Lecture 14

Graphs I

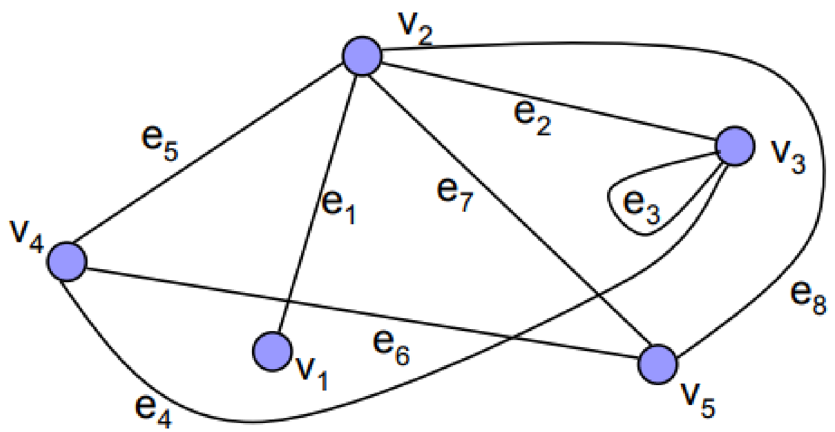
Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

Graphs

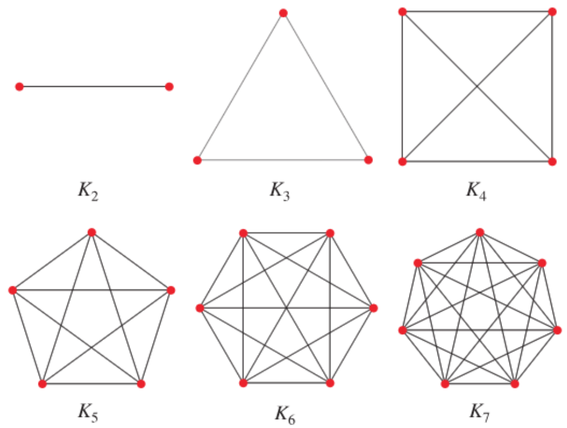
- A graph is a diagram consisting of a set of points/vertices and edges connecting them.
- Could be defined as an ordered triple (V,E, ψ)
  - V: List of vertices
  - E: List of edges
  - Ψ: Incidence function

V = {v1, v2, v3, v4, v5}
E = {e1, e2, e3, e4, e5, e6, e7, e8}
ψG(e1) = v1v2,
ψG(e2) = v2v3,
ψG(e3) = v3v3
ψG(e4) = v3v4,
ψG(e5) = v2v4,
ψG(e6) = v4v5,
ψG(e7) = v2v5,
ψG(e8) = v2v5



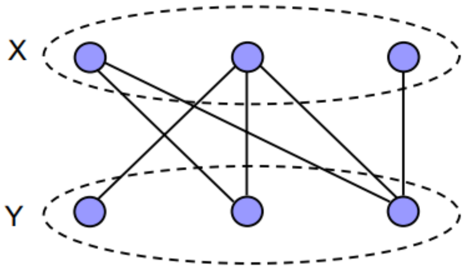
Graphs Definitions

- **Complete Graph:** Each pair of vertices are connected.



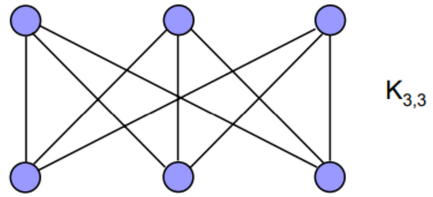
https://mathworld.wolfram.com/images/eps-svg/CompleteGraphs_801.svg

- **Bipartite graph:** A graph which could be partitioned into two sets, where there are no inner edge connection in each set

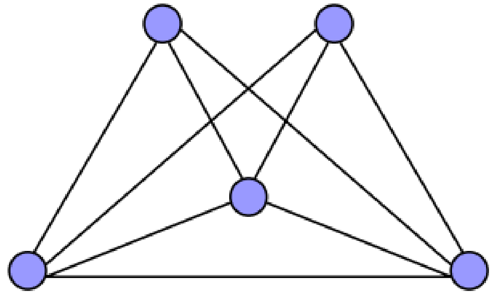


Empty Graph: A graph with no edges.

Complete Bipartite Graph: Each couple from the partitioned sets are connected.



