Graph Basic Concepts*

Yannis Kotidis

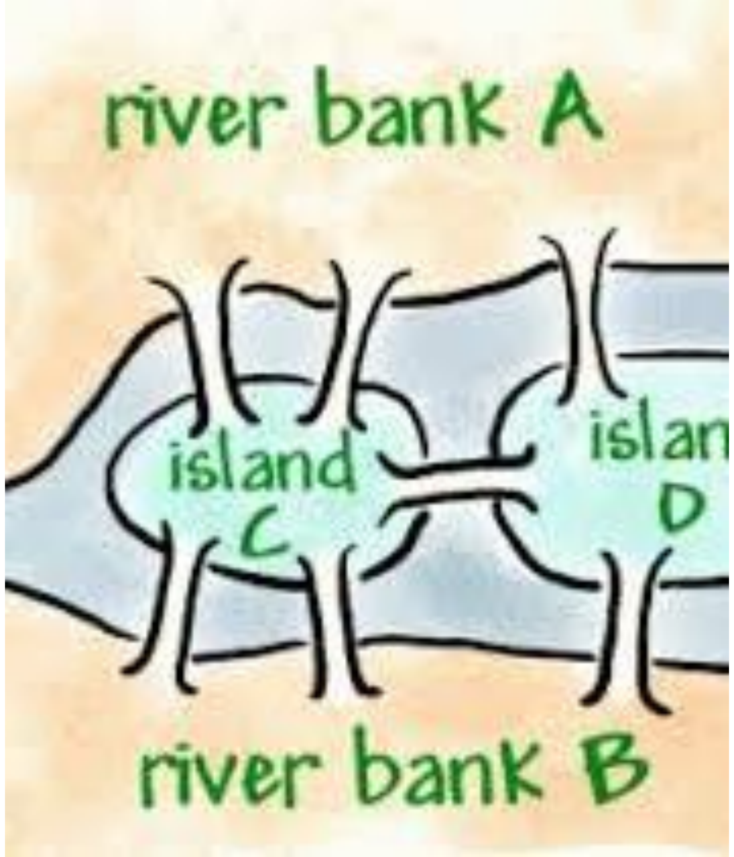
*slides adapted from I. Filippidou’s original presentation

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Outline

- History of Graphs
- Graph Definitions
- Graph Representations
- Graph Topology Metrics
- Walks, Trails and Paths
- Shortest Paths
- Centrality Metrics

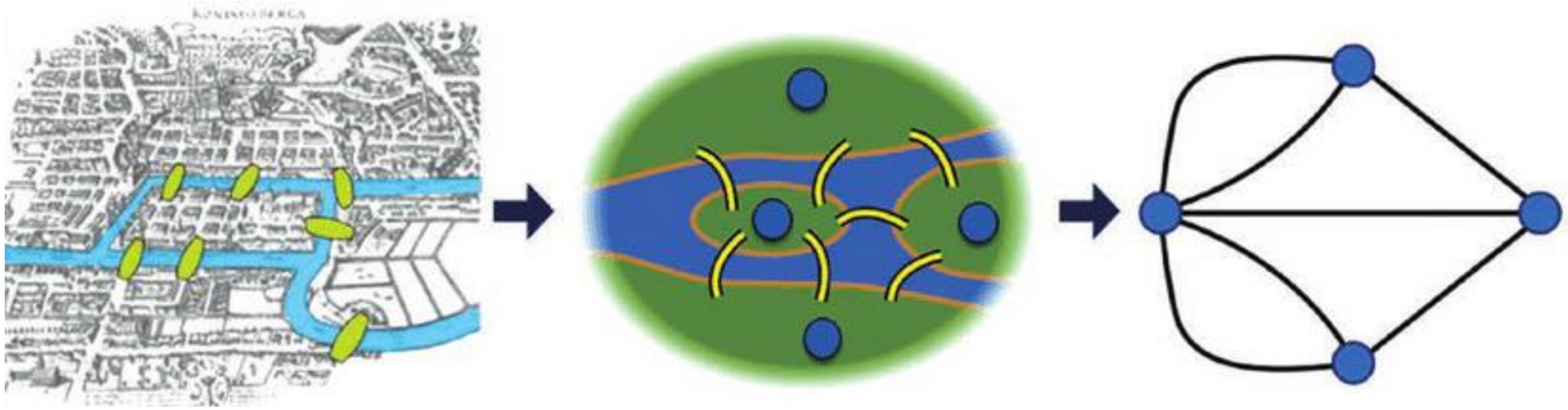
Seven Bridges of Königsberg (solved by Leonhard Euler in 1736)



- Königsberg (now Kaliningrad) is a city on the Pregel river in Prussia
- The city occupied two islands plus areas in both river banks
- **Problem:** ***Walk through all parts of the city and cross each bridge only once?***

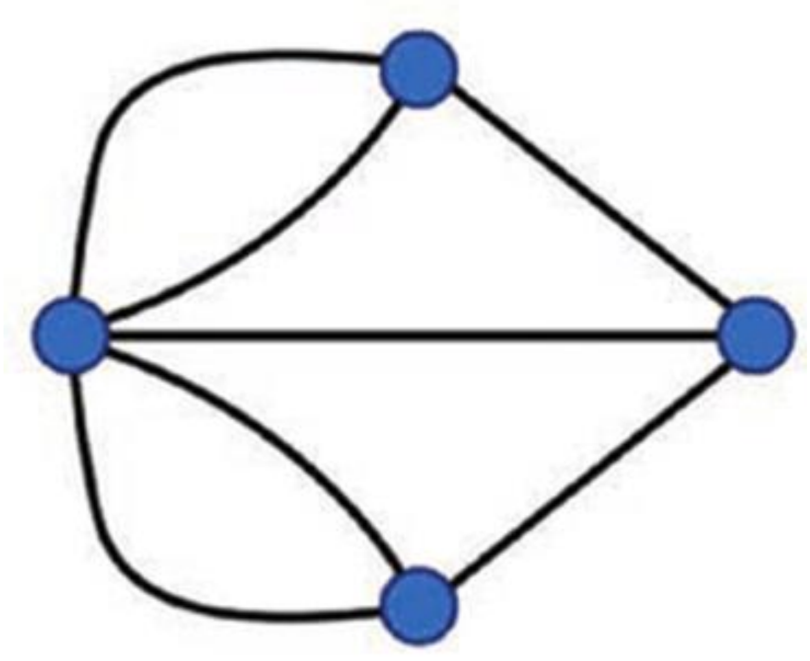
Euler’s idea

- Path inside each land mass is irrelevant.
  - The only important feature is the sequence of bridges crossed.
- Thus, remove all features from consideration except the list of land masses and the bridges connecting them.
- Abstraction: model your input as:
  - **Vertices**: island, river bank
  - **Edges**: bridge



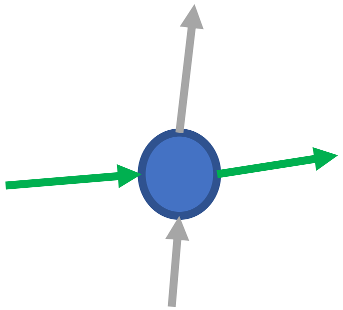
Graph abstraction

- Map city to graph elements
  - City parts → graph nodes
  - City bridges → graph edges
- Cross every edge (bridge) exactly once in a walk?
- Observation: sequence of bridges crossed is important to solving this problem



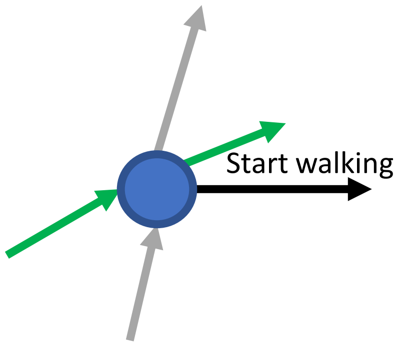
Key observation 1

- Intermediate nodes in the route need an even number of edges (bridges)
  - Because you arrive and leave from these parts of the city

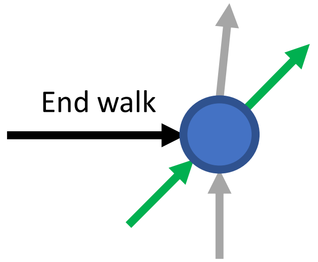


Key observation 2

- Assume **start** and end **nodes** are **different** (otherwise previous observation holds for these nodes as well)
- Start node must have an odd number of bridges
  - Otherwise you will get stuck in that part of the city if you ever visit it again



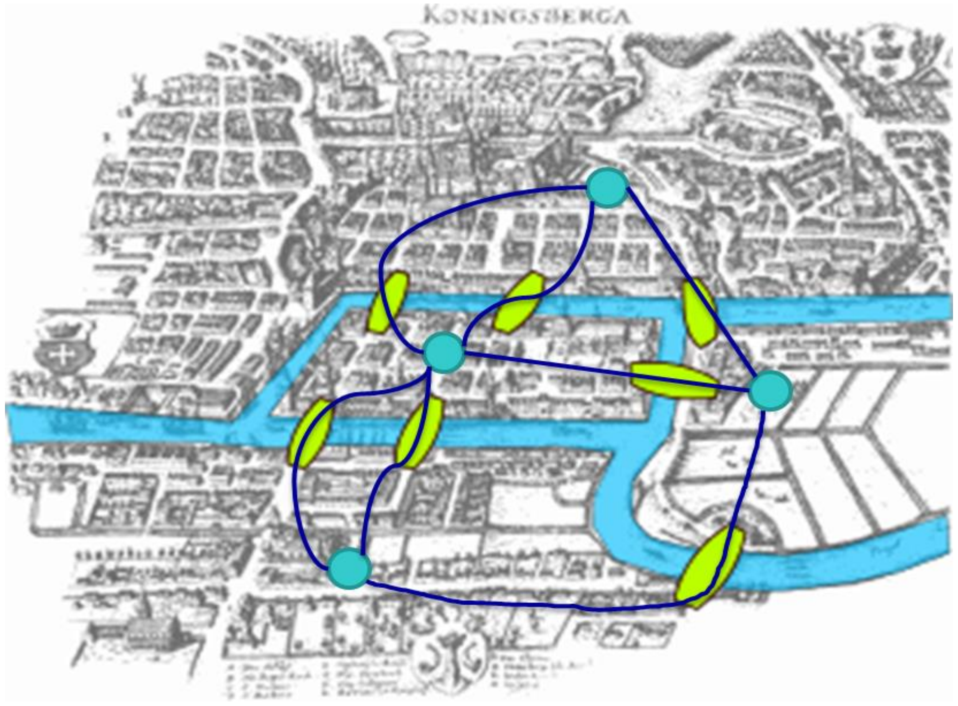
- Same argument for ending node:



Seven Bridges of Konigsberg

Euler's Conjecture:

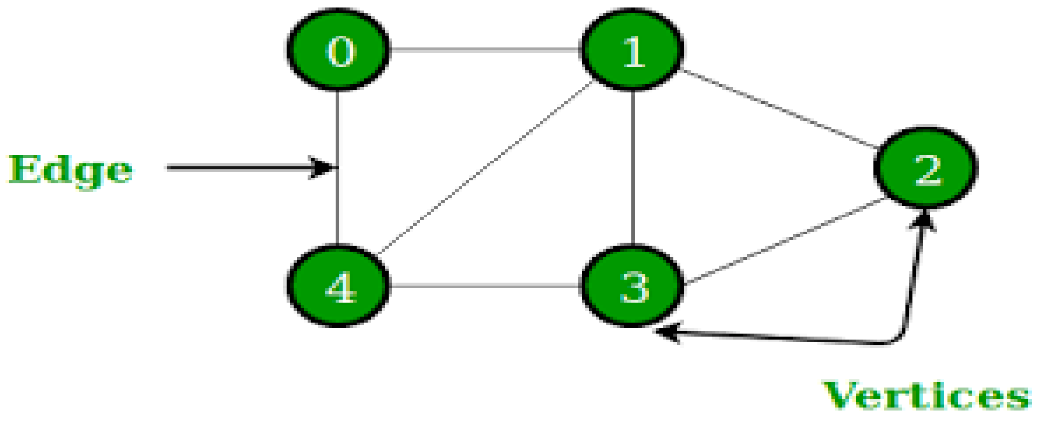
- Graph nodes must have even number of edges
- There can be zero or two nodes with odd number of edges
- All parts of the city have odd number of bridges connecting them with the rest of the city
- **Thus, no Eulerian trail exists**



Graph Definitions

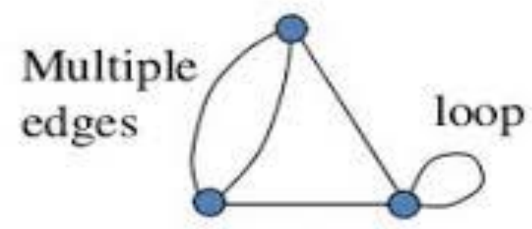
What is a Graph

- An undirected graph G is defined as G = (V, E)
- **V** is a set of all **vertices** or **nodes**
- **E** is a set of all **edges** or **relationships** with endpoints from set V

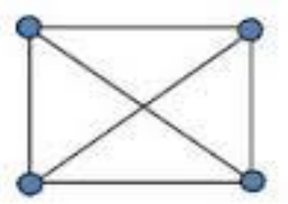


What is a Graph

- Special edges: **loops** and **multiple** edges
- **Loop**: An edge whose endpoints are equal
- **Multiple edges**: Edges that have the same pair of endpoints



It is not simple.

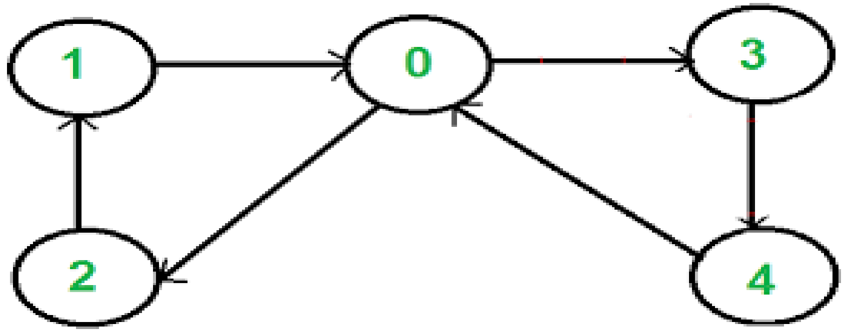


It is a simple graph.

- Graphs without loops and multiple edges: simple graphs
- Graphs with multiple edges: multigraphs

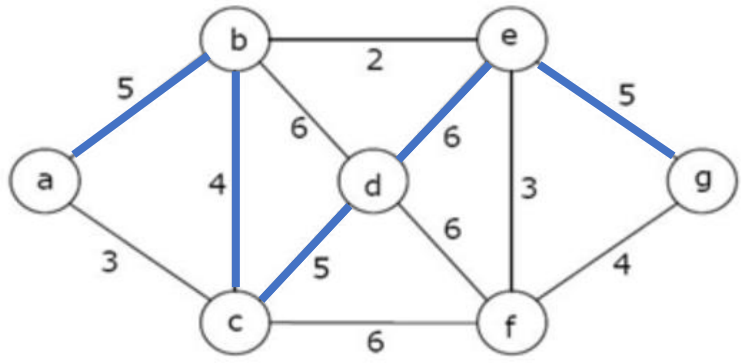
Directed Graphs

- A directed graph G is defined as G = (V, E)
- **V** is a set of all **vertices**
- **E** is a set of all directed **edges (u,v)**, a directed edge (u,v) is an **outgoing** edge of u, and an **incoming** edge of v.



Weighted Graphs

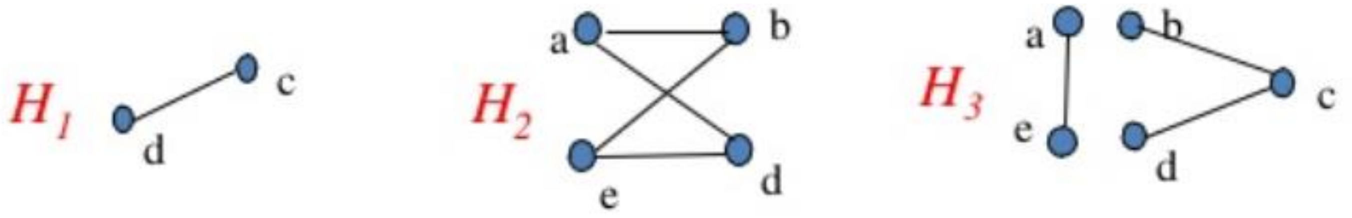
- A weighted graph is a graph whose edges have been **labeled** with some **weights** (numbers).
- The length of a path in a weighted graph is the sum of the weights of all the edges in the path.



