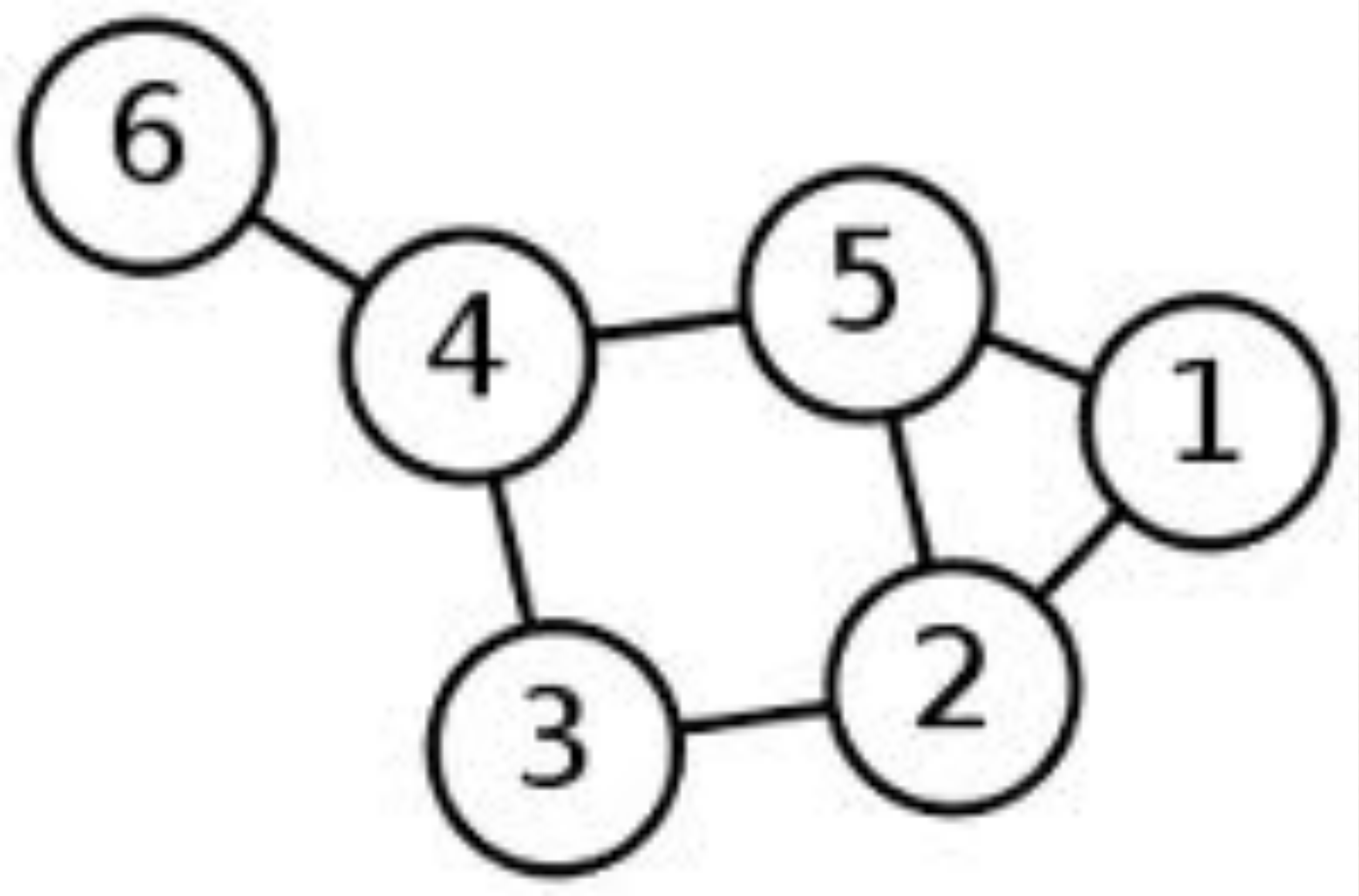
Graph Theory

Chapter 9

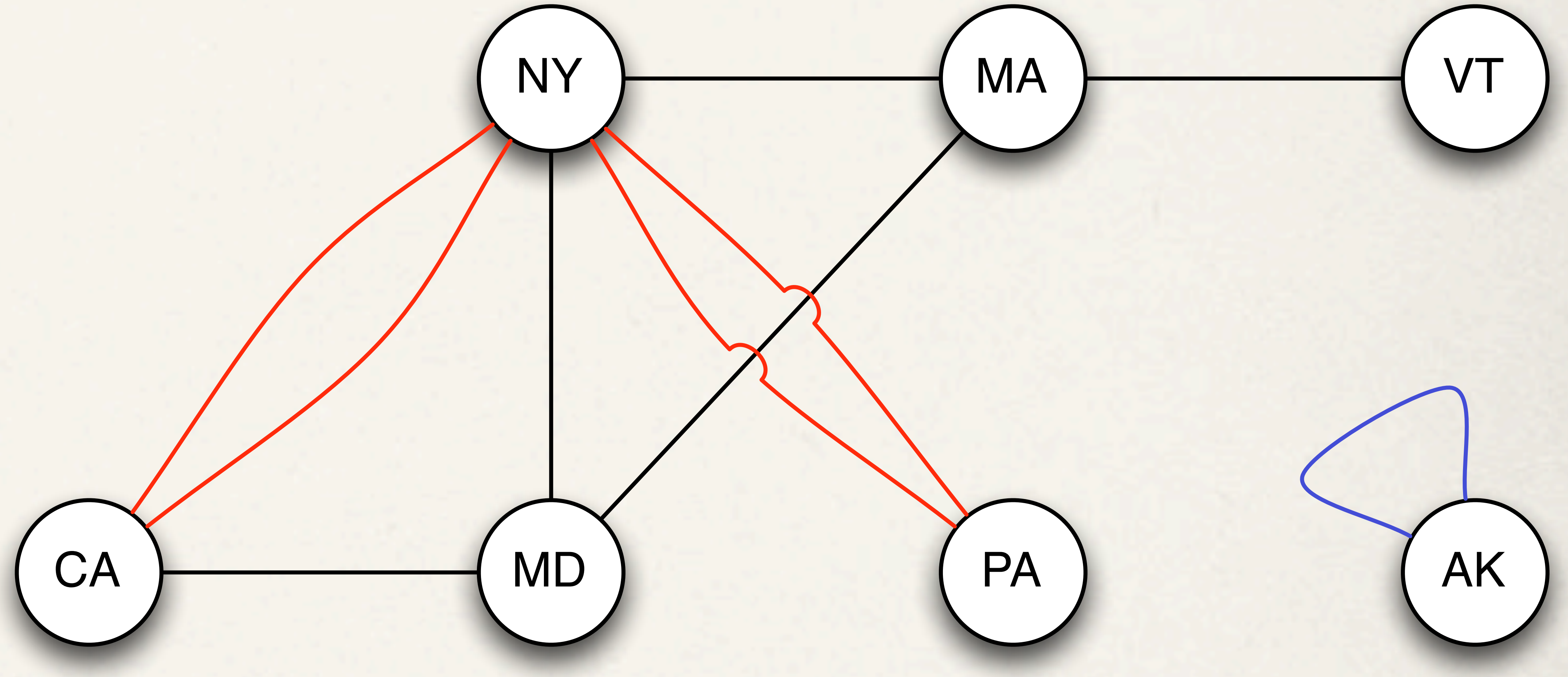
Graphs

- A **graph** consists of **nodes** and **edges**
  - The set of all nodes (or **vertices**) in a graph is *V*
  - The set of all edges in a graph is *E*