Planar Graph: Graphs that could be represented in 2D without any edge intersections.

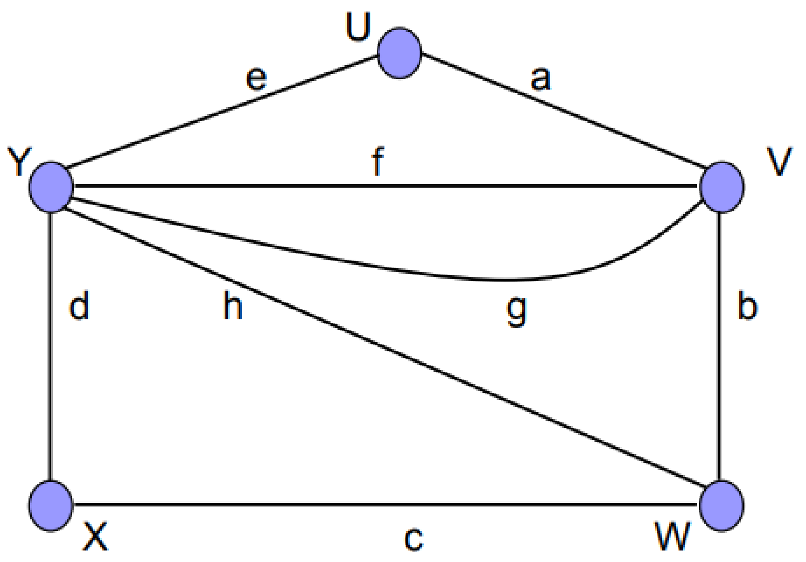


Subgraph of G: A graph containing selected edges and vertices from G.

Spanning subgraph of G: A subgraph containing all vertices from G.

Paths, Walks, Trails

- **Walk:** A finite sequence of connected edges and vertices.
- **Trail:** A walk where the edges are distinct.
- **Path:** A trail if the vertices are distints.



walk: UaVfYfVgYhWbV
trail: WcXdYhWbVgY
path: XcWhYeUaV

- Two vertices u and v in a graph are considered connected if there exists a path from u to v.

Graph structure

```c
typedef struct Node {
    int id;
    struct Node** adj;
    int capacity;
    int numAdj;
} Node;
```

```c
typedef struct Graph {
    Node** nodes;
    int numNodes;
    int capacity;
} Graph;
```

```c
Graph* createGraph() {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->nodes = NULL;
    graph->numNodes = 0;
    graph->capacity = 0;
    return graph;
}
```

```c
Node* createNode(int id) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->id = id;
    newNode->adj = NULL;
    newNode->capacity = 0;
    newNode->numAdj = 0;
    return newNode;
}
```

```c
void addEdgeToNode(Node* node, Node* adjacentNode) {
    if (node->numAdj == node->capacity) {
        int newCapacity = node->capacity == 0 ? 2 : node->capacity * 2;
        Node** newAdj = (Node**)malloc(newCapacity * sizeof(Node*));

        for (int i = 0; i < node->numAdj; i++) {
            newAdj[i] = node->adj[i];
        }

        free(node->adj);
        node->adj = newAdj;
        node->capacity = newCapacity;
    }
    node->adj[node->numAdj++] = adjacentNode;
}
```

Graph structure

```c
void addNodeToGraph(Graph* graph, int id) {
    if (graph->numNodes == graph->capacity) {
        int newCapacity = graph->capacity == 0 ? 4 : graph->capacity * 2;
        Node** newNodes = (Node**)malloc(newCapacity * sizeof(Node*));

        for (int i = 0; i < graph->numNodes; i++) {
            newNodes[i] = graph->nodes[i];
        }

        free(graph->nodes);
        graph->nodes = newNodes;
        graph->capacity = newCapacity;
    }

    graph->nodes[graph->numNodes] = createNode(id);
    graph->numNodes++;
}
```

```c
void addEdgeToGraph(Graph* graph, int u, int v) {
    if (u < graph->numNodes && v < graph->numNodes && u != v) {
        addEdgeToNode(graph->nodes[u], graph->nodes[v]);
    }
}
```

```c
void printGraph(Graph* graph) {
    for (int i = 0; i < graph->numNodes; i++) {
        printf("Node %d has edges to: ", graph->nodes[i]->id);
        for (int j = 0; j < graph->nodes[i]->numAdj; j++) {
            printf("%d ", graph->nodes[i]->adj[j]->id);
        }
        printf("\n");
    }
}
```

```c
void freeNode(Node* node) {
    if (node->adj != NULL) {
        free(node->adj);
    }
    free(node);
}
```

```c
void freeGraph(Graph* graph) {
    for (int i = 0; i < graph->numNodes; i++) {
        freeNode(graph->nodes[i]);
    }
    free(graph->nodes);
    free(graph);
}
```