- The length of the path a -> b -> c -> d -> e -> g, is 5 + 4 + 5 + 6 + 5 = 25

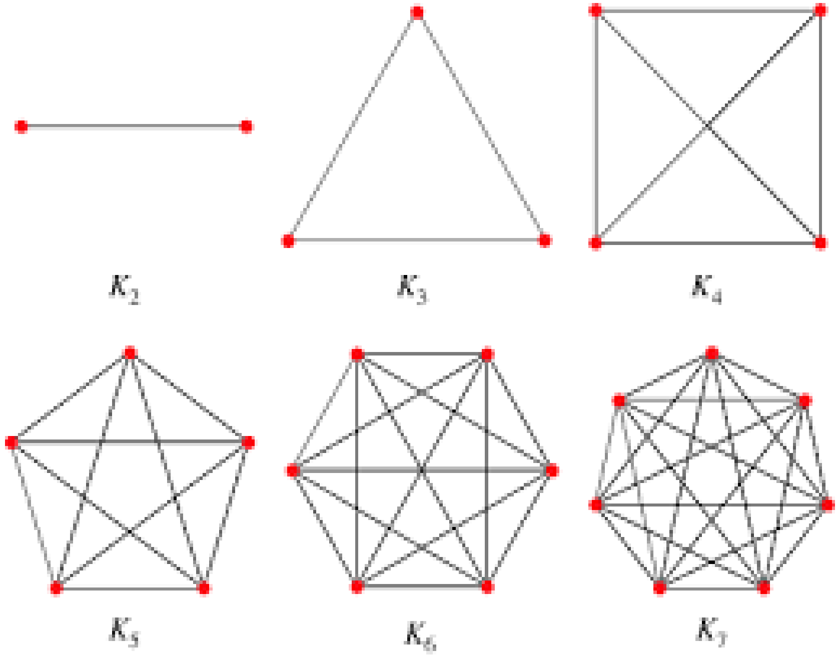
Connected-Disconnected Graphs

- **Connected**: Exists at least one path between any two vertices
- **Disconnected**: Otherwise
- Example:
  - H1 and H2 are connected
  - H3 is disconnected



Complete Graph

- **Complete Graph**: A *simple graph* in which every pair of vertices are adjacent
- If number of vertices=n, then there are
  - n(n-1)/2 edges for undirected graphs
  - n(n-1) edges for directed graphs
- **Sparse** Graph: If $|E| \approx |V|$
- **Dense** Graph: if $|E| \approx |V|^2$

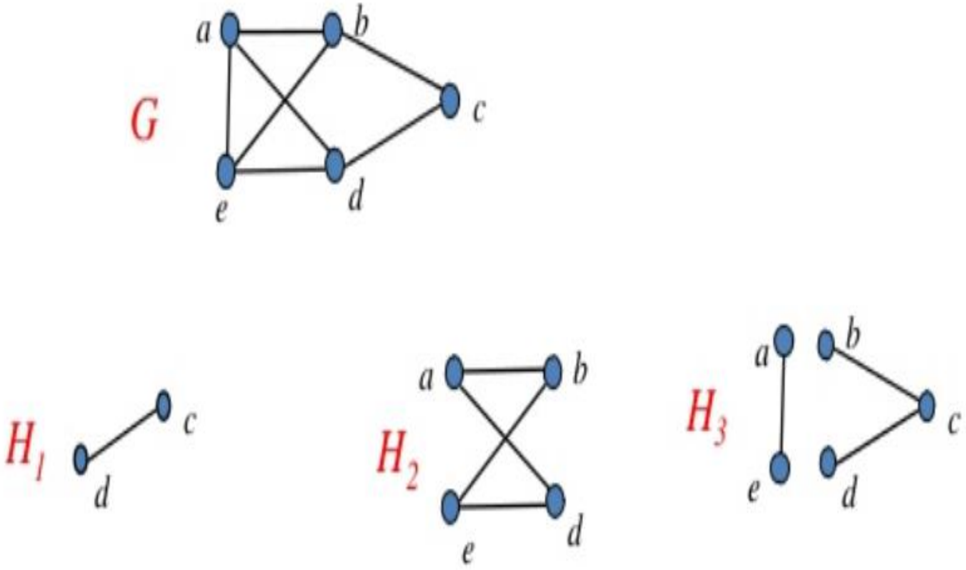


Complete undirected graphs of increasing size (n)

Subgraphs

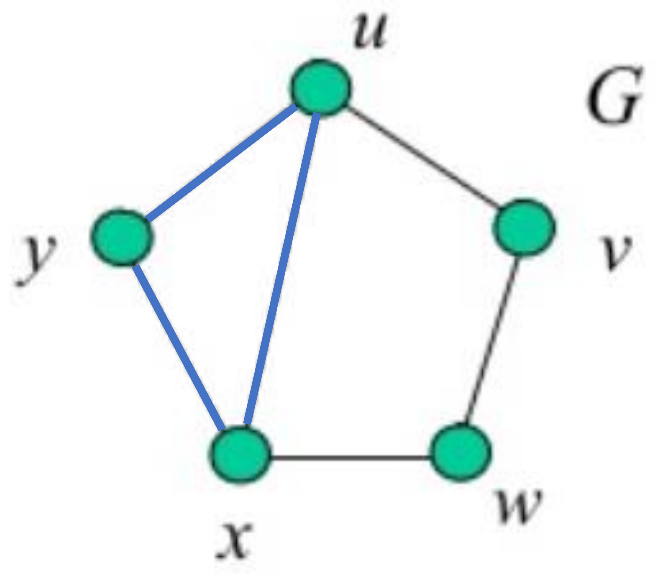
- **A subgraph** of a graph G is a graph H such that:
  - $V(H) \subseteq V(G)$
  - $E(H) \subseteq E(G)$
  - $(v_1, v_2) \in E(H) \rightarrow v_1, v_2 \in G(H)$

  (you cannot pick an edge without selecting its endpoints)
- If the subgraph contains every possible edge between the nodes in V(H) it is an induced subgraph
- H1, H2, H3 are subgraphs of G



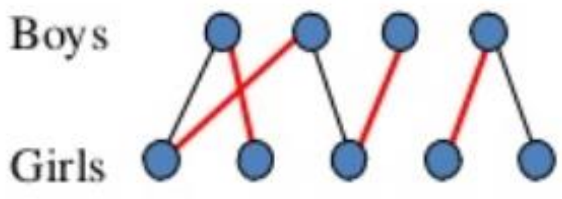
Clique – Independent Set

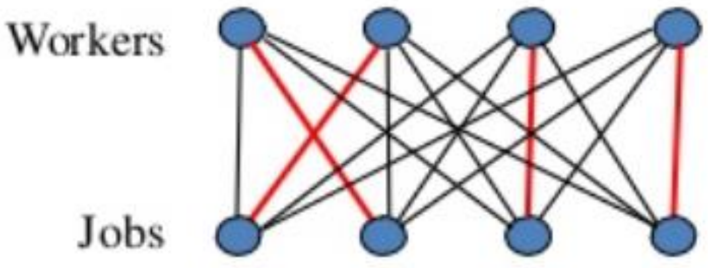
- **Clique**: A set of pairwise adjacent vertices (a complete subgraph of a graph G)
- **Independent set**: A set of pairwise nonadjacent vertices
- Example:
  - {x,y,u} is a clique in G
  - {u,w} is an independent set



Bipartite Graphs

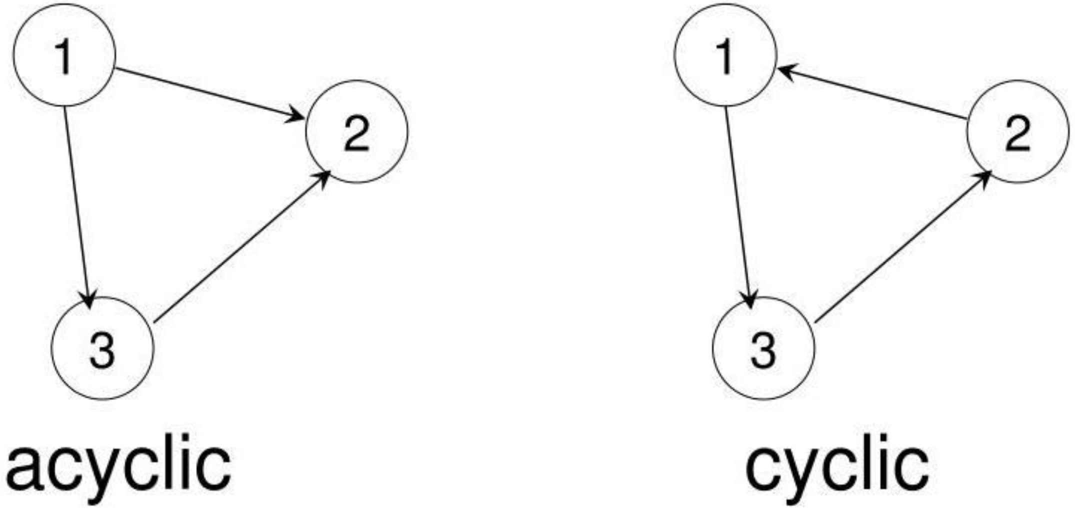
- **A bipartite graph** is a graph whose vertices can be divided into two disjoint and independent sets U and V, such that every edge connects a vertex in U to one in V





Cyclic - Acyclic Graphs

- A path from a vertex to itself is called a **cycle**
- A graph is called **cyclic** if it contains a cycle
  - Otherwise it is called **acyclic**



Graph Representations

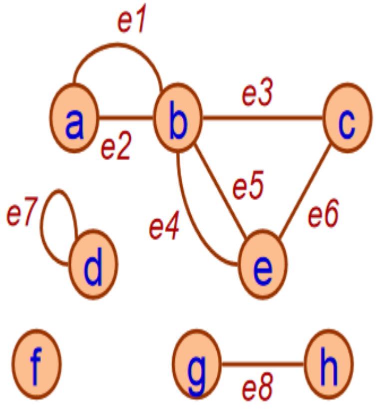
Graph Descriptions (1): **Incidence Matrix**

- One row per edge
- One column per vertex
- Value=1 if edge and vertex are **incident**
- Used mainly for simple undirected graphs
  - Can be extended for more general graphs (hypergraphs, directed, with loops)
    - ….but becomes ugly

nodes

edges

edge\vertex	a	b	c	d	e	f	g	h
e1	1	1	0	0	0	0	0	0
e2	1	1	0	0	0	0	0	0
e3	0	1	1	0	0	0	0	0
e4	0	1	0	0	1	0	0	0
e5	0	1	0	0	1	0	0	0
e6	0	0	1	0	1	0	0	0
e7	0	0	0	1	0	0	0	0
e8	0	0	0	0	0	0	1	1



Graph Descriptions (2): **Adjacency Matrix**

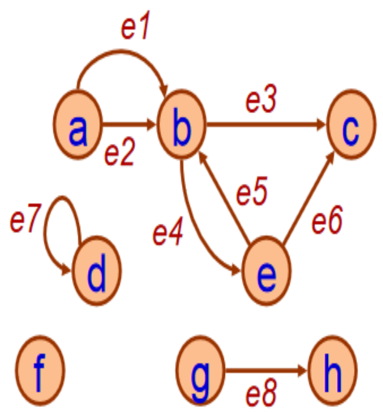
- One row per vertex
- One column per vertex
- Value=1 if vertices are connected via an edge
- Diagonal: self loops

- **Pros:**
- Easier to implement and follow
- Removing an edge takes O(1) time
- Queries like whether there is an edge from vertex u to vertex v are efficient and can be done O(1).

- **Cons:**
- Can't represent multi-edges
- Inefficient storage $O(|V|^2)$ (many empty cells especially for sparse graphs)

nodes (columns) / nodes (rows)

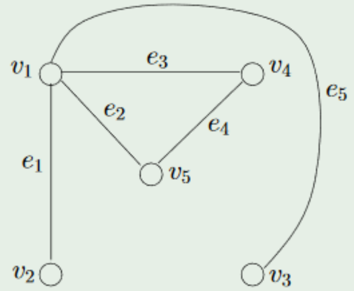
from\to	a	b	c	d	e	f	g	h
a	0	1	0	0	0	0	0	0
b	0	0	1	0	1	0	0	0
c	0	0	0	0	0	0	0	0
d	0	0	0	1	0	0	0	0
e	0	1	1	0	0	0	0	0
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	1
h	0	0	0	0	0	0	0	0



Graph Descriptions (2): Adjacency Matrix

- Adjacency Matrix undirected graph:
- Matrix **must** be symmetric.
  - We can optimize storage by maintaining e.g. only the lower triangle

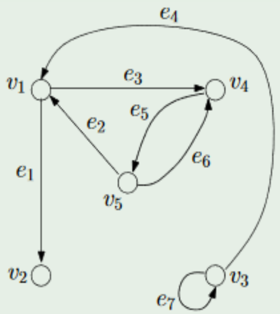
Example 3.



	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	1
v_2	1	0	0	0	0
v_3	1	0	0	0	0
v_4	1	0	0	0	1
v_5	1	0	0	1	0

- Adjacency Matrix directed graph:
- Matrix **may not** be symmetric.

Example 4.

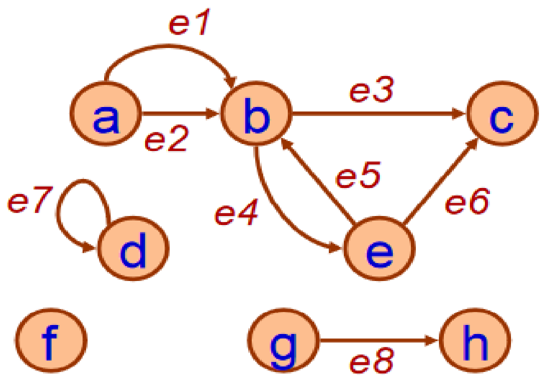


	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	1	0
v_2	0	0	0	0	0
v_3	1	0	1	0	0
v_4	0	0	0	0	1
v_5	1	0	0	1	0

Graph Descriptions (3): **Adjacency List**

- A list of out-going vertices is associated to each vertex
  - Example: in a social network, keep list of friends for each user (node)
- Compact representation
- Optionally, a list of in-going vertices can be added for reverse traversal (directed graphs)

Vertex	Out	In
a	(b,b)	()
b	(c,e)	(a,a,e)
c	()	(b,e)
d	(d)	(d)
e	(b,c)	(b)
f	()	()
g	(h)	()
h	()	(g)



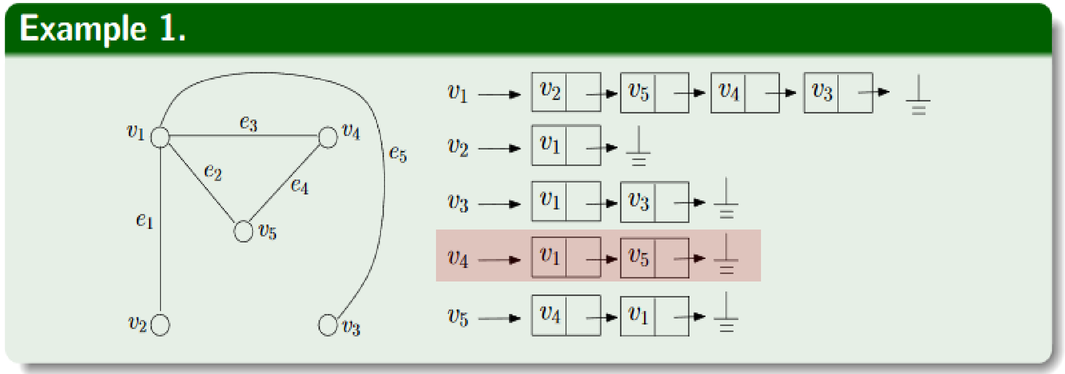
Graph Descriptions (3): Adjacency List

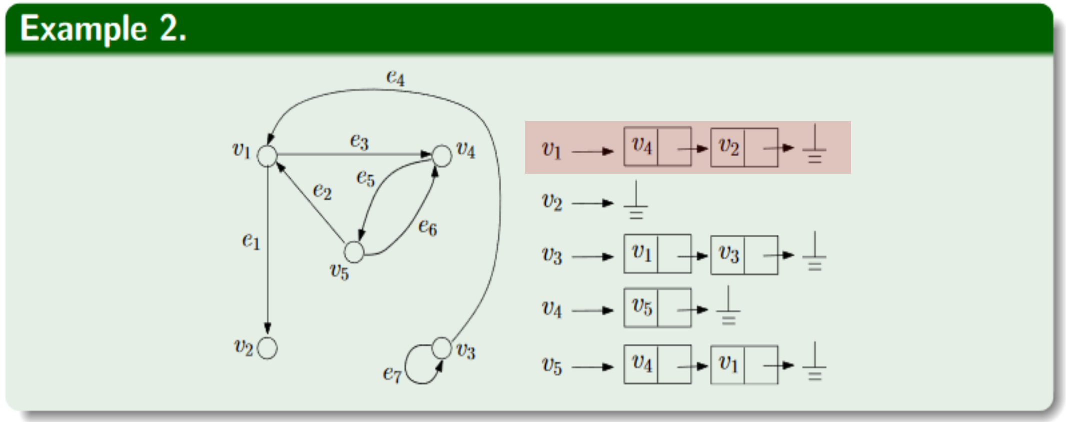
- Adjacency list undirected graph:
- Space =O(|V|+|E|)