  - Each edge **connects** one or two vertices (called its' **endpoints**)
  - A graph is formally represented (*V*, *E*)
  - We will (usually) stick to **simple graphs**



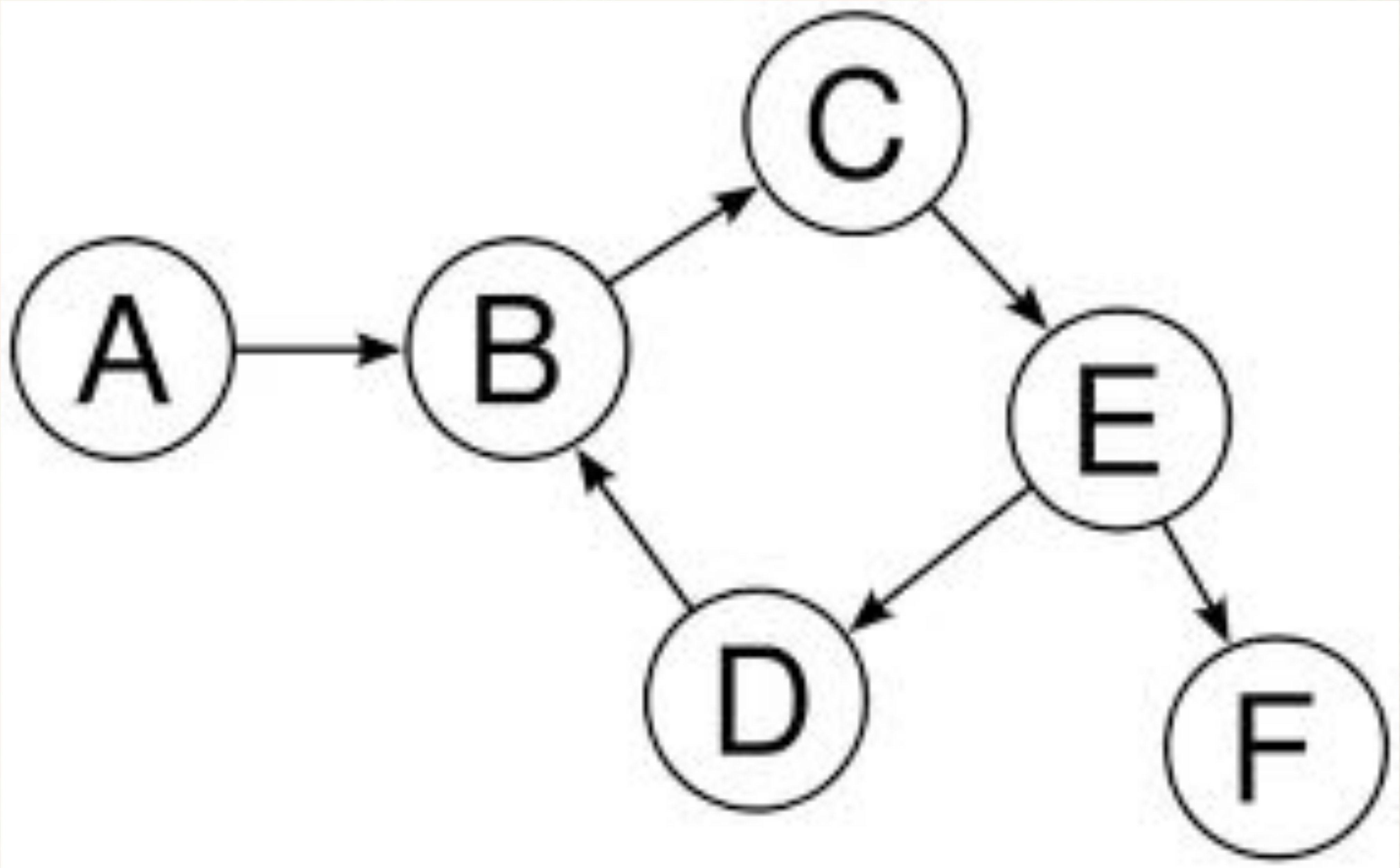
Simple Graphs

- No **loops**
  - Endpoints of an edge are distinct
- No **multiple edges**
  - between the same two vertices