Graph structure

```
Graph* g = createGraph();

addNodeToGraph(g, 0);
addNodeToGraph(g, 1);
addNodeToGraph(g, 2);
addNodeToGraph(g, 3);

addEdgeToGraph(g, 0, 1);
addEdgeToGraph(g, 0, 2);
addEdgeToGraph(g, 1, 2);
addEdgeToGraph(g, 2, 3);
addEdgeToGraph(g, 3, 0);

printGraph(g);
freeGraph(g);
```

```
Node 0 has edges to: 1 2
Node 1 has edges to: 2
Node 2 has edges to: 3
Node 3 has edges to: 0
```

It is cheaper to store the graph in a 2D matrix.

- Adjacency matrix

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

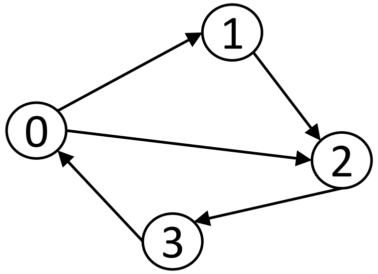
Adjacency Matrix

It is cheaper to store the graph in a 2D matrix.

- Adjacency matrix

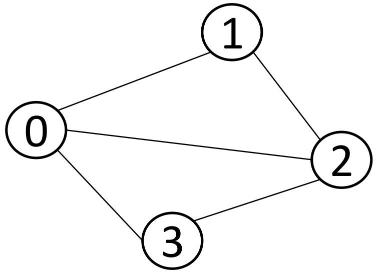
Directed Graph

	0	**1**	**2**	**3**
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	1	0	0	0



Undirected Graph

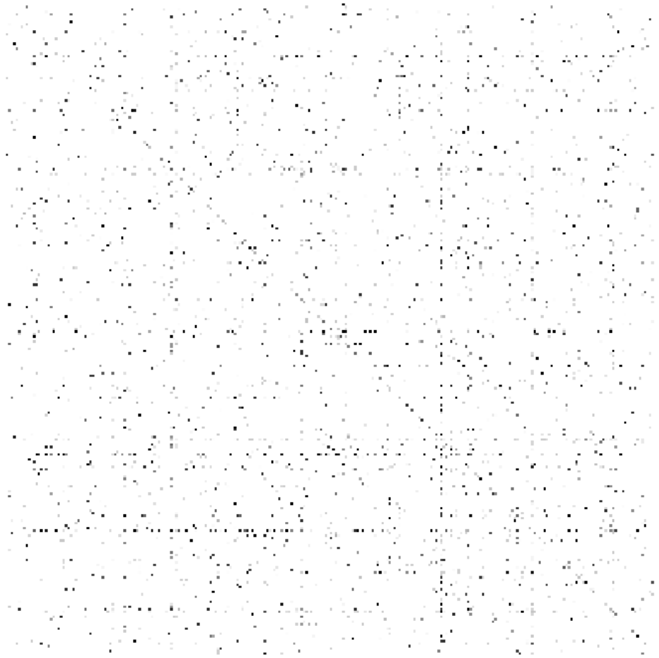
	0	**1**	**2**	**3**
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0



A degree of a vertex $v_i$is $\sum_{j=0}^{n-1} adj[i][j]$.

Rows: Indegree
Columns: Outdegree

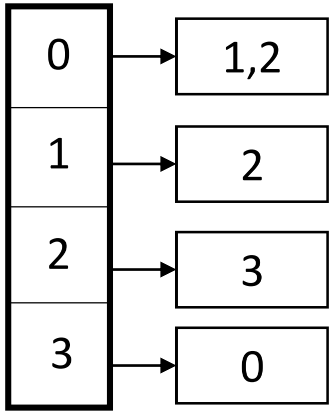
Adjacency matrices are generally sparse.