- Assume edges denote friendships in FB
  - Query 1: Who are the friends of user v_4

- Adjacency list directed graph:
- Space =O(|V|+|E|)

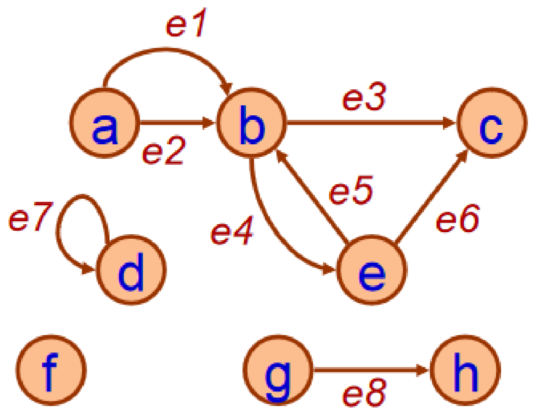
- Assume edges denote links among web pages
  - Query 2: Find all links emanating from page v_1





Graph Descriptions (4): **Edge List**

- One row per edge
- One column for starting node (heads)
- One column for ending node (tails)
- Optional columns for edge attributes (label, weight, color,…)



head	tail	label
a	b	e1
a	b	e2
b	c	e3
b	e	e4
e	b	e5
e	c	e6
d	d	e7
g	h	e8

- Straightforward to store in a relational table or a dataframe
- Traversals require costly operators (self-joins)

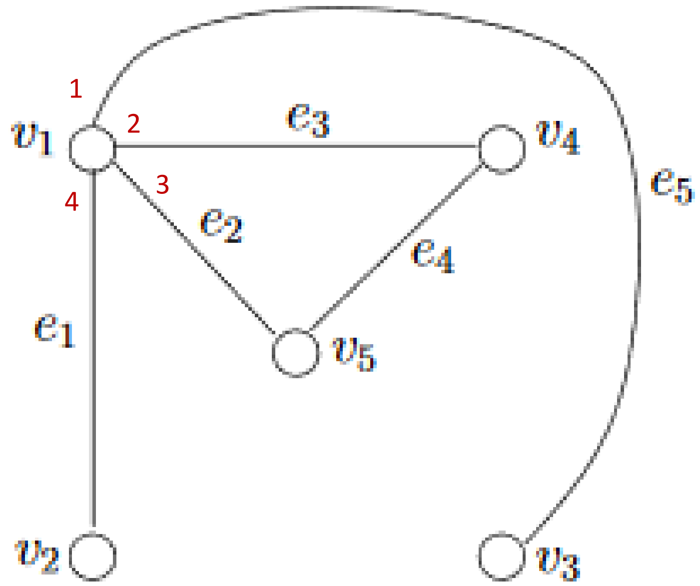
Graph Topology Metrics

Degree – Undirected Graphs

- In an undirected graph the degree (k) of a node is the number of edges for which it is an endpoint.
- Examples:
  - The degree of node v_3 is 1
  - The degree of node v_1 is 4

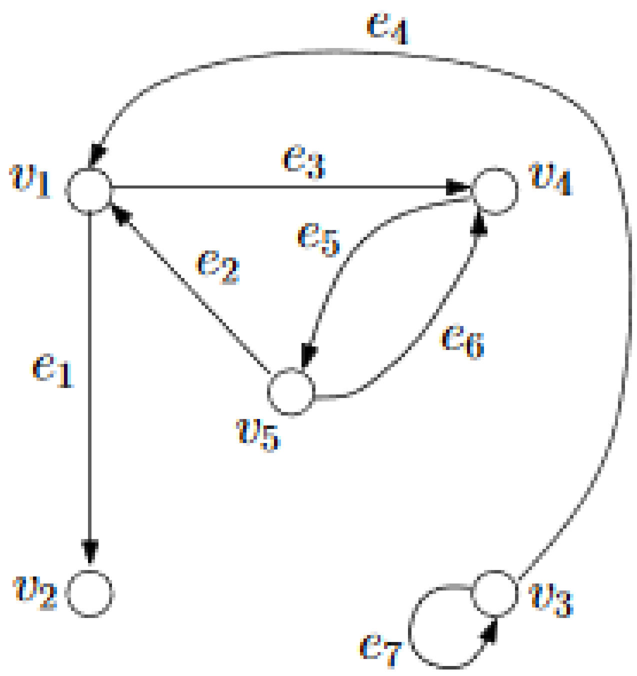
- Minimum degree: $\delta(G) = \min_{u \in V} d(u)$
- Maximum degree: $\Delta(G) = \max_{u \in V} d(u)$

- Quiz: what is $\Sigma_{u \in V} d(u)$?



Degree – Directed Graphs

- The **in-degree (k_{in})** of a node is the number of edges for which it is the tail
- The **out-degree (k_{out})** of a node is the number of edges for which it is the head
- The **total degree (k)** of a node is the sum of in-degree and out-degree
  - $k=k_{in}+k_{out}$
- Examples:
  - The in-degree of v_3 is 1
  - The out-degree of v_3 is 2
  - The total degree of node v_1 = 2+2



Local Clustering Coefficient

- The Local clustering coefficient **C(i)** of a node i, quantifies how close its neighbors (k) are to being a clique

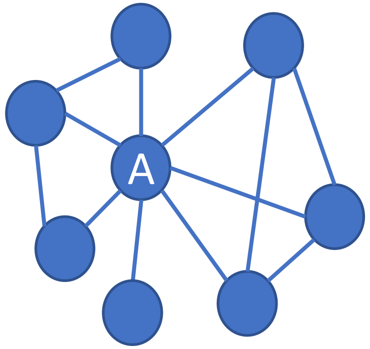
- Assume nodes depict users in a social network and edges their relationships
  - The clustering coefficient C(A) of node A is defined as the probability that two randomly selected friends of A are friends themselves
    - i.e. the fraction of all pairs of A's friends who are also friends

- Defined only if A has at least two friends (otherwise 0)

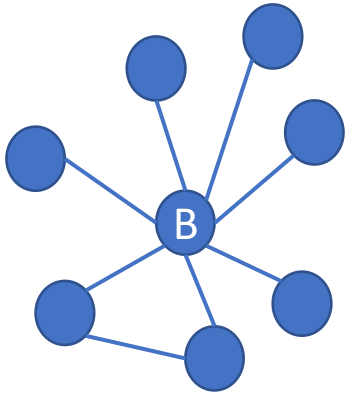
- The clustering coefficient is always between 0 and 1

Detect Fake Users In Social Networks

- Assumption: fake accounts add friends at random

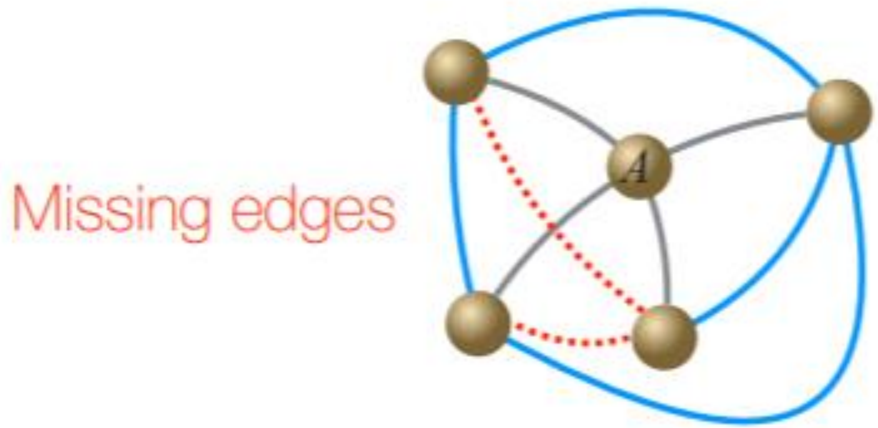


$TC_A = 5$



$TC_B = 1$

Local Clustering Coefficient (Simple undirected graph)



- **Node A** has k=4 friends
- Among the four friends, there are k×(k-1)/2 = (4×3)/2 = 6 possible friendships
- But only four of them are actually present
- Two are missing
- Thus, the clustering coefficient of **node A** is C(A)=4/6=0.6666, or about 67%

Local Clustering Coefficient general formula

- Local clustering coefficient C(i) of a node i is computed as the ratio between the number of edges (n) among its k_i neighbors divided by the number of links (M) that could possibly exist among them:

$$\mathbf{C_i = \frac{n}{M}}$$

- Note that the maximal number of edges (M) depends on the graph type
  - Directed or undirected
  - With or without self-loops

Local Clustering Coefficient

Existing edge

Missing edge

- Undirected, without self-loops:

$$C_i = \frac{n}{M} = \frac{n}{k_i(k_i-1)/2}$$

- Undirected, with self-loops:

$$C_i = \frac{n}{M} = \frac{n}{\frac{k_i(k_i-1)}{2} + k_i}$$

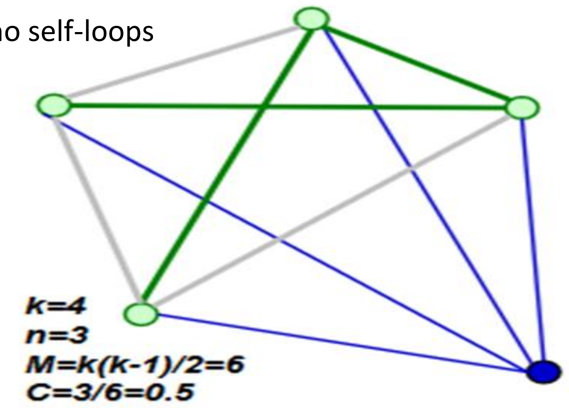
- Directed, without self-loops:

$$C_i = \frac{n}{k_i(k_i-1)}$$

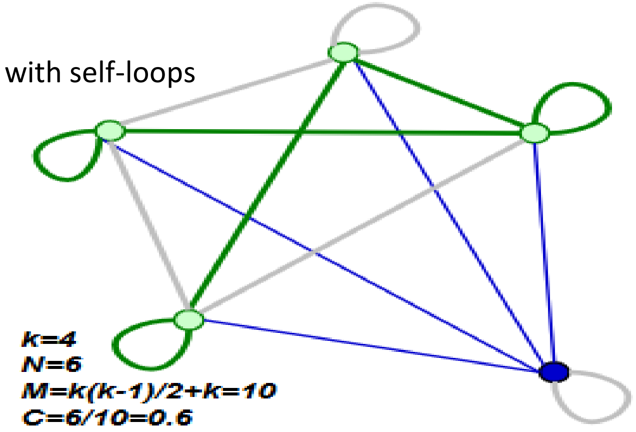
- Directed, with self-loops:

$$C_i = \frac{n}{k_i^2}$$

Undirected, no self-loops

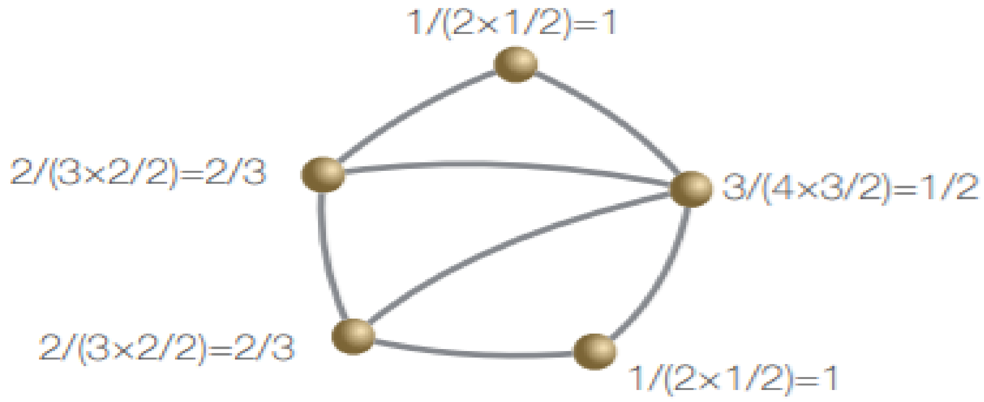


Undirected, with self-loops



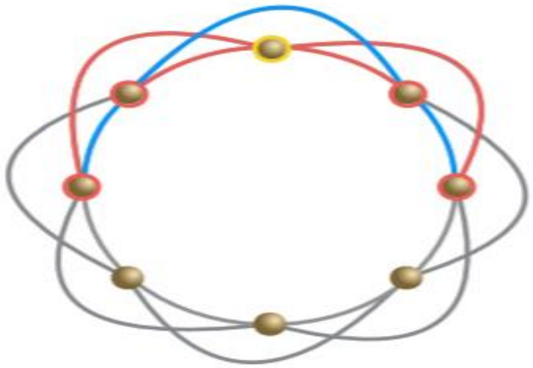
Average Clustering Coefficient

- **Average Clustering Coefficient CC of a graph G** is the average of the clustering coefficients of all nodes in G



$$CC = (1+2/3+2/3+1+1/2)/5 = 0.7666$$

Average Clustering Coefficient



- All nodes are identical and have 4 neighbors
- Possible edges between pairs of neighbors is $4 \times 3/2 = 6$
- How many pairs of neighbors are actually connected? 3
- Clustering coefficient of any node: $3/6 = 0.5$
- Clustering coefficient of the entire graph: $CC = 0.5$

Edge Density

- Edge density of a graph is the actual number of edges m in proportion to the maximum possible number of edges
- E.g. for undirected simple graphs

$$\rho = \frac{m}{n(n-1)/2} = \frac{2m}{n(n-1)}$$

- The edge density takes values between 0 and 1
- Suppose we pick two nodes of a graph at random without regard to the graph structure (e.g., whether the two nodes share a common neighbor or not)
- **What is the probability p that the two nodes are connected?**
  - It is given exactly by the edge density of the graph, probability $p=\rho$
  - Density captures the general degree of cohesion (=συνοχή) in a graph

Sparse and Dense Graphs

- If ρ is "small", then graph is sparse
- If ρ is "large", then the graph is dense

Sparse (ρ=3/(8×7/2)=3/28=0.1071) Denser (ρ=11/28=0.3928)

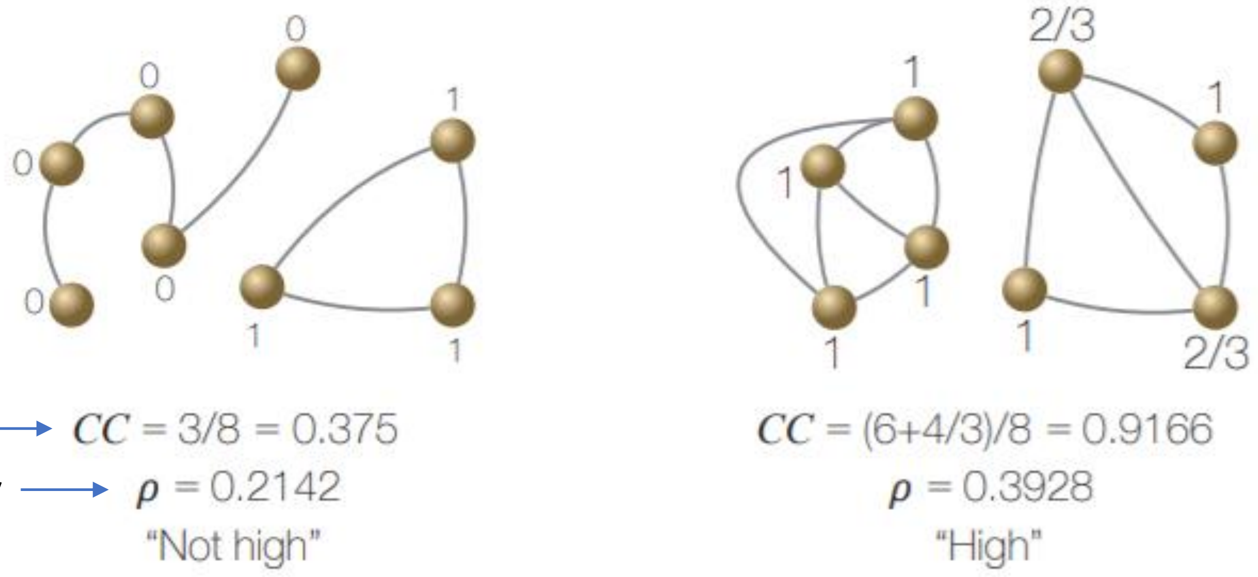
Highly Clustered Graphs

- A graph may contain dense “clusters” even if it is sparse
- Compare the average clustering coefficient CC of a graph to its edge density
- We consider a graph to be highly clustered if CC ≫ ρ

