'), ('C', 'G')]
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

Exploring Graph Algorithms

DFS with NetworkX

02 DFS (Depth-First Search)




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

# Create a graph
G = nx.Graph()
G.add_edges_from(edge_nodes)

# Perform DFS starting from node 'A'
dfs_result = list(nx.dfs_edges(G, source='A'))

# Print the DFS result
print("DFS Result:", dfs_result)

# Draw the graph
nx.draw(G, with_labels=True, node_color='lightblue',
node_size=3000, font_size=12, font_weight='bold')
plt.title('Graph with DFS traversal starting from A')
plt.show()
```

```
DFS Result: [('A', 'B'), ('B', 'D'), ('B', 'E'), ('A',
'C'), ('C', 'F'), ('C', 'G')]
```

DR. B. KHALDI
Assoc. Prof.
email: b.khaldi@esi-sba.dz

THANK YOU