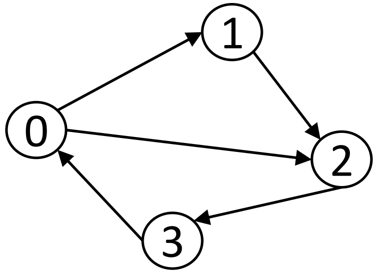


Adjacency List

The same structure could be represented using linked lists.

	0	**1**	**2**	**3**
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	1	0	0	0

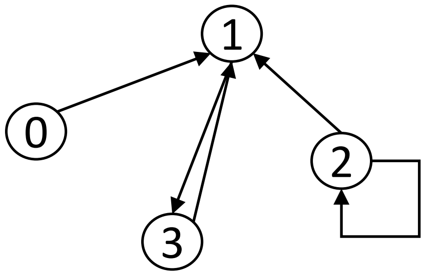




Connection Matrix

- K[th] matrix product of the adjacency matrix: A^k
- If the graph has n vertices, then the number of walks of length < n can be found as: $A^0 + A^1 + \cdots + A^{n-1}$

- The connection matrix C shows whether some vertices are connected.



$$A^0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad A^1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad A^3 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Node and Edge Attributes

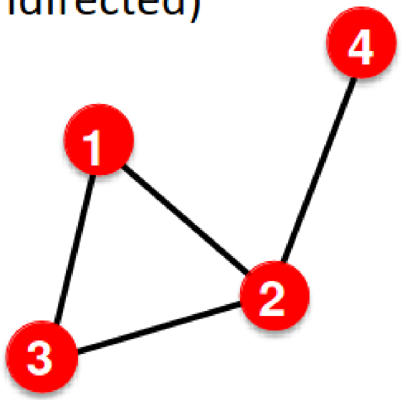
Possible options:

- Weight (*e.g.*, frequency of communication)
- Ranking (best friend, second best friend…)
- Type (friend, relative, co-worker)
- Sign: Friend vs. Foe, Trust vs. Distrust
- Properties depending on the structure of the rest of the graph: Number of common friends

Graph Weights

- **Unweighted**

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

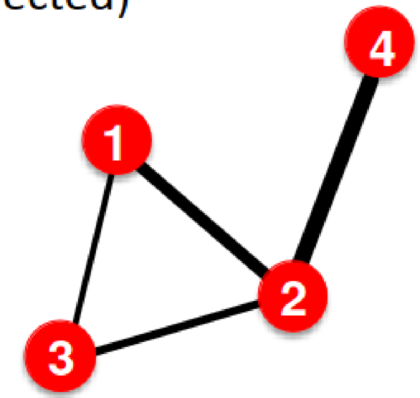
$$A_{ii} = 0 \qquad A_{ij} = A_{ji}$$

$$E = \frac{1}{2}\sum_{i,j=1}^{N} A_{ij} \qquad \bar{k} = \frac{2E}{N}$$

Examples: Friendship, Hyperlink

- **Weighted**

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 2 & 0.5 & 0 \\ 2 & 0 & 1 & 4 \\ 0.5 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 \end{pmatrix}$$

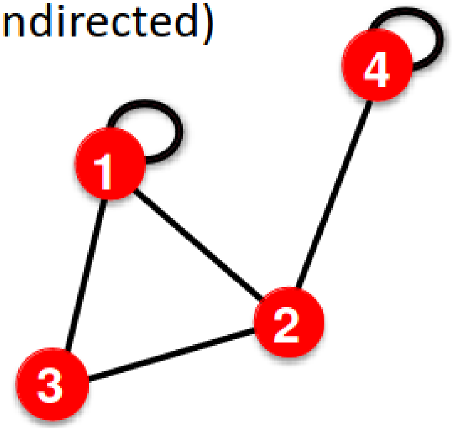
$$A_{ii} = 0 \qquad A_{ij} = A_{ji}$$

$$E = \frac{1}{2}\sum_{i,j=1}^{N} nonzero(A_{ij}) \qquad \bar{k} = \frac{2E}{N}$$

Examples: Collaboration, Internet, Roads

Graph Edges

- **Self-edges (self-loops)**
  (undirected)