Local clustering coefficient

Average clustering coefficient

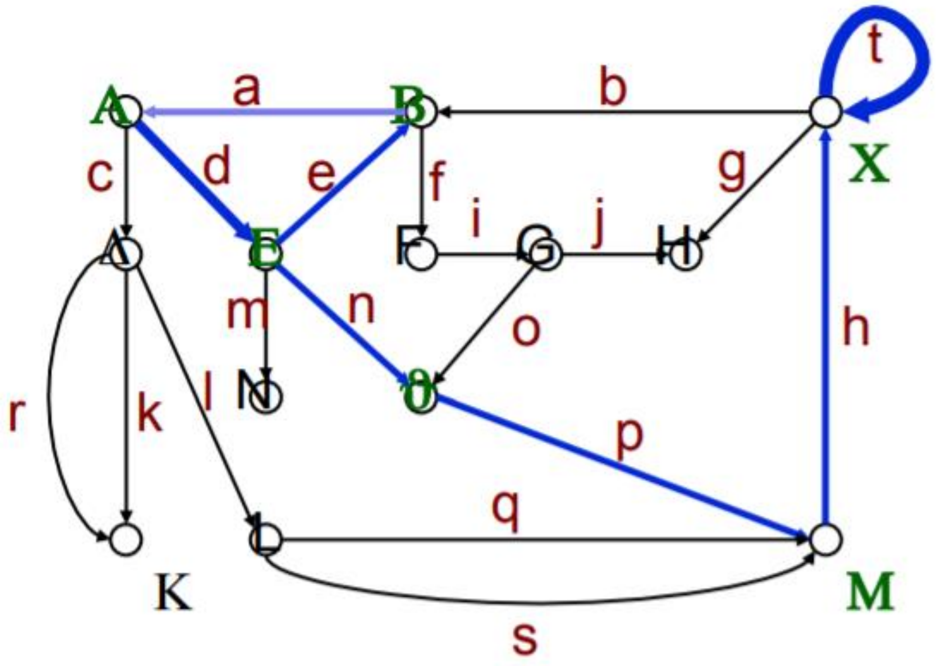
density



Walks, trails and paths

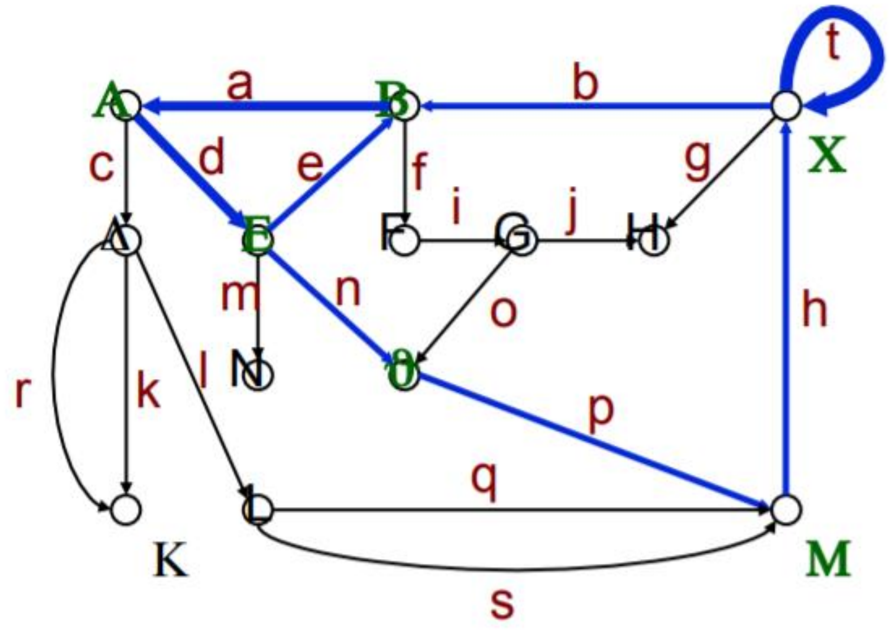
Walk

- A **walk** is defined as a finite length alternating sequence of vertices and edges
- The total number of edges covered in a walk is called as **Length of the Walk**
- Remarks:
  - A walk can be described unequivocally by the sequence of edges (e.g.: **d, e, a, d, n, p, h, t, t, t**)
  - An edge or a vertex can appear repeatedly in the same walk (e.g.: edges **d and t** , and vertices **A, E, X**)



Open – Closed Walks

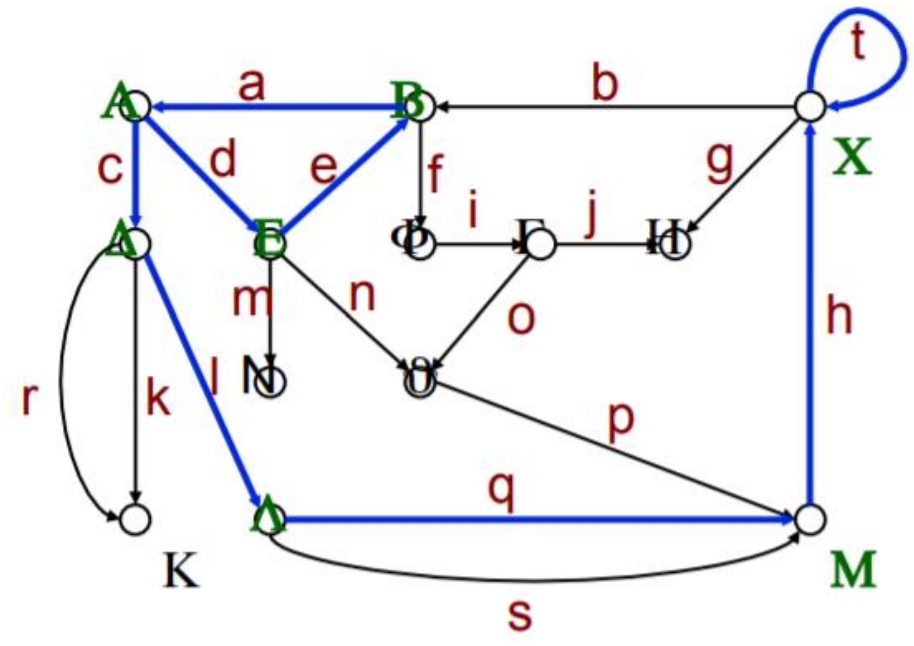
- **Open walk**: The vertices at which the walk starts and ends are different
  - **(d, e, a, d, n, p, h, t, t, t)**
- **Closed Walk:** The vertices at which the walk starts and ends are same
  - **(d, e, a, d, n, p, h, t, t, t, b, a)**



Trail

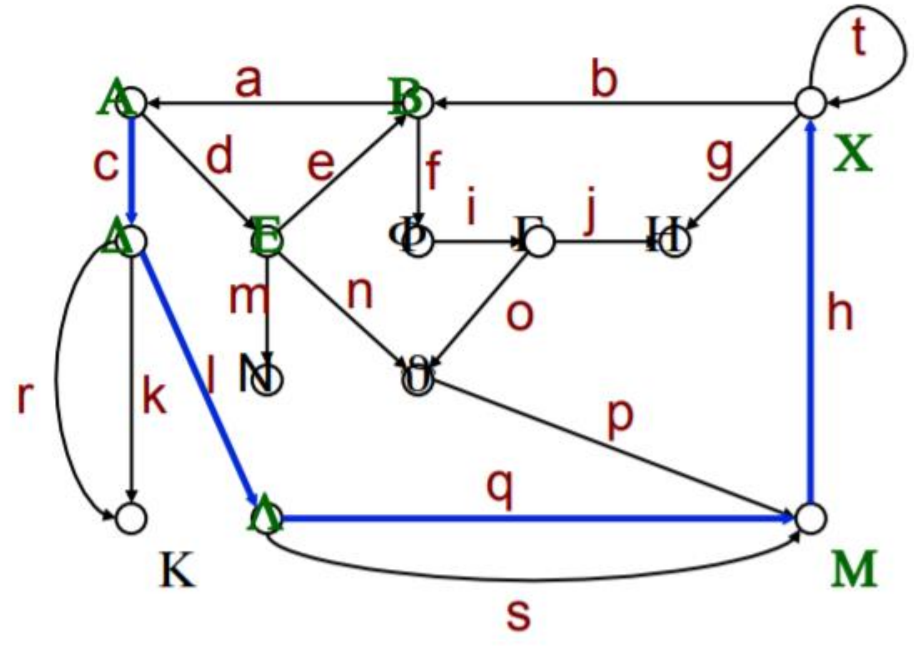
- A **trail** is a walk with no repeated edges
  - **(d, e, a, c, l, q, h, t)**

- Remark: a vertex can appear repeatedly in the same trail
  - (e.g.: A and X)



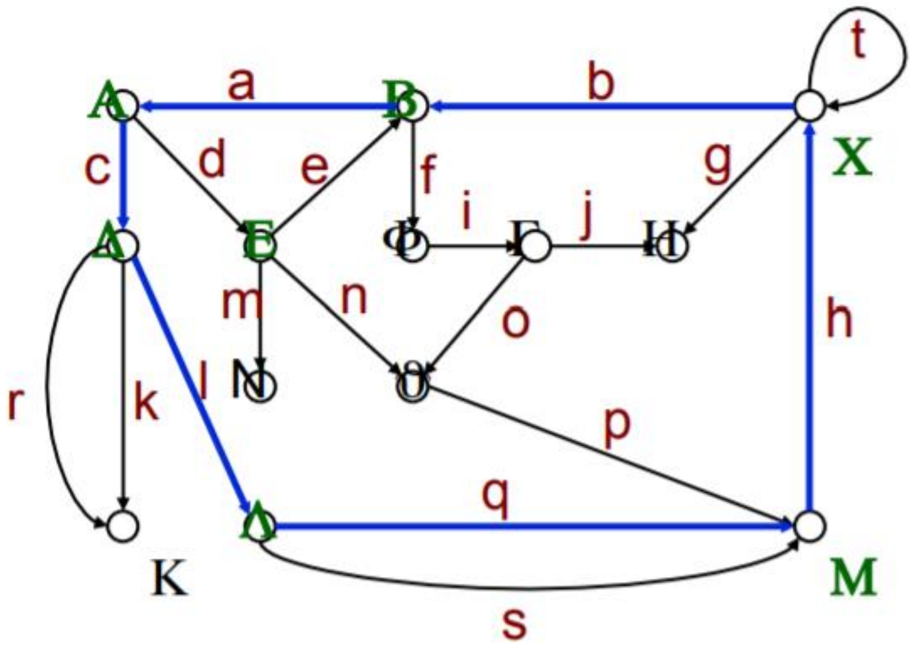
Path

- A **path** is a trail with no repeated vertices, except possibly the initial and final vertex (nor edges are allowed to repeat)
  - **(c, l, q, h)**

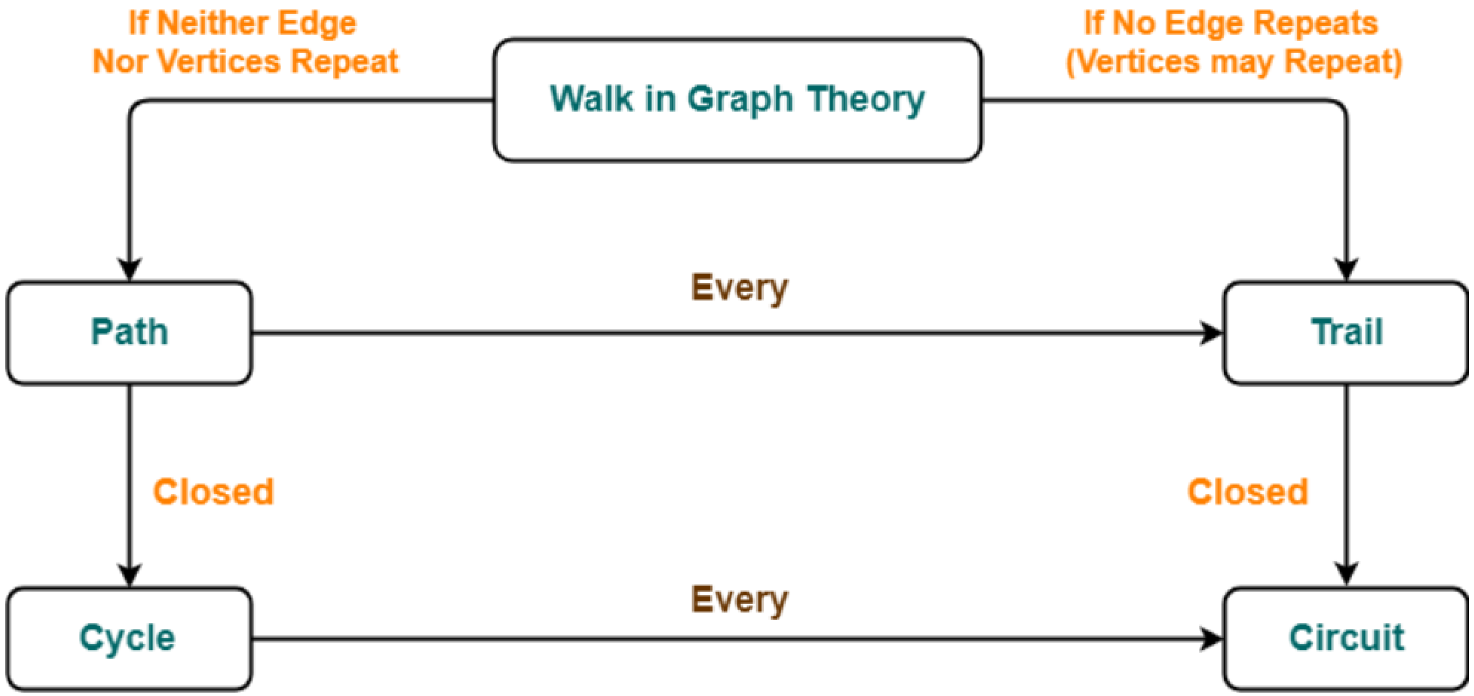


Cycle

- A **cycle** is a closed path with at least one edge
  - **(c, l, q, h, b, a)**

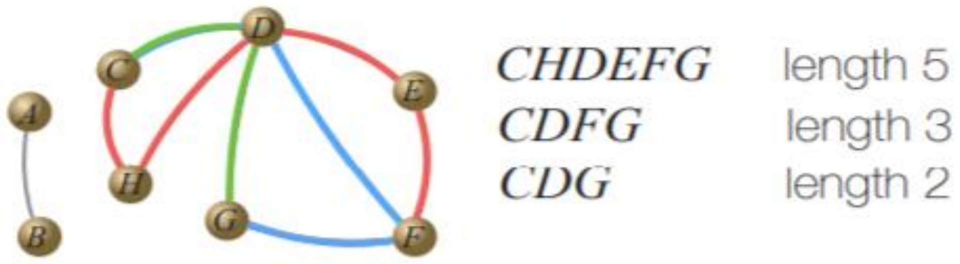


Walks - Paths - Trails



Length - Distance

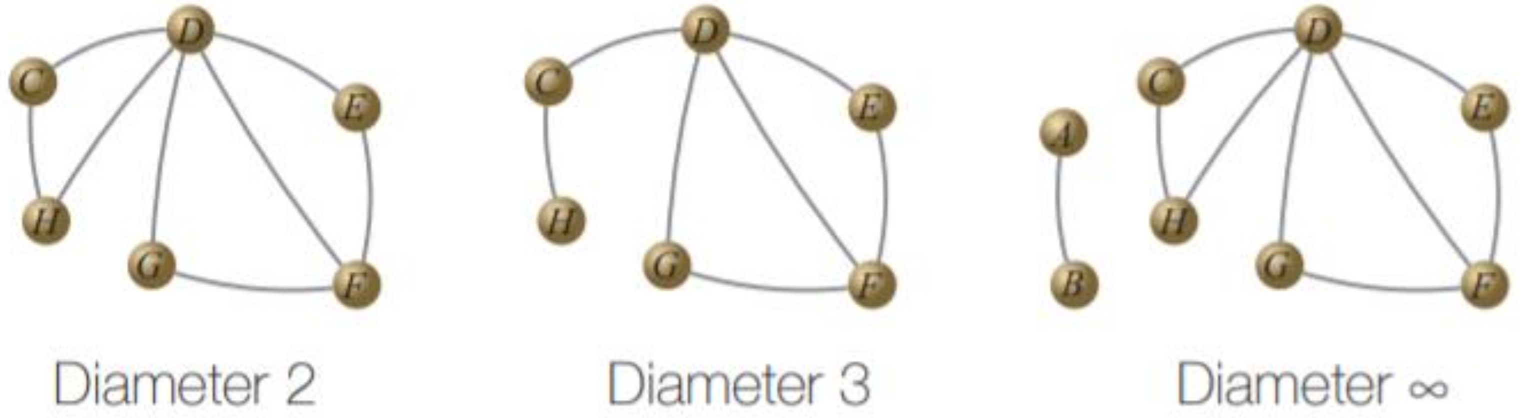
- The **length** of a path in a graph is the number of steps it contains from beginning to end (number of edges)