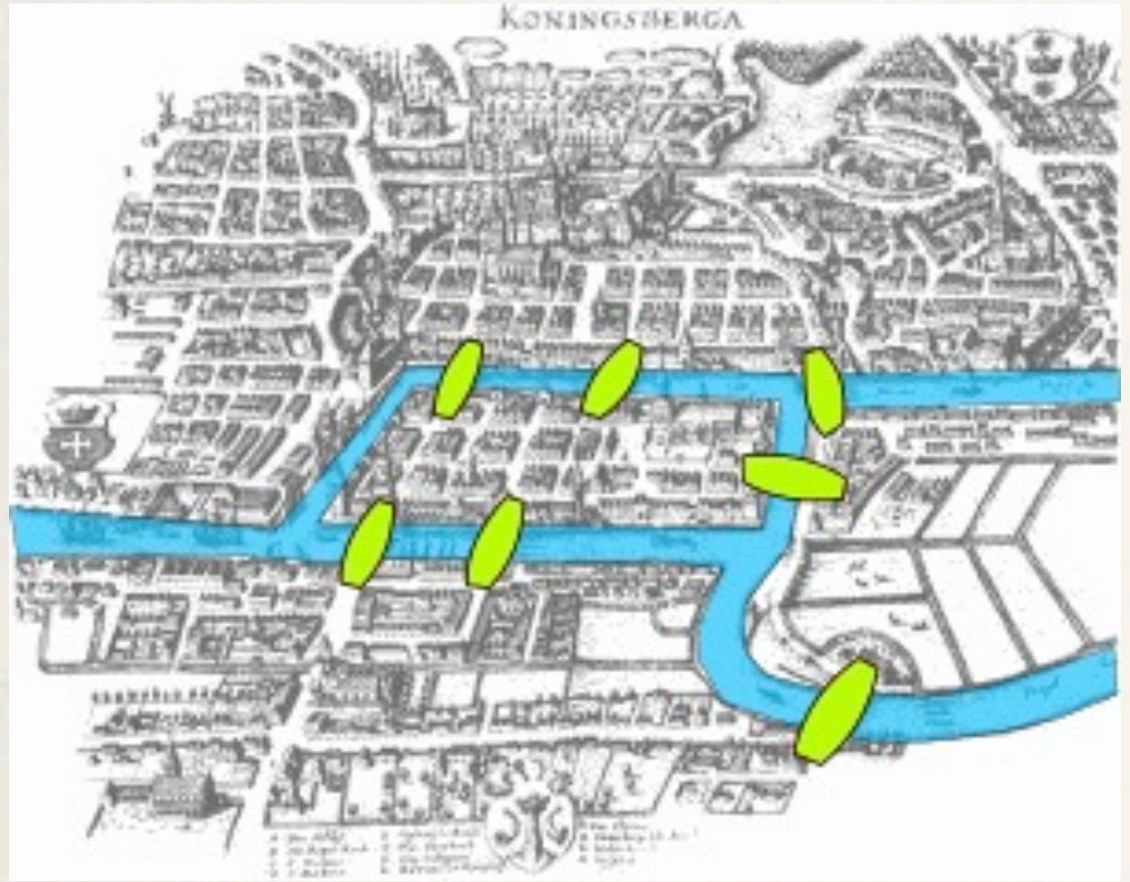


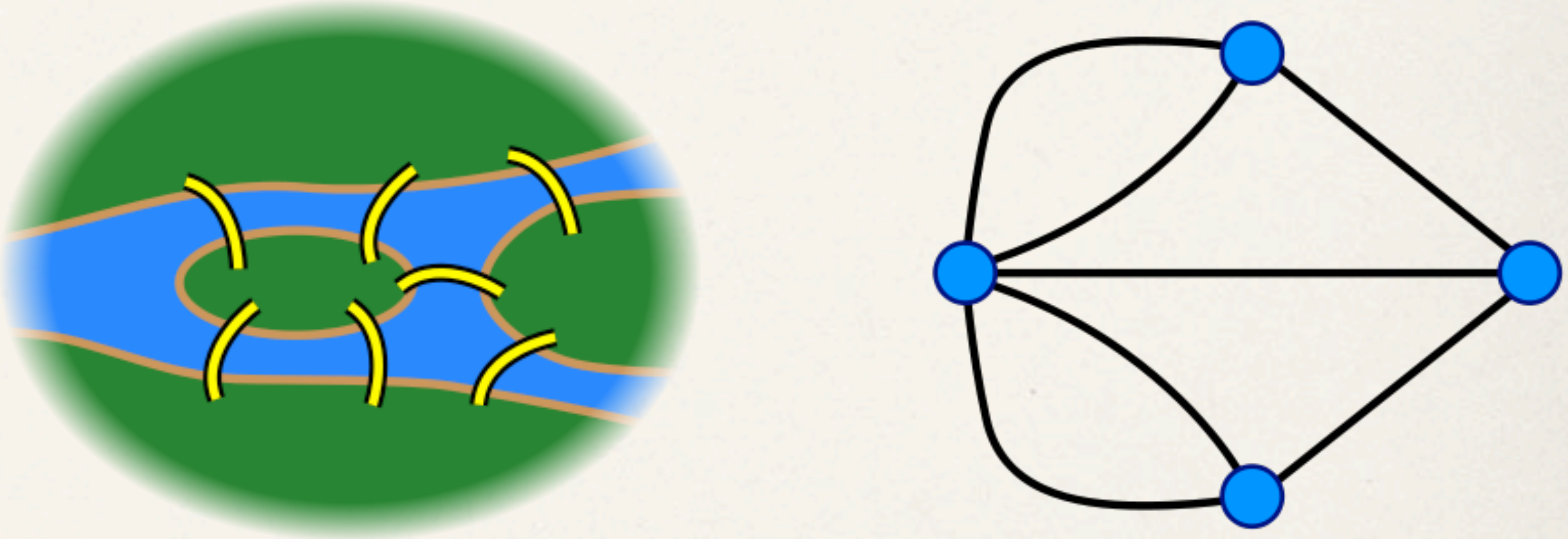
Directed Graphs

- A **directed edge**, or **arc** connects two vertices, but has a **start** and **end**
    - Formally an arc (u, v) starts at u and ends at v
- A graph with directed edges is a **directed graph** or **digraph**