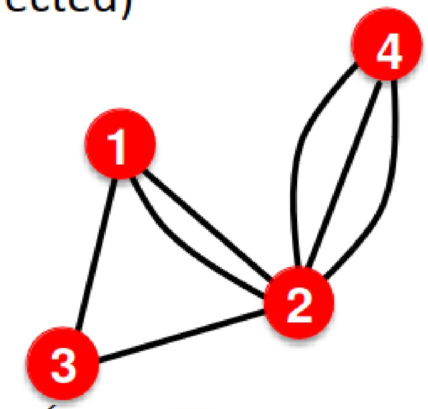
$$A_{ij} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$A_{ii} \neq 0 \qquad A_{ij} = A_{ji}$$

$$E = \frac{1}{2} \sum_{i,j=1, i \neq j}^{N} A_{ij} + \sum_{i=1}^{N} A_{ii}$$

Examples: Proteins, Hyperlinks

- **Multigraph**
  (undirected)



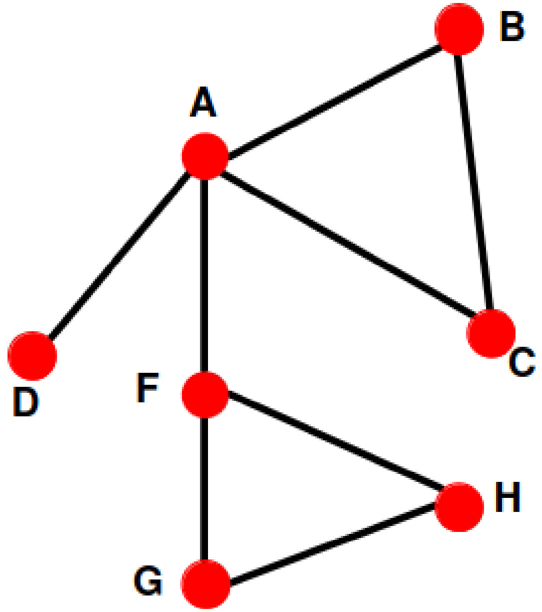
$$A_{ij} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 1 & 3 \\ 1 & 1 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix}$$

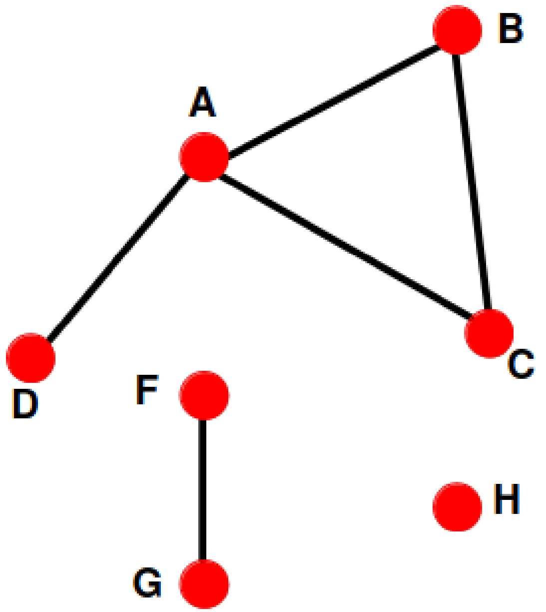
$$A_{ii} = 0 \qquad A_{ij} = A_{ji}$$

$$E = \frac{1}{2} \sum_{i,j=1}^{N} nonzero(A_{ij}) \qquad \bar{k} = \frac{2E}{N}$$

Examples: Communication, Collaboration

- **Connected (undirected) graph:**
  - Any two vertices can be joined by a path
- A disconnected graph is made up by two or more connected components

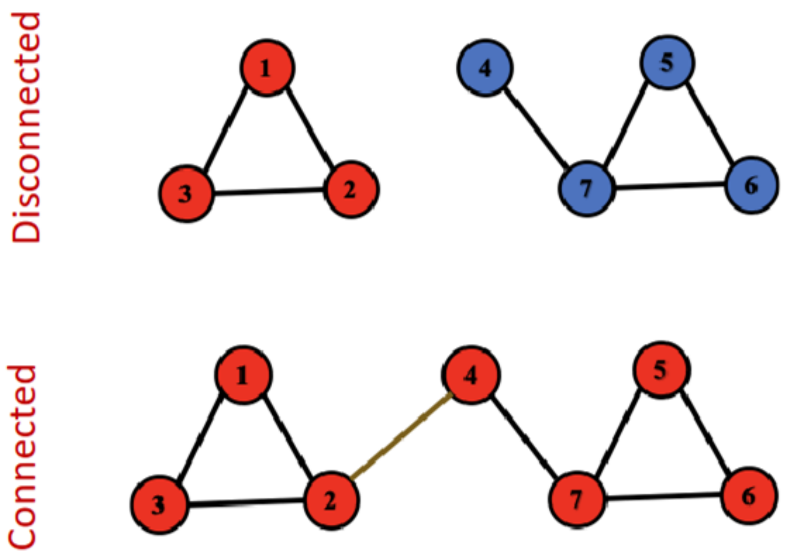




Largest Component:
Giant Component

Isolated node (node H)