- The **distance** between two nodes in a graph is the length of the shortest path between them
  - Distance between C and G is 2
  - Distance between A and B is 1
  - Distance between A and C is infinite (or undefined)

Diameter

- **Diameter** of a graph is the longest of the distances between all pairs of nodes (the longest shortest path)



Shortest Paths

Unweighted Graphs: Shortest Path

Unweighted graphs:

- Input: an unweighted graph (all edges are of equal weight)
- Goal:
  - **Single-source shortest path**: Given a graph G and a source vertex s, find the path with smallest number of hops to every other vertex in G
  - **Point to Point SP problem:** Given G and two vertices A and B, find a shortest path from A (source) to B (destination)
  - **All Pairs Shortest Path Problem:** Given G find a shortest path between all pairs of vertices

Unweighted Graphs: Shortest Path

Unweighted graphs:

- Goal:
  - **Single-source shortest path**: Given a graph G and a source vertex s, find the path with smallest number of hops to every other vertex in G
  - **Solution:**
    - Use BFS Algorithm starting with source vertex s
    - Time: O(|E|)

Unweighted Graphs: Shortest Path

Unweighted graphs:

- Goal:
  - **Point to Point SP problem:** Given G and two vertices A and B, find a shortest path from A (source) to B (destination)
- **Solution:**
  - Run BFS using source as A
  - Stop algorithm when B is reached.
  - Time: O(|E|)

Unweighted Graphs: Shortest Path

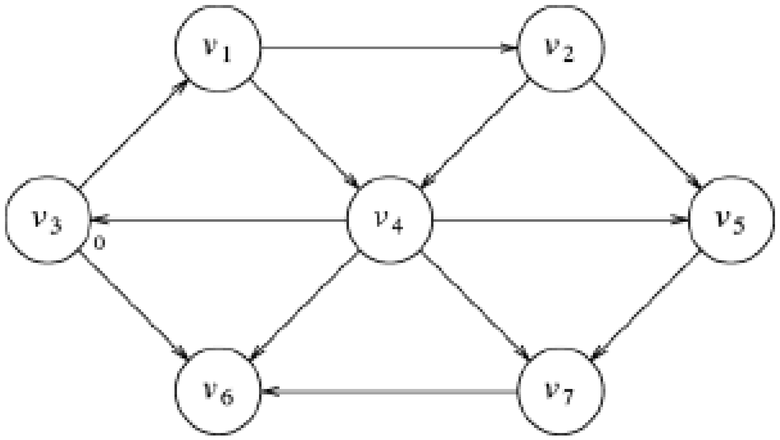
Unweighted graphs:

- Goal:
- **All Pairs Shortest Path Problem:** Given G find a shortest path between all pairs of vertices
- **Solution:**
  - Solve Single Source Shortest Path for each vertex as source
  - Time: $O(|V||E|)$

BFS Algorithm

For each vertex, keep track of:

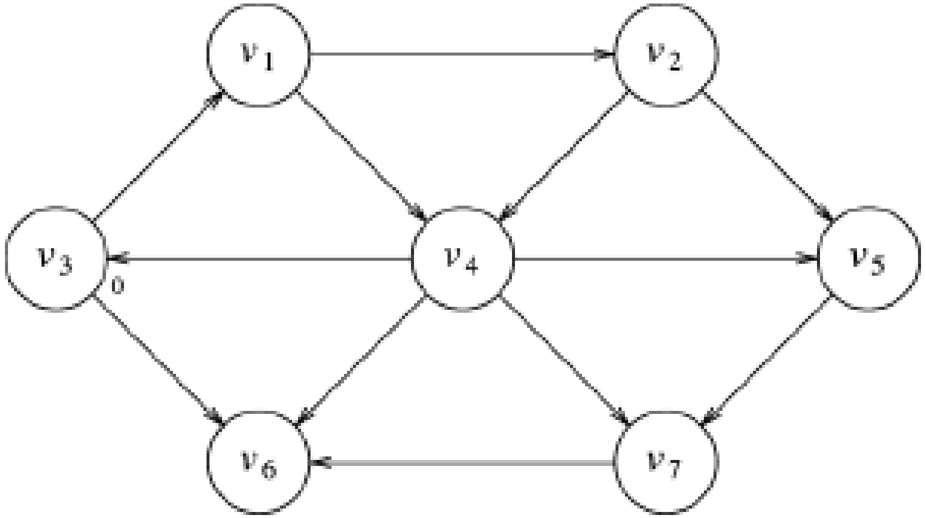
- Whether we have visited it (known)
- Its distance from the start vertex (d_v)
- Its predecessor vertex along the shortest path from the start vertex (p_v)



	Initial State		
v	*known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

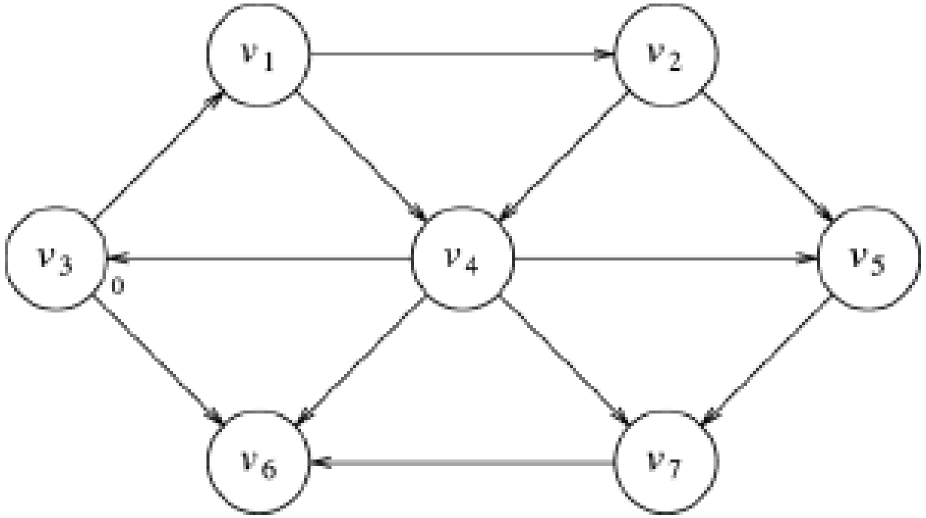
BFS Algorithm

- Ignore vertices that have already been visited by keeping only unvisited vertices (distance = ∞) on the queue



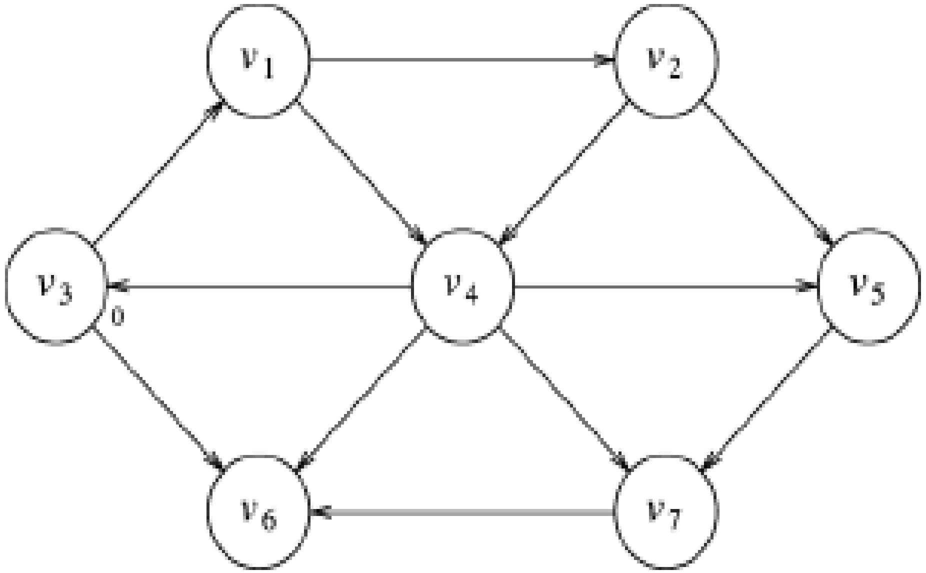
	Initial State			v_3 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3
v_2	F	∞	0	F	∞	0
v_3	F	0	0	T	0	0
v_4	F	∞	0	F	∞	0
v_5	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3
v_7	F	∞	0	F	∞	0
Q:		v_3			v_1, v_6	

BFS Algorithm



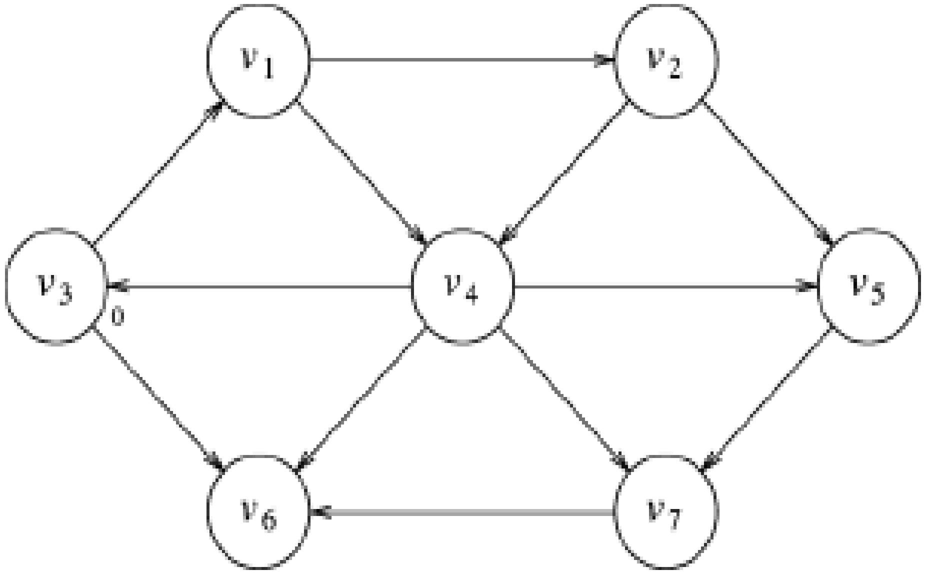
	v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3
v_2	F	2	v_1	F	2	v_1
v_3	T	0	0	T	0	0
v_4	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0
v_6	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0
Q:	v_6, v_2, v_4			v_2, v_4		

BFS Algorithm



	v_2 Dequeued			v_4 Dequeued		
v	known	d_v	p_v	known	d_v	p_v
v_1	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2
v_6	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4
Q:	v_4, v_5			v_5, v_7		

BFS Algorithm



	v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0
v_4	T	2	v_1	T	2	v_1
v_5	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3
v_7	F	3	v_4	T	3	v_4
Q:		v_7			empty	

BFS Algorithm

Given (undirected or directed) graph G = (V, E) and source node s ∈ V

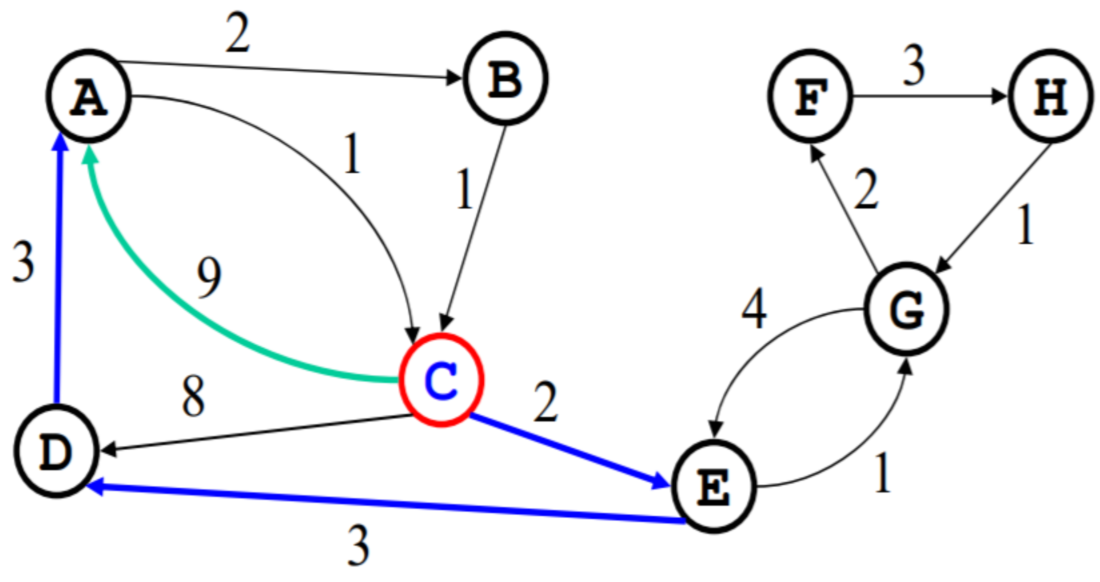
```
BFS(s)
Mark all vertices as unvisited
Initialize search tree T to be empty
Mark vertex s as visited
Set Q to be the empty queue
Enq(s) (Adds an element to the end of the list)
while Q is nonempty do
        u = deq(Q) (Removes an element from the front of the list)
        for each vertex v ∈ Adj(u)
                if v is not visited then
                        add edge (u, v) to T
                        mark v as visited and enq(v)
```

Weighted Graphs: Shortest Path

What if edges have weights?