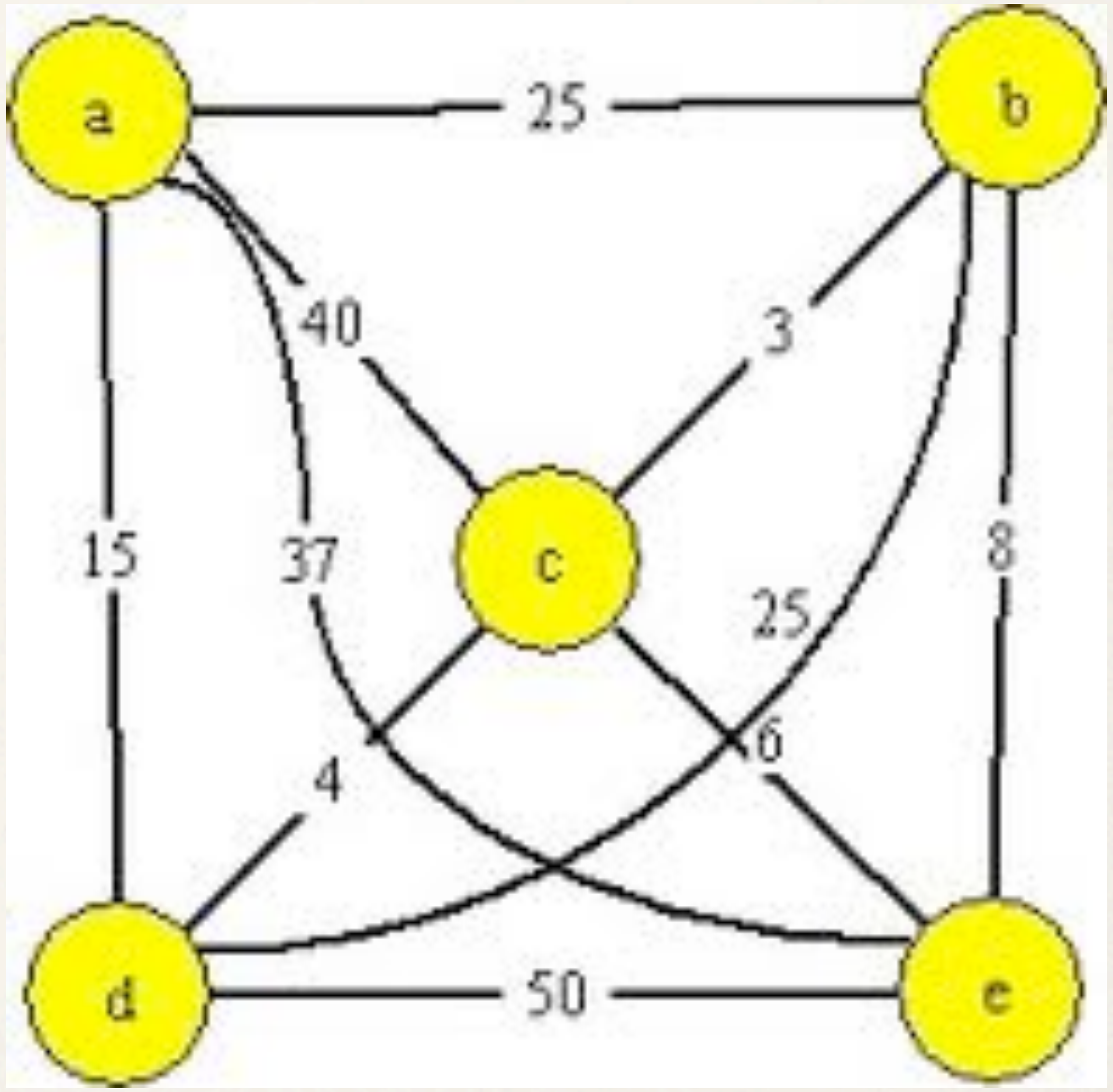


Seven Bridges of Königsberg



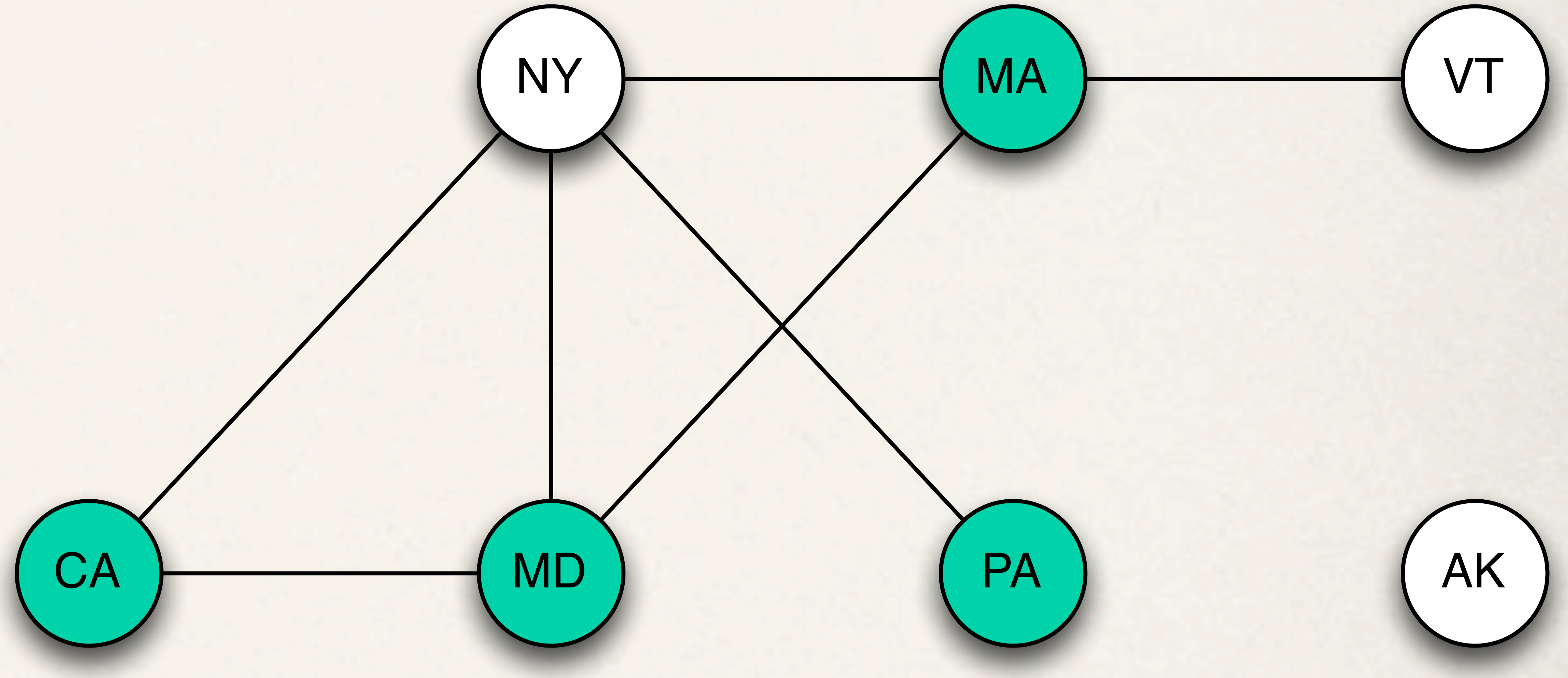


Traveling Salesman Problem



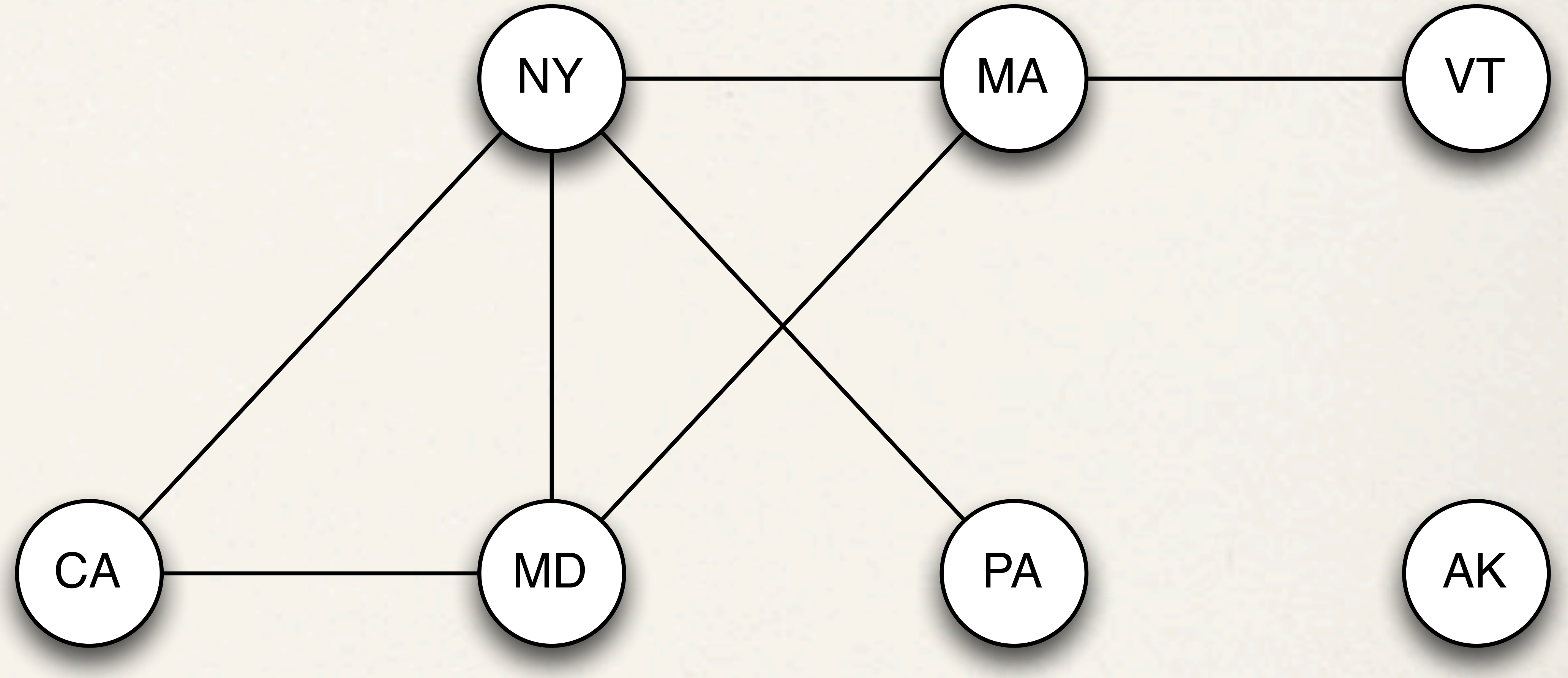
Terminology

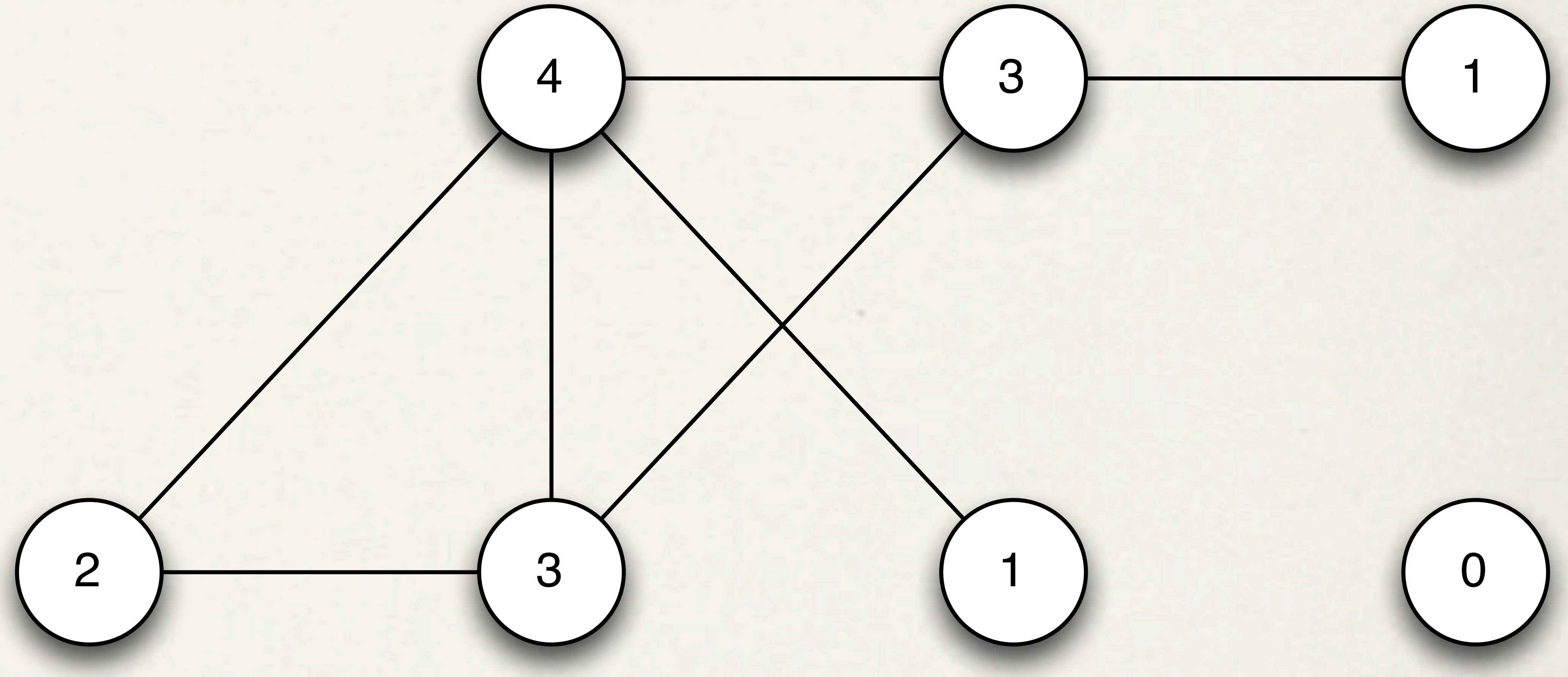
- Two vertices u and v in an undirected graph G are **neighbors** if there exists an edge e in G which connects u, and v
  - Equivalently, u and v are said to be **adjacent** if there exists an edge which connects them
- The set of all neighbors of u (u's **neighborhood**) is $N(u)$
- This is defined similarly for sets of vertices



Degree

- If an edge *e* has vertex *u* as one of its' endpoints, *e* is **incident with** *u*
- The **degree** of a node *u*, deg(*u*), is the count of the number of edges incident with *u*
- A node with degree 0 is **isolated**