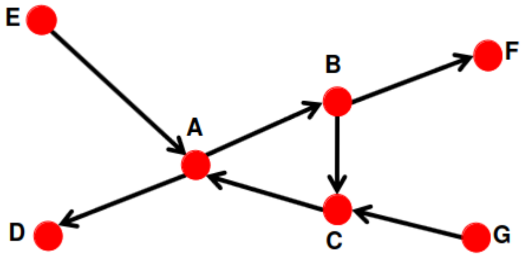
Connectivity



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

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

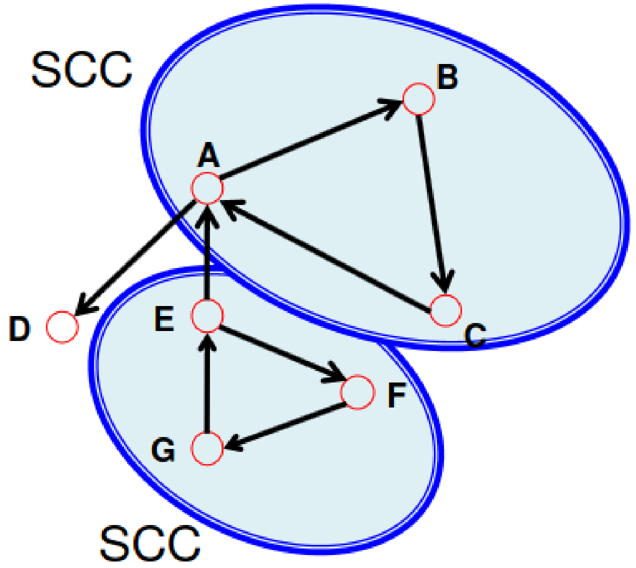
- **Strongly connected directed graph**
  - has a path from each node to every other node and vice versa (e.g., A-B path and B-A path)
- **Weakly connected directed graph**
  - is connected if we disregard the edge directions

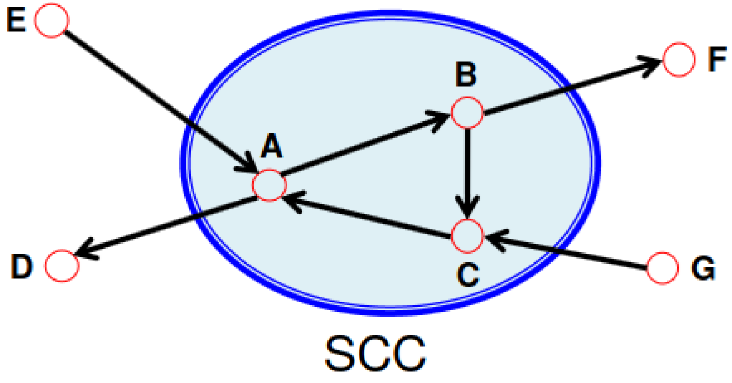


Graph on the left is connected but not strongly connected (e.g., there is no way to get from F to G by following the edge directions).

Connectivity

- **Strongly connected components (SCCs)** can be identified, but not every node is part of a nontrivial strongly connected component.