- Breadth First Search does not work anymore -› minimum cost path may have more edges than minimum length path

- **Shortest path (length) from C to A:** C->A (cost = 9)
- **Minimum Cost Path** = C->E->D->A (cost = 8)



Weighted Graphs: Shortest Path

Weighted graphs:

- Input: a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge
- Cost of a path $v_1, v_2, \dots, v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$
- Goal: to find a smallest cost path

- **Single-source shortest path**: Given a weighted graph G (V,E) and a source vertex s, find the minimum weighted path from s to every other vertex in G
- **Point to Point SP problem:** Given a weighted graph G and two vertices A and B, find a shortest path from A (source) to B (destination)
- **All Pairs Shortest Path Problem:** Given a weighted graph G find a shortest path between all pairs of vertices

Weighted Graphs: Shortest Path

Weighted graphs:

- Goal:
  - **Single-source shortest path**: Given a weighted graph G (V,E) and a source vertex s, find the minimum weighted path from s to every other vertex in G
  - **Solution:**
    - Use Dijkstra's algorithm starting with source vertex s
    - Time: O((n + m) log n)
    - Does not work with **negative** weights

Weighted Graphs: Shortest Path

Weighted graphs:

- Goal:
- **Point to Point SP problem:** Given a weighted graph G and two vertices A and B, find a shortest path from A (source) to B (destination)
- **Solution:**
  - Run Dijkstra's algorithm using source as A
  - Stop algorithm when B is reached.
  - Time: $O((n + m) \log n)$
  - Does not work with **negative** weights

Weighted Graphs: Shortest Path

Weighted graphs:

- Goal:
  - **All Pairs Shortest Path Problem:** Given a weighted graph G find a shortest path between all pairs of vertices
- **Solution:**
  - Run Dijkstra's algorithm for each vertex as source
  - Time: $O(mn+n^2 \log n)$
  - Does not work with **negative** weights

Dijkstra’s Algorithm

- Classic **greedy** algorithm for solving shortest path in weighted graphs (without negative weights)

Basic Idea:

- Find the vertex with smallest cost that has not been “marked” yet
- Mark it and compute the cost of its neighbors
- Do this until all vertices are marked
- Note that each step of the algorithm we are marking one vertex and we won’t change our decision: hence the term “greedy” algorithm
- Works for directed and undirected graphs

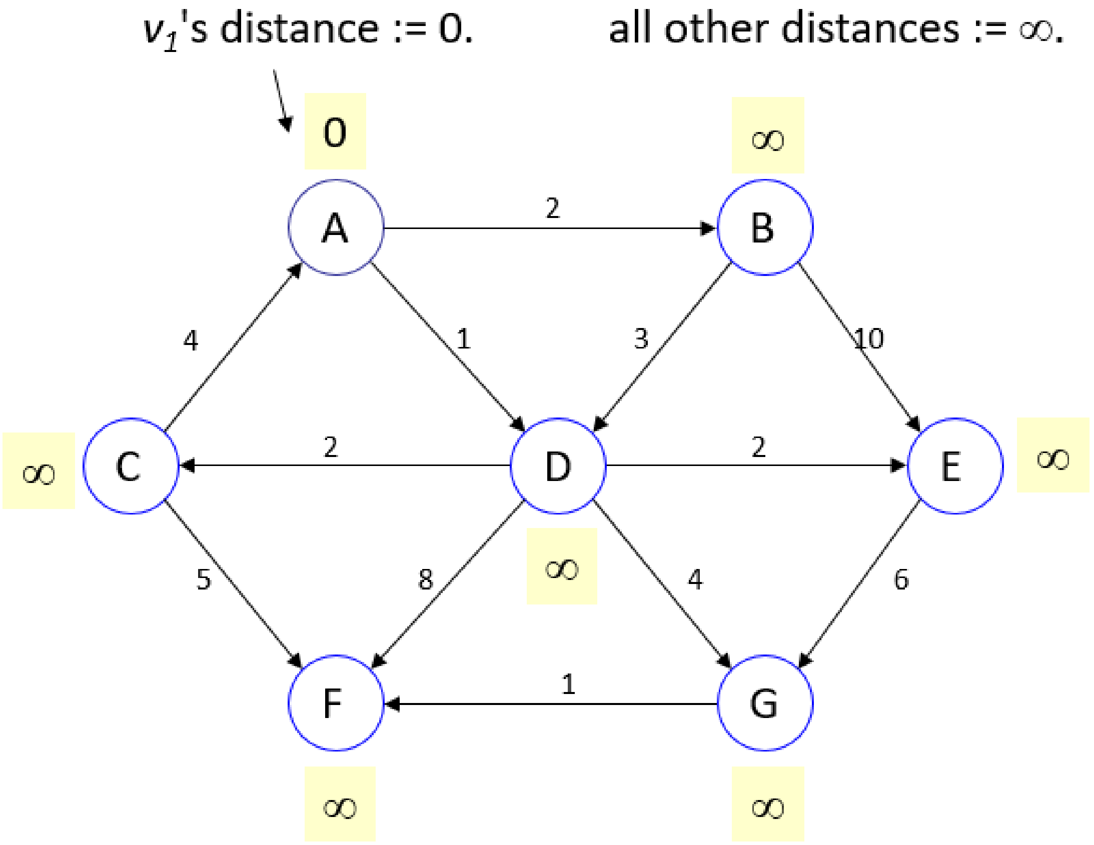
Dijkstra's Algorithm

- Initialize the cost of s to 0, and all the rest of the nodes to ∞
- Initialize **set S** to be $\emptyset$
  - S is the set of nodes to which we have a shortest path
- While S is not all vertices
  - Select the node A with the lowest cost that is not in S and identify the node as now being in S
  - For each node B adjacent to A
    - if cost(A)+cost(A,B) < B's currently known cost – set cost(B) = cost(A)+cost(A,B)
    - set previous(B) = A so that we can remember the path

Dijkstra’s Algorithm Example

Initialization:

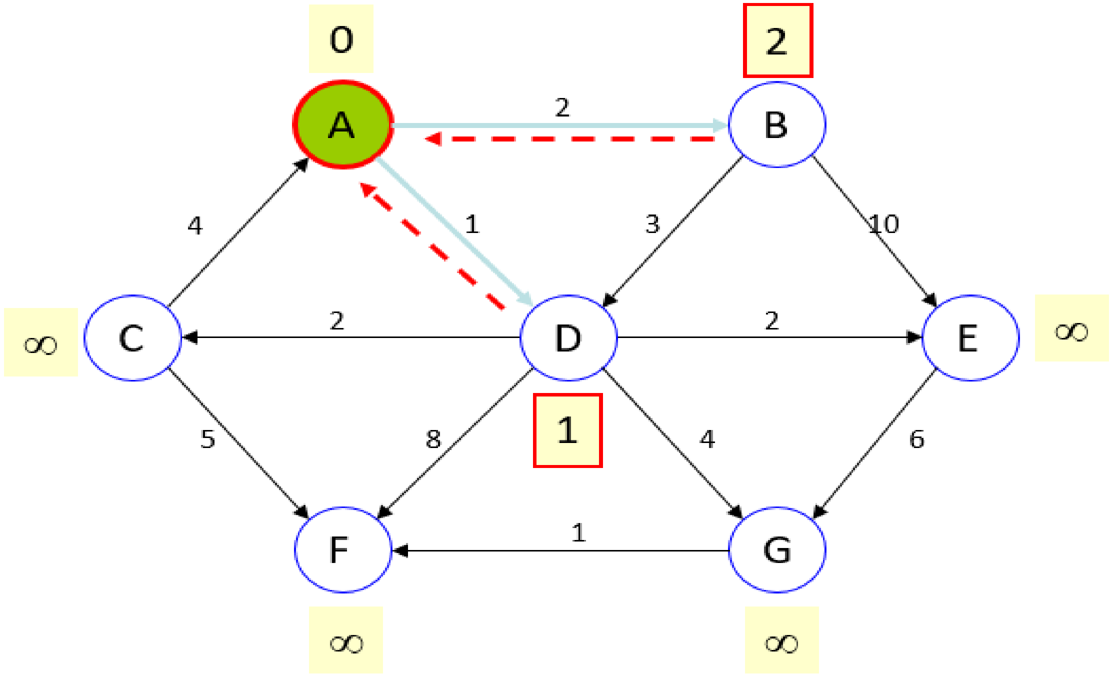
- S=[A]
- **Cost(A)=0**
- Cost(B)=∞
- Cost(C)=∞
- Cost(D)=∞
- Cost(E)=∞
- Cost(F)=∞
- Cost(G)=∞



Dijkstra’s Algorithm Example

Update Cost neighbors:

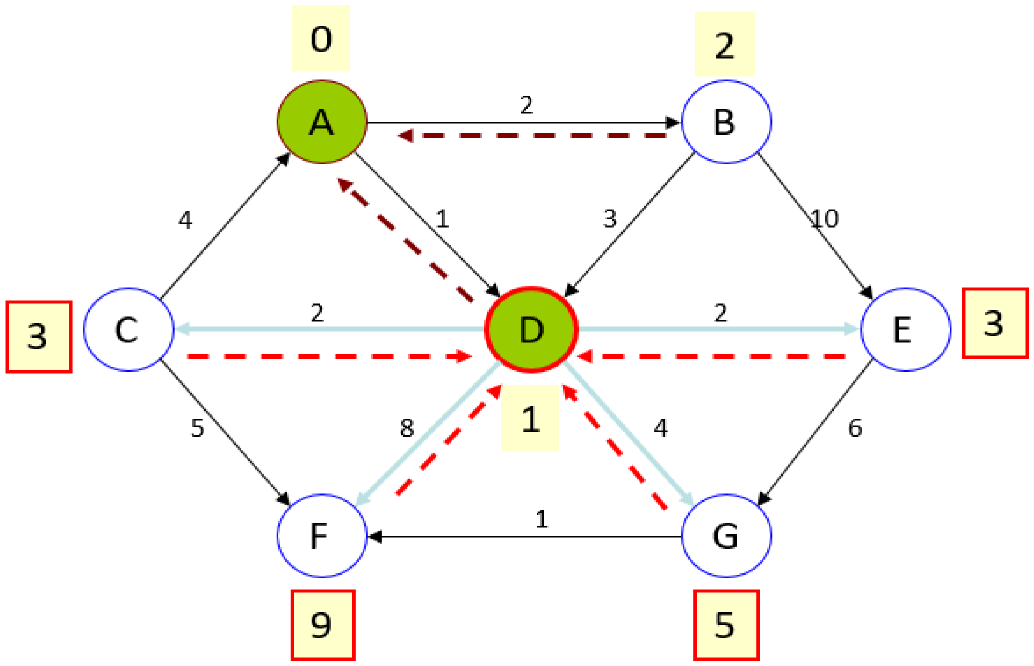
- Cost(B)=2
- Cost(C)=∞
- **Cost(D)=1**
- Cost(E)=∞
- Cost(F)=∞
- Cost(G)=∞



Dijkstra’s Algorithm Example

Pick vertex not in S with lowest cost and update neighbors

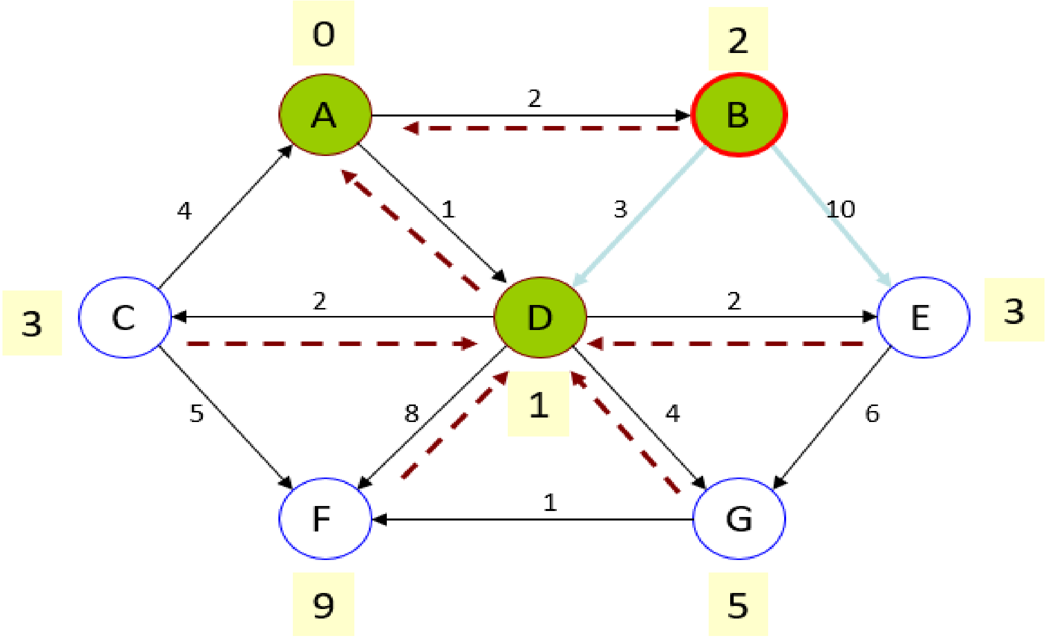
- S=[D,A]
- Cost(B)=2
- Cost(C)=1+2=3
- Cost(E)=1+2=3
- Cost(F)=1=8=9
- Cost(G)=1+4=5



Dijkstra's Algorithm Example

Pick vertex not in S with lowest cost and update neighbors

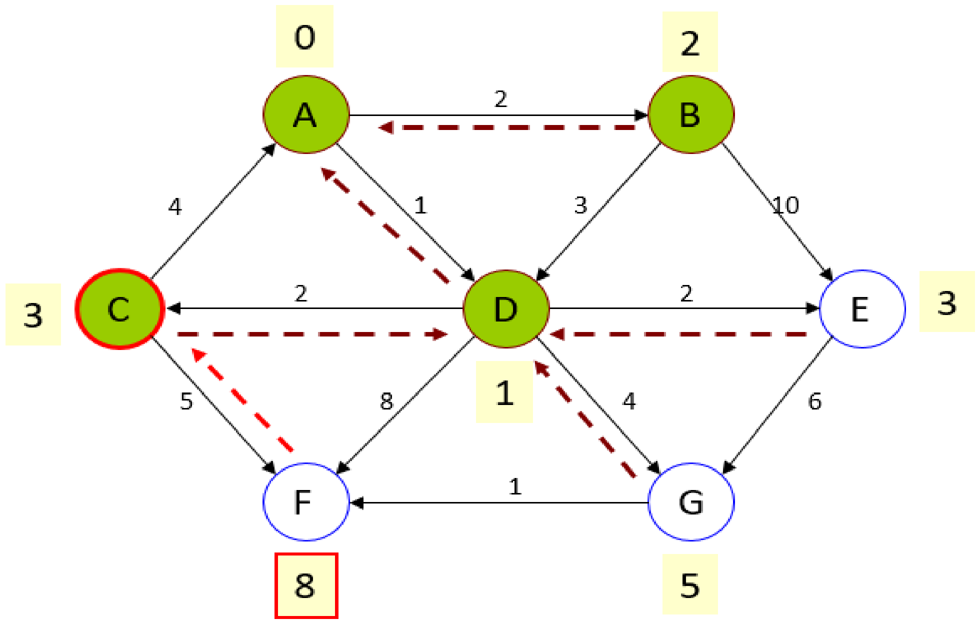
- S=[B,D,A]
- Cost(C)=3
- Cost(E)=3
- Cost(F)=9
- Cost(G)=5



Dijkstra's Algorithm Example

Pick vertex not in S with lowest cost and update neighbors

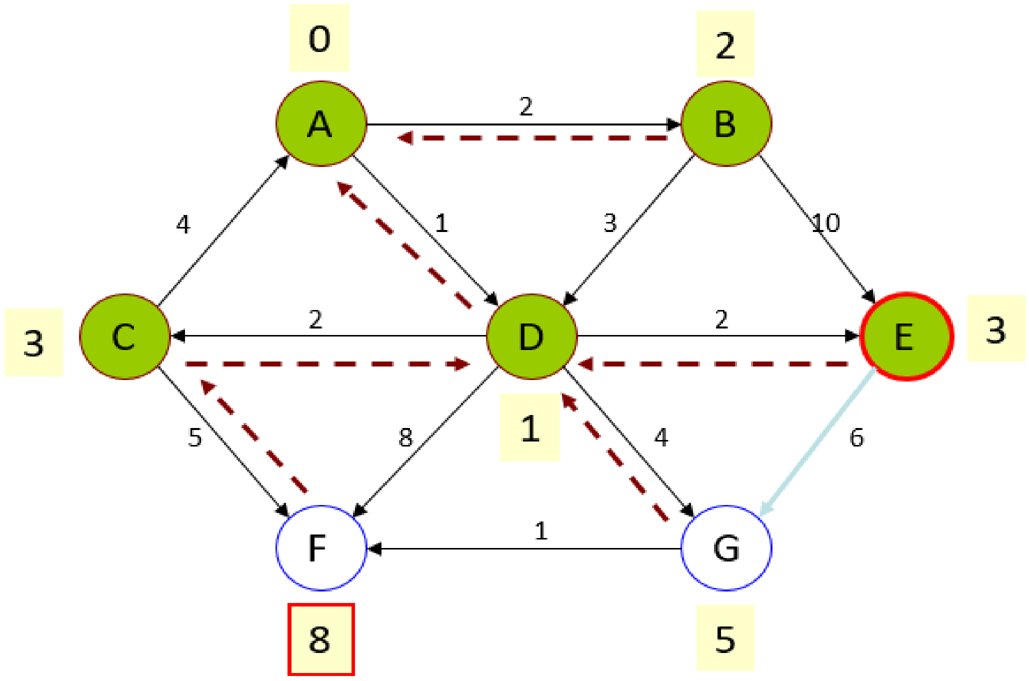
- S=[C,B,D,A]
- Cost(E)=3
- **Cost(F)=3+5=8**
- Cost(G)=5