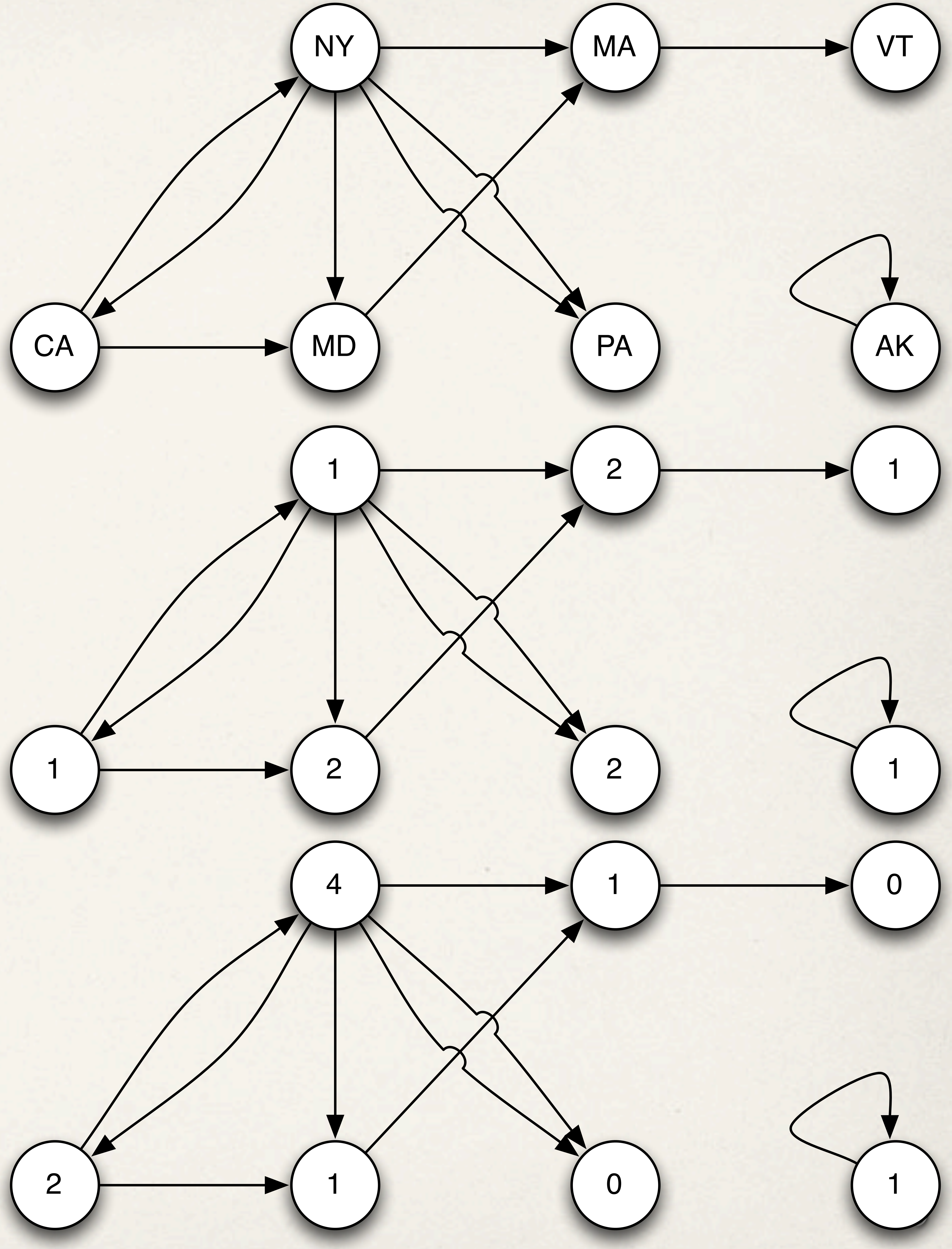


In/Out -Degree

- In directed graphs, separate degrees
  - **In-Degree**: # of edges ending at *u*

    $\deg^-(u)$
  - **Out-Degree**: # of edges starting at *u*
    $\deg^+(u)$
- Vertices with in-degree 0 are **sources**
- Vertices with out-degree 0 are **sinks**