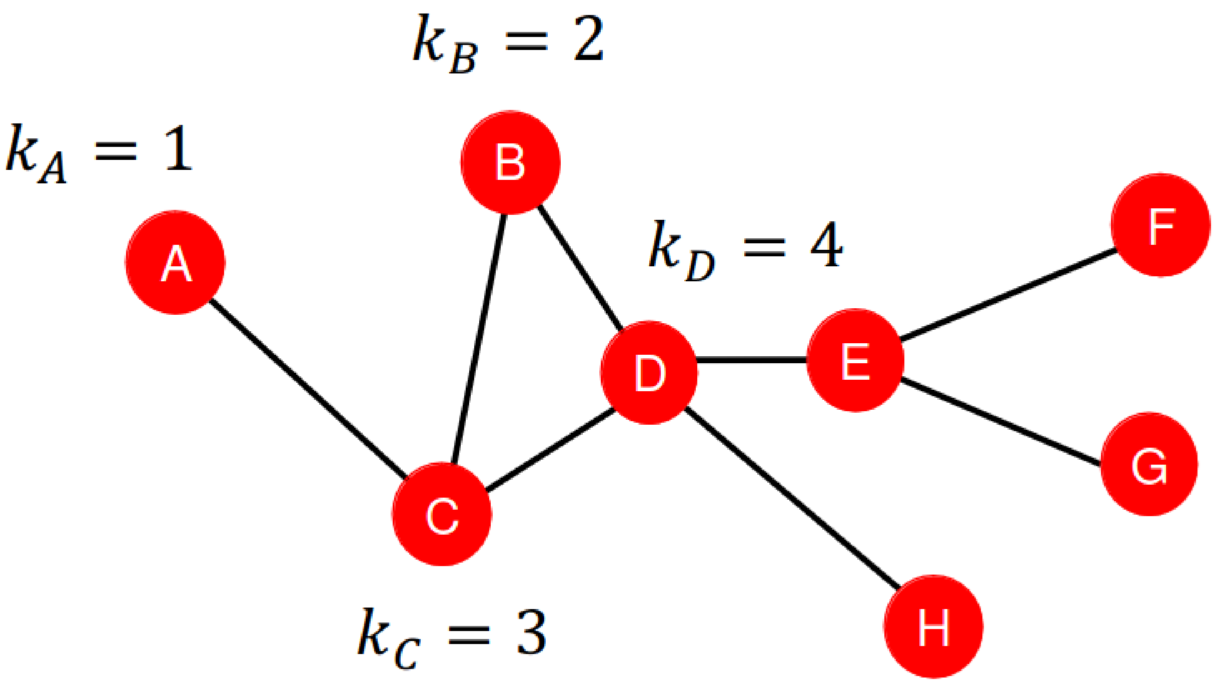


In-component: nodes that can reach the SCC,

Out-component: nodes that can be reached from the SCC.

Node-level Features

- The degree k_v of node v is the number of edges (neighboring nodes) the node has.
- Treats all neighboring nodes equally.



Node-level Features

- Node degree counts the neighboring nodes without capturing their importance.
- Node centrality c_v takes the node importance in a graph into account
- **Different ways to model importance:**
  - Eigenvector centrality
  - Betweenness centrality
  - Closeness centrality
  - and many others…

Node centrality

- **Eigenvector centrality:**
  - A node v is important if surrounded by important neighboring nodes $u \in N(v)$.
  - We model the centrality of node v as the sum of the centrality of neighboring nodes:

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

λ is normalization constant (it will turn out to be the largest eigenvalue of A)

  - Notice that the above equation models centrality in a recursive manner. **How do we solve it?**

Eigenvector centrality:

- Rewrite the recursive equation in the matrix form.

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u \quad \longleftrightarrow \quad \lambda \boldsymbol{c} = \boldsymbol{A}\boldsymbol{c}$$

λ is normalization const (largest eigenvalue of A)

- $\boldsymbol{A}$: Adjacency matrix
  $\boldsymbol{A}_{uv} = 1$ if $u \in N(v)$
- $\boldsymbol{c}$: Centrality vector
- λ: Eigenvalue

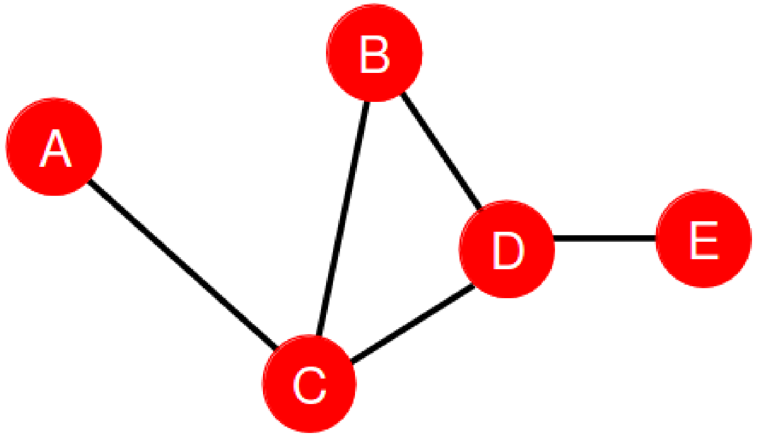
- We see that centrality c is the **eigenvector of $\boldsymbol{A}$**!
- The largest eigenvalue λ_{max} is always positive and unique (by Perron-Frobenius Theorem).
- The eigenvector $\boldsymbol{c}_{max}$ corresponding to λ_{max} is used for centrality.