Dijkstra’s Algorithm Example

Pick vertex not in S with lowest cost and update neighbors

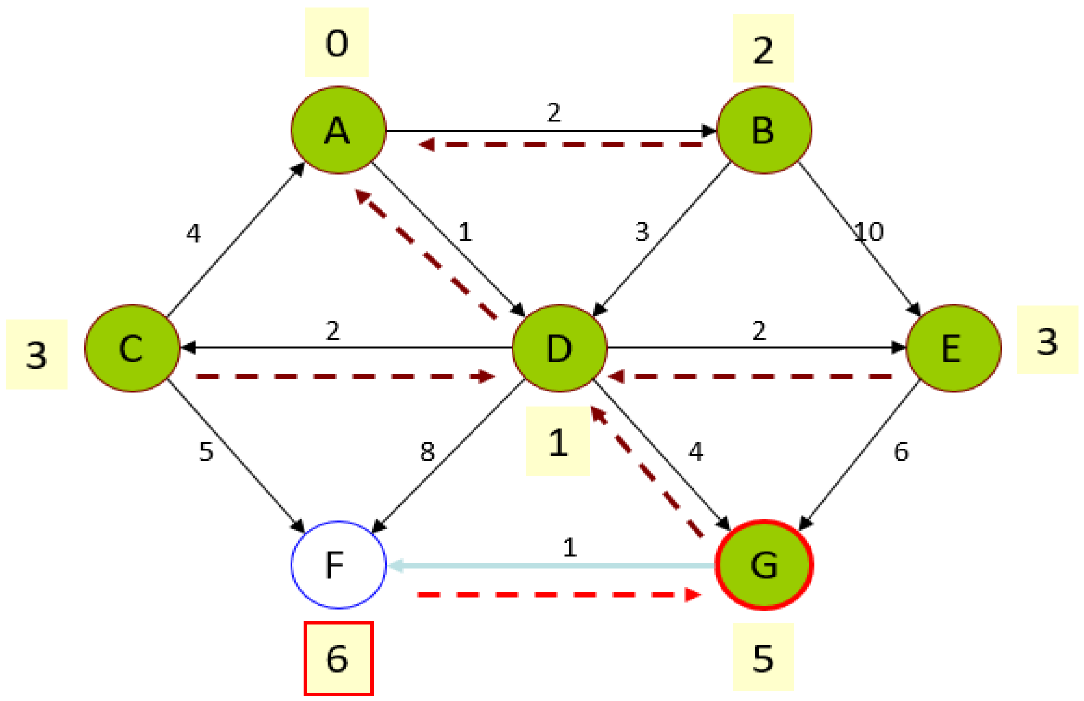
- S=[E,C,B,D,A]
- Cost(F)=8
- Cost(G)=5



Dijkstra’s Algorithm Example

Pick vertex not in S with lowest cost and update neighbors

- S=[G,E,C,B,D,A]
- **Cost(F)=5+1=6**

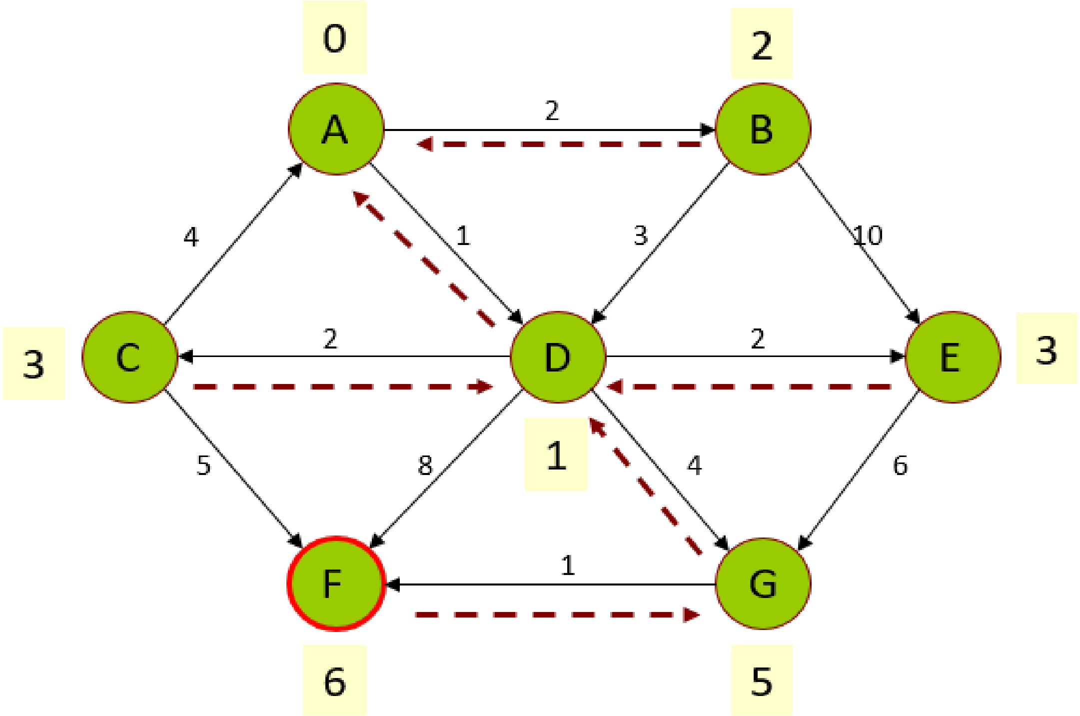


Dijkstra’s Algorithm Example

Pick vertex not in S with lowest cost and update neighbors

- S=[F,G,E,C,B,D,A]

- Shortest Paths from A:
- A->B=2
- A->C=3
- A->D=1
- A->E=3
- A->F=6
- A->G=5



Dijkstra's Algorithm

- For sparse graphs, (i.e. graphs with much less than $|V|^2$ edges) Dijkstra's is implemented most efficiently with a priority queue
  - Initialization: $O(|V|)$
  - while loop: $O(|V|)$ times
    - remove min-cost vertex from queue: $O(\log |V|)$
    - potentially perform $|E|$ updates on cost/previous
    - update costs in queue: $O(\log |V|)$
  - reconstruct path: $O(|E|)$
- Total runtime: $O(|V| \log |V| + |E| \log |V|)$
    - = **$O(|E| \log |V|)$**, because $|V| = O(|E|)$ if graph is connected
    - if a list is used instead of a queue: $O(|V^2| + |E|) = O(|V|^2)$

Dijkstra's Algorithm

Why Dijkstra Works?

- Hypothesis (**Optimal Substructure property**): A least cost path from X to Y contains least-cost paths from X to every node on the path to Y
- E.g.: if X->C1->C2->C3->Y is the least-cost path from X to Y, then
  - X->C1->C2->C3 is the least-cost path from X to C3
  - X->C1->C2 is the least-cost path from X to C2
  - X->C1 is the least-cost path from X to C1

Dijkstra’s Algorithm

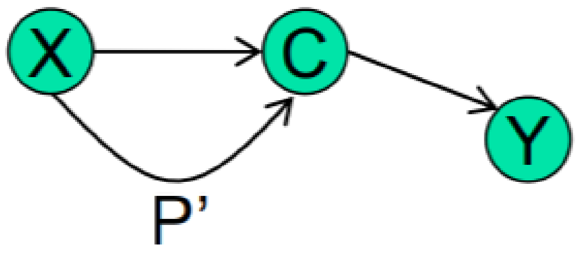
Proof by Contradiction:

Assume hypothesis is false: Given a least-cost path P from X to Y that goes through C, there is a better path P’ from X to C than the one in P

Show a contradiction:

- But we could replace the subpath from X to C in P with this lesser-cost path P’
- The path cost from C to Y is the same
- Thus we now have a better path from X to Y
- But this violates the assumption that P is the least-cost path from X to Y

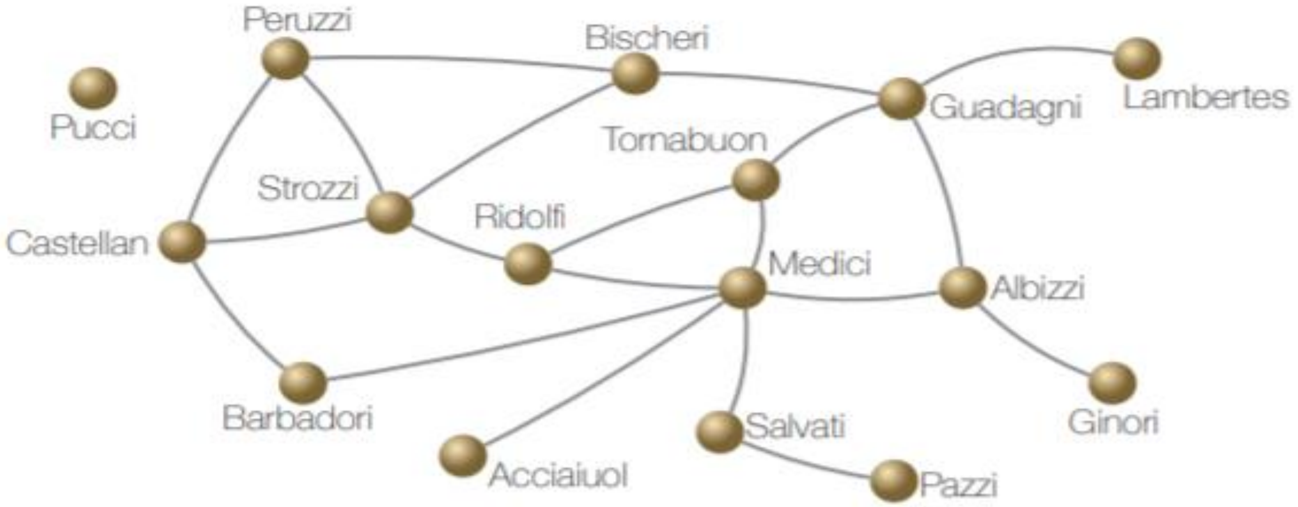
Therefore, the original hypothesis must be true!



Centrality Metrics

Centrality Metrics

- Measure which nodes are **important, influential or popular** in a network based on the topological structure



- Why were the Medici an important family in 15th century Florence?

Centrality Metrics

- Different notions of node centrality:
  - Degree — well connectedness
  - Betweenness — criticality for connectedness
  - Closeness — short distances to the rest of the graph
  - Eigenvector — importance

Degree Centrality

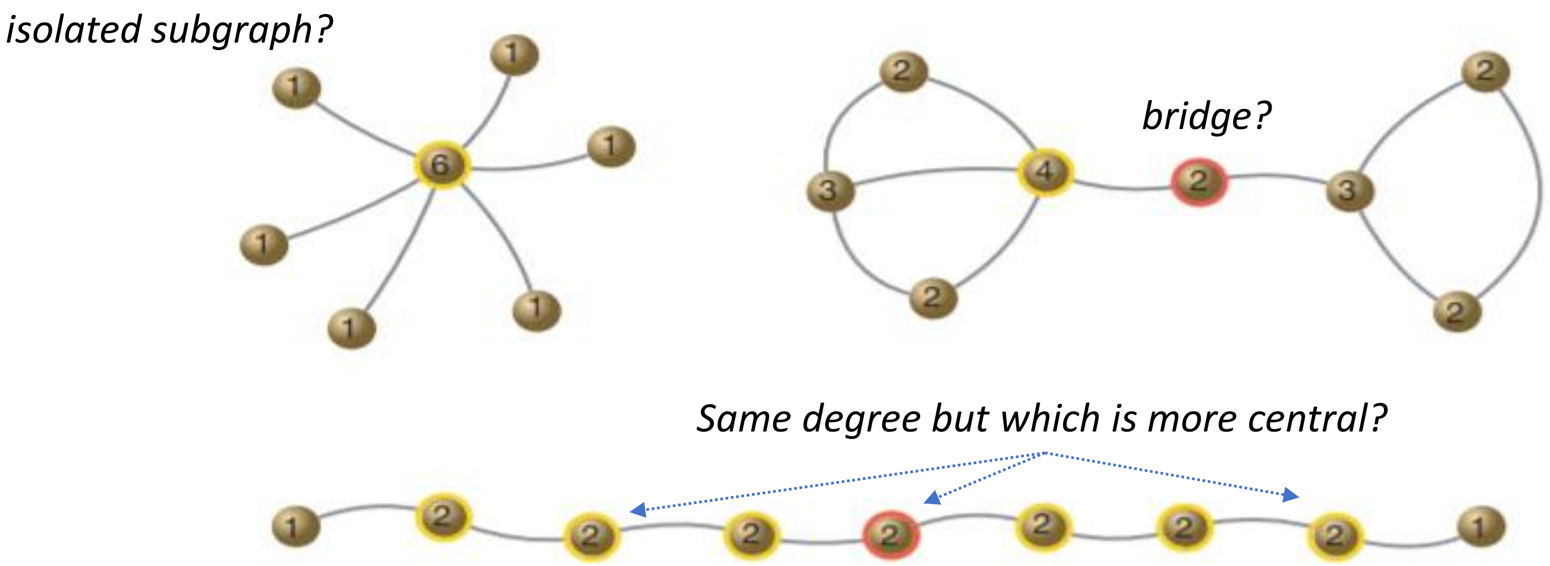
- The node with the **most connections** is the most important according to this metric
- For a graph G = (V, E), the degree centrality of a given node v is:

$$\mathbf{C_D(v) = degree(v)}$$

- For a directed network we have in- and out-degree centralities
- Appropriate for some settings:
  - Social network example: a node (user) of high degree might be thought as influential
  - Citation networks: choose papers with may citations (in-degree centrality) when doing literature surveys

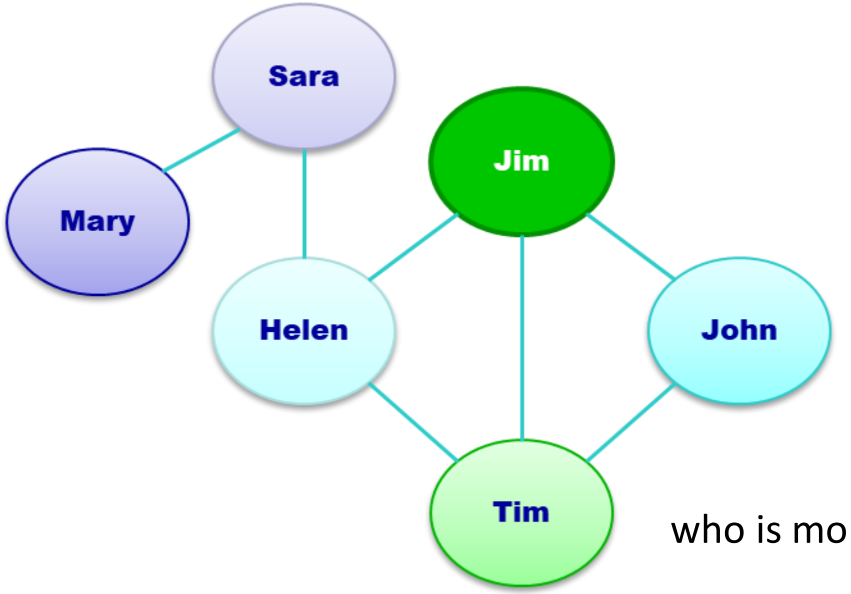
Degree Centrality

- Problems with degree-based centrality:



- Node degree captures connectivity to adjacent nodes but ignores distances to other nodes in the graph

Degree Centrality Example



who is more important?

Node	Degree
Mary	1
Sara	2
Helen	3
Jim	3
Tim	3
John	2

Closeness Centrality

- An important node in a **central** position, close to the rest of the graph
  - Important nodes require fewer number of edges to transfer information to all other nodes

- Define closeness of node u as the inverse of the average of the shortest path lengths between node u and every other node in the graph

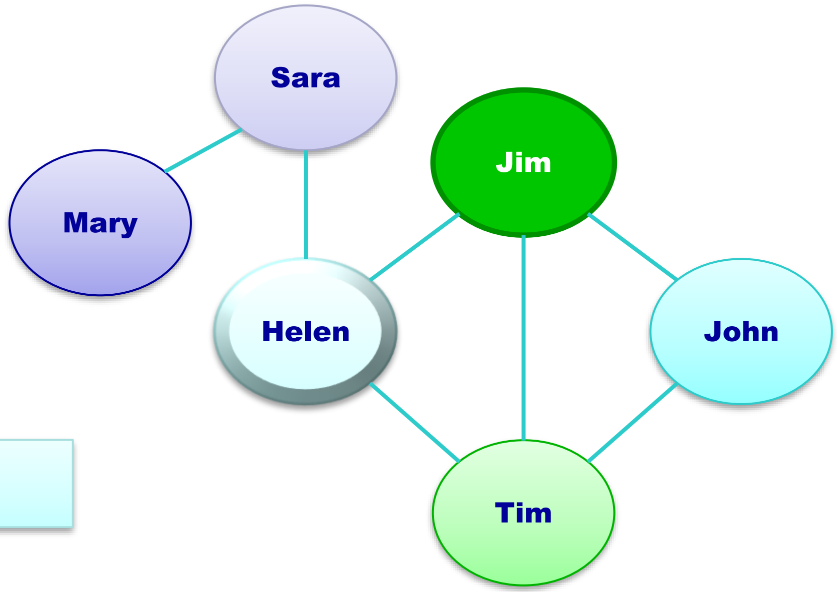
$$\mathbf{C_C(u) = \frac{n-1}{\sum_i d(u,i)}}$$

- where d(u,i) = length of shortest path between nodes u and i

Closeness Centrality Example

- Lengths of **shortest paths** from Helen to all other nodes
- Helen->Mary : 2
- Helen->Sara: 1
- Helen->Jim: 1
- Helen->Tim: 1
- Helen->John: 2

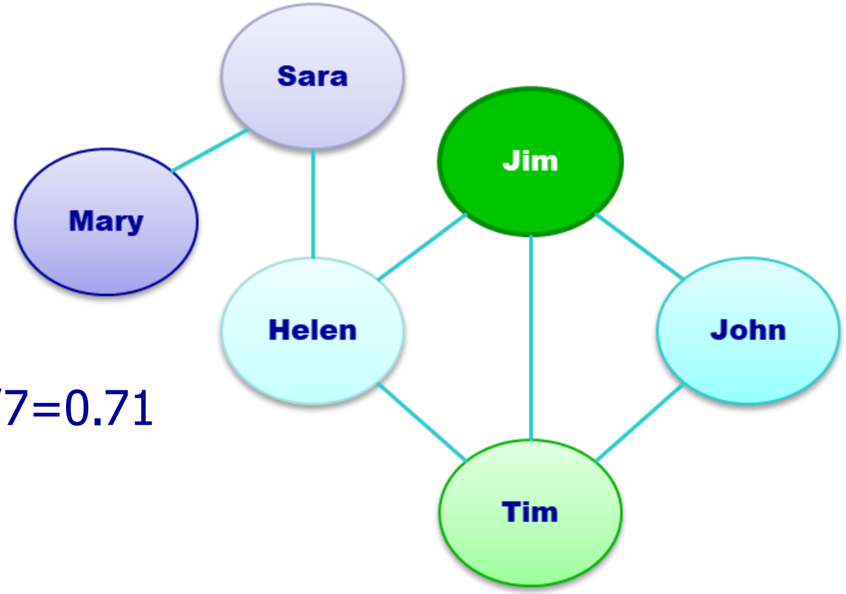
AVG Length = 7/5 = 1.4



A node is deemed "central" if this number is small

Closeness Centrality Example

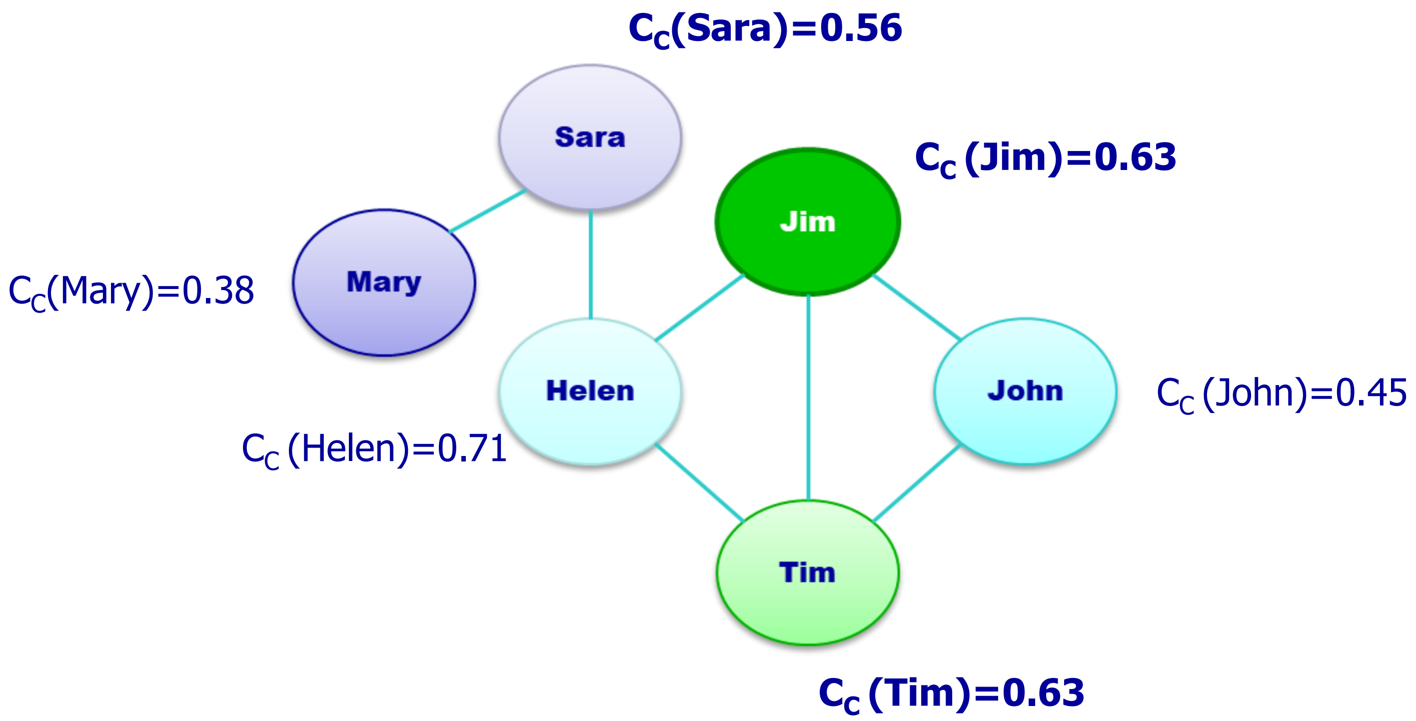
- C_C = **inverse** of avg distance
- Small avg distance →high closeness centrality



$C_C(\text{Helen})=5/7=0.71$

Closeness Centrality Example

- Note that Jim & Tim are more central than Sara
- However, removal of Sara **bisects** the graph



Betweenness Centrality

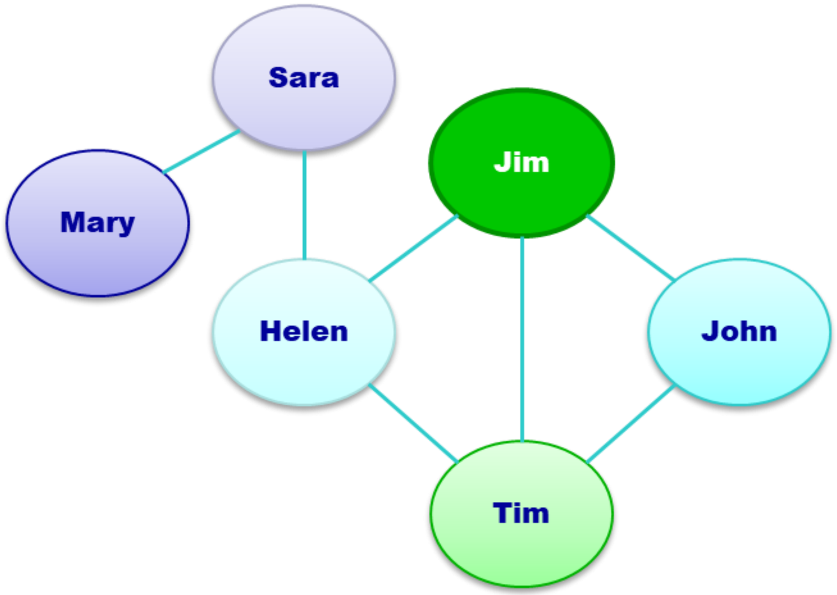
- Degree & closeness-based centrality are not able to capture the ability of a node in a graph to act as a bridge between different components
- Calculate **betweenness** of node u based on the fraction of all pairwise shortest paths that go through u

$$\mathbf{C_B(u) = \sum_{all\ pairs\ i,j} \frac{g_{ij}(u)}{g_{ij}}}$$

- Where:
  - g_{ij} = total number of shortest paths between nodes i, j
  - $g_{ij}(u)$ = number of shortest paths between i, j that go through u

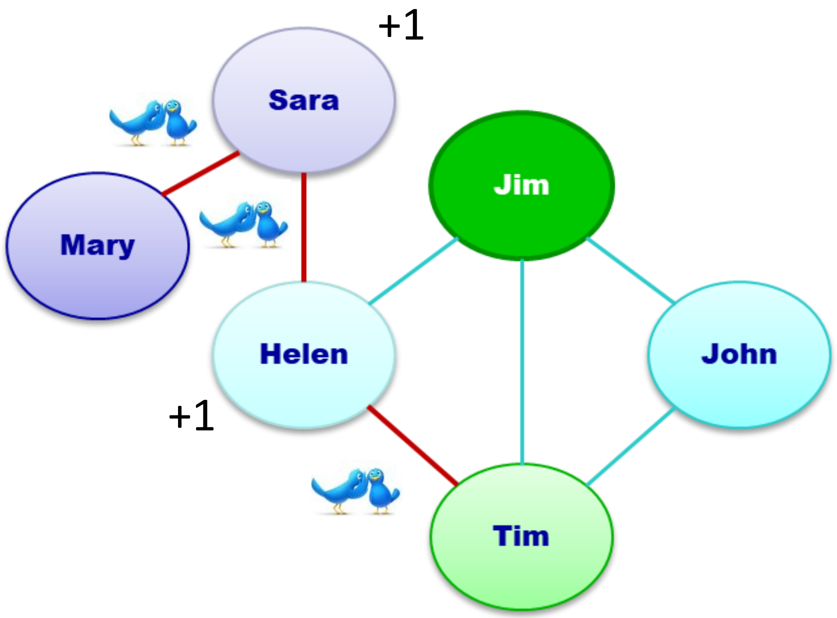
Betweenness Centrality Example

- **BC** want to capture importance of nodes in information passing
- CC measures inverse of avg path length to all other nodes
  - Some of these paths are not as important if alternative routes exist



Betweenness Centrality Example

- Shortest path: fastest method to pass a message across
- Mary sends a message to Tim through Sara & Helen
- Sara & Helen are **rewarded** for their contribution

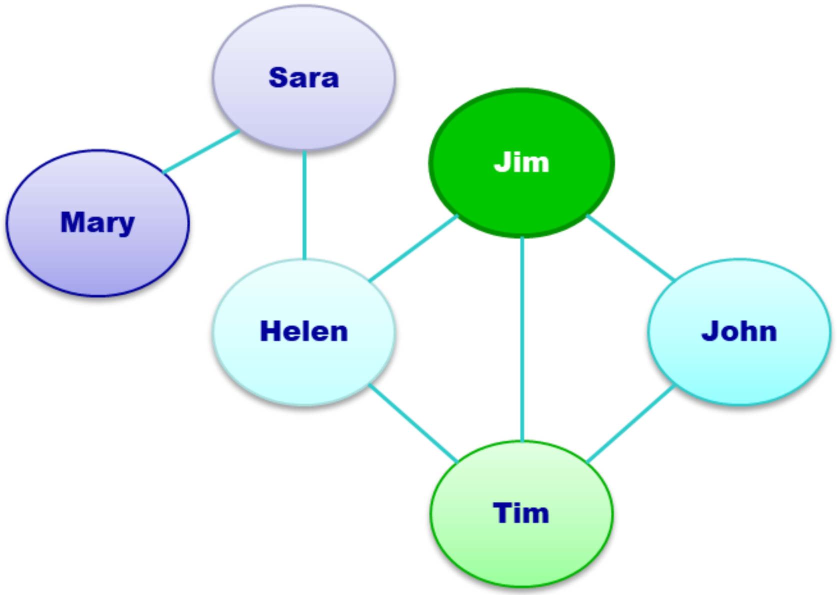


Betweenness Centrality Example

- **BC** = number of shortest paths from all vertices to all others that pass through that node

Note:

- Only consider paths with more than 2 nodes (no direct edges)
- When multiple shortest paths exist, split rewards



Betweenness Centrality Example

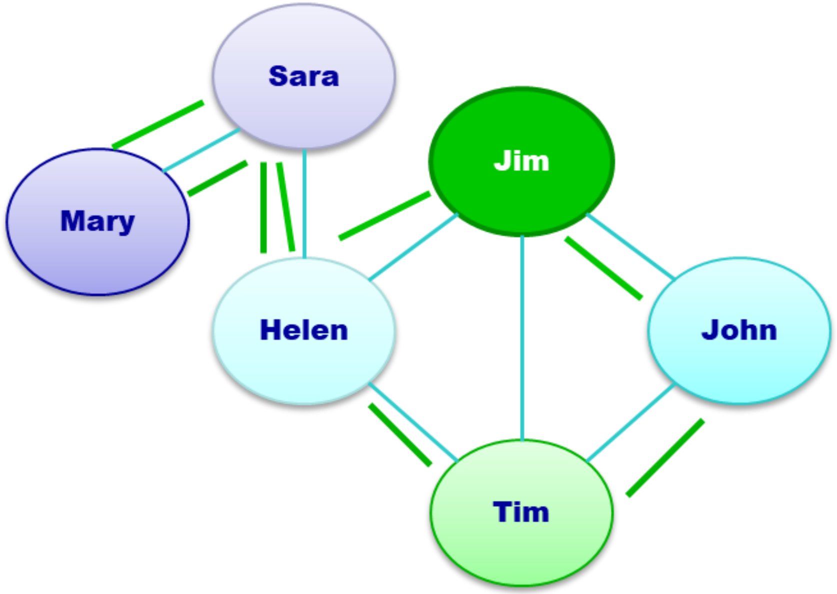
- Mary sends message to John

SP1: Mary→Sara→Helen→Jim→John

SP2: Mary→Sara→Helen→Tim→John

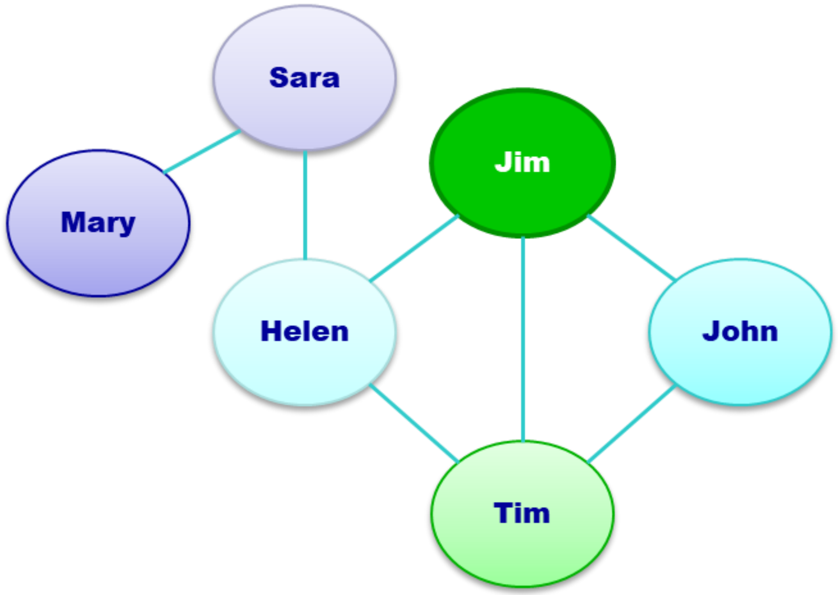
Rewards:

- Sara: +.5 + .5
- Helen: +.5 + .5
- Jim: +.5
- Tim: +.5



Betweenness Centrality Example

Node	Betweenness Centrality
Mary	0
Sara	4
Helen	6
Jim	1.5
Tim	1.5
John	0



Centrality Metrics in Directed Graphs

- Degree, betweenness and closeness centrality definitions extend naturally to directed graphs
- Out-degree centrality (based on out-degree)
- In-degree centrality (based on in-degree)
- Betweenness centrality of a node considers the fraction of all pairwise shortest directed paths that go through it
- In-closeness (based on path lengths from all other nodes to the given node)
- Out-closeness (based on path lengths from the given node to all other nodes)