$$\sum_{v\in V}\deg^-(v)=\sum_{v\in V}\deg^+(v)=|E|$$



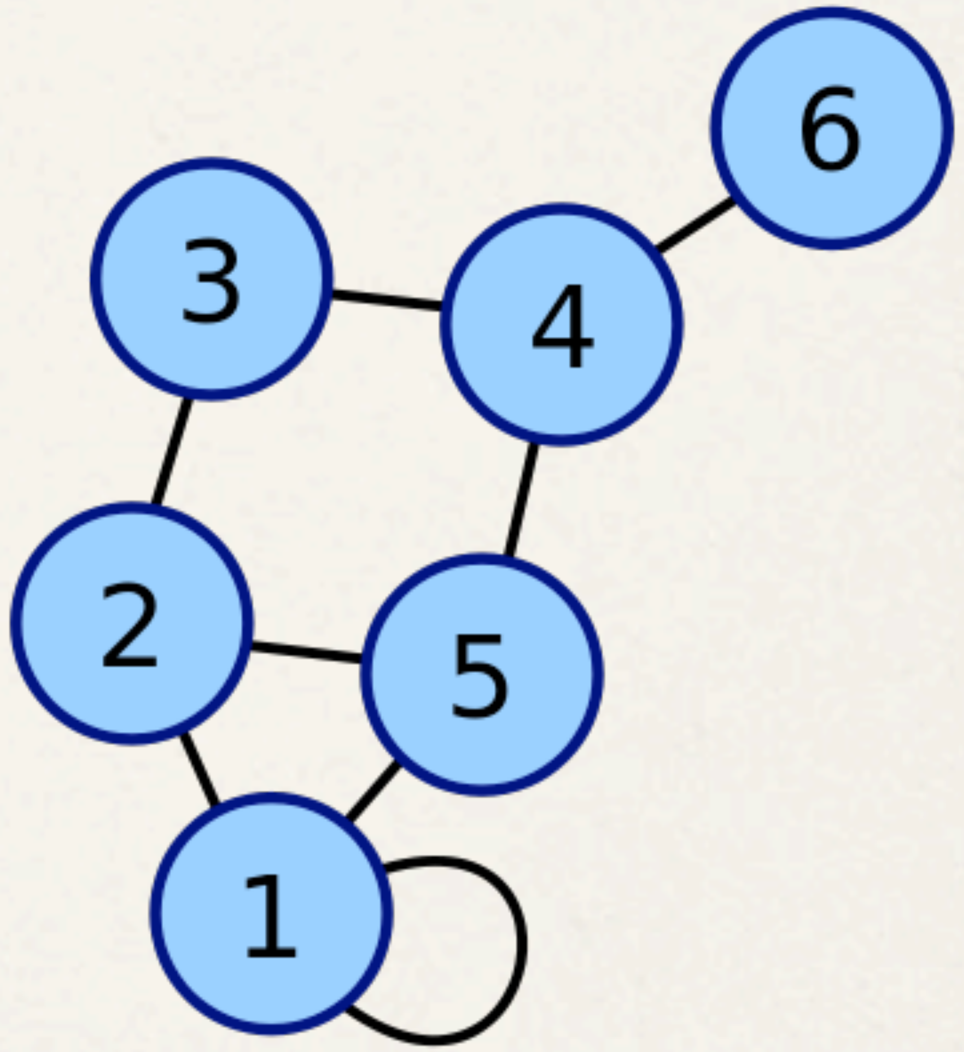
Representations

Graph ADT

- V = set of vertices
  E = set of edges
- Three operations
  - `getDegree(u)`
  - `getAdjacent(u)`
  - `isAdjacentTo(u, v)`

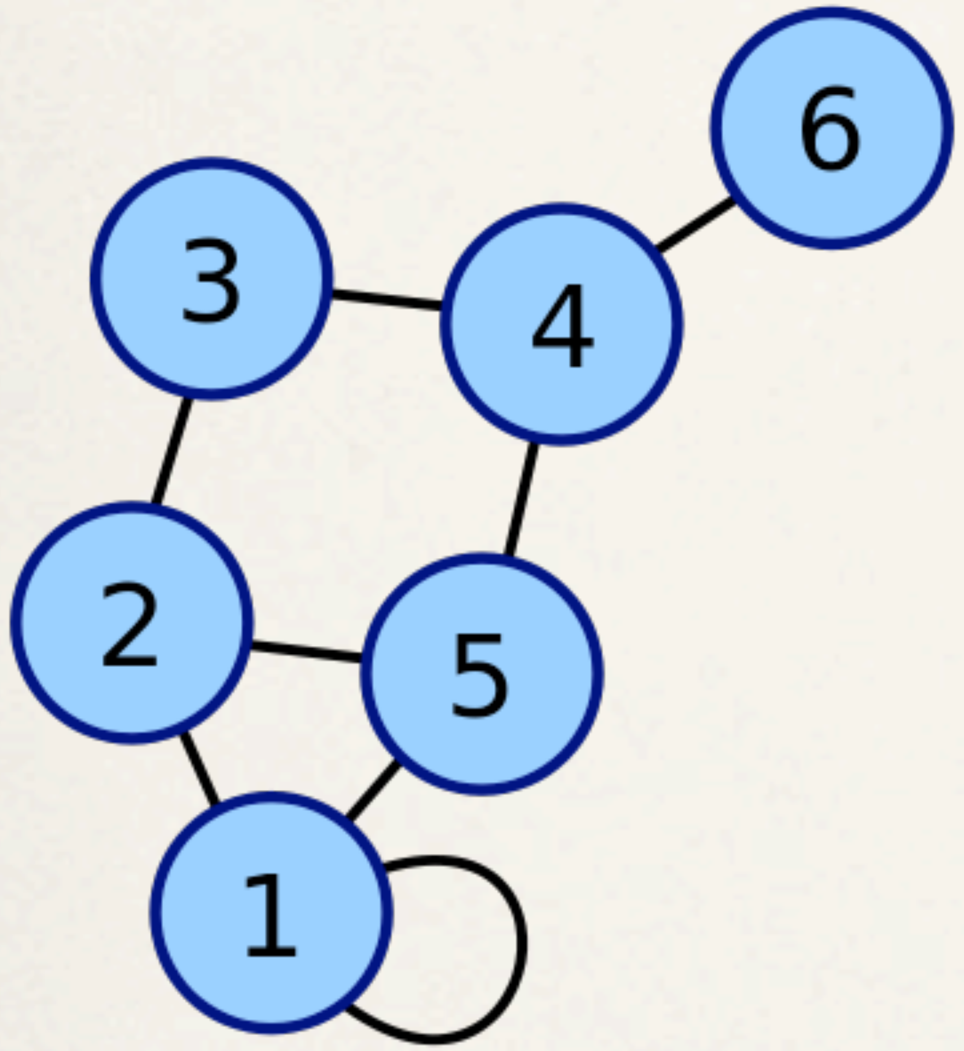
Graph Representations

- **Adjacency list**
  - For each vertex, list the adjacent vertices
  - Good for **sparse** graphs with few edges
- **Adjacency matrix**
  - 2D matrix of 1s and 0s
    - 1 iff there is an edge from *i* to *j*
  - Good for **dense** graphs with many edges



Graph Representations

Graph



Adjacency List