Node centrality

- **Betweenness centrality:**
  - A node is important if it lies on many shortest paths between other nodes.

$$c_v = \sum_{s \neq v \neq t} \frac{\#(\text{shortest paths betwen } s \text{ and } t \text{ that contain } v)}{\#(\text{shortest paths between } s \text{ and } t)}$$

  - **Example:**



$$c_A = c_B = c_E = 0$$
$$c_C = 3$$
(A-<u>C</u>-B, A-<u>C</u>-D, A-<u>C</u>-D-E)

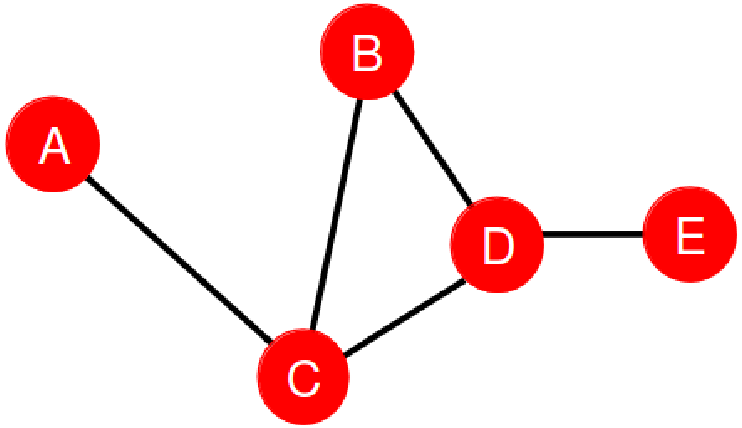
$$c_D = 3$$
(A-C-<u>D</u>-E, B-<u>D</u>-E, C-<u>D</u>-E)

Node centrality

- **Closeness centrality:**
  - A node is important if it has small shortest path lengths to all other nodes.

$$c_v = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$$

  - **Example:**



$c_A = 1/(2+1+2+3) = 1/8$
(A-C-B, A-C, A-C-D, A-C-D-E)

$c_D = 1/(2+1+1+1) = 1/5$
(D-C-A, D-B, D-C, D-E)

Clustering Coefficient

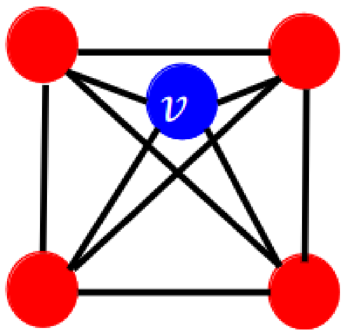
- Measures how connected $v's$ neighboring nodes are:

$$e_v = \frac{\#(\text{edges among neighboring nodes})}{\boxed{\binom{k_v}{2}}} \in [0,1]$$

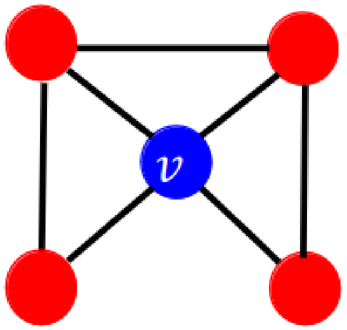
#(node pairs among k_v neighboring nodes)

In our examples below the denominator is 6 (4 choose 2).

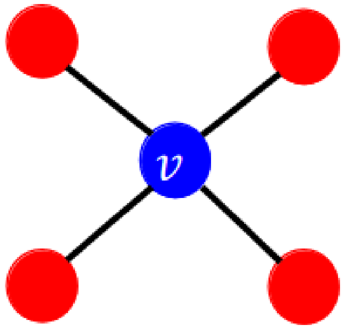
- **Examples:**



$e_v = 1$



$e_v = 0.5$



$e_v = 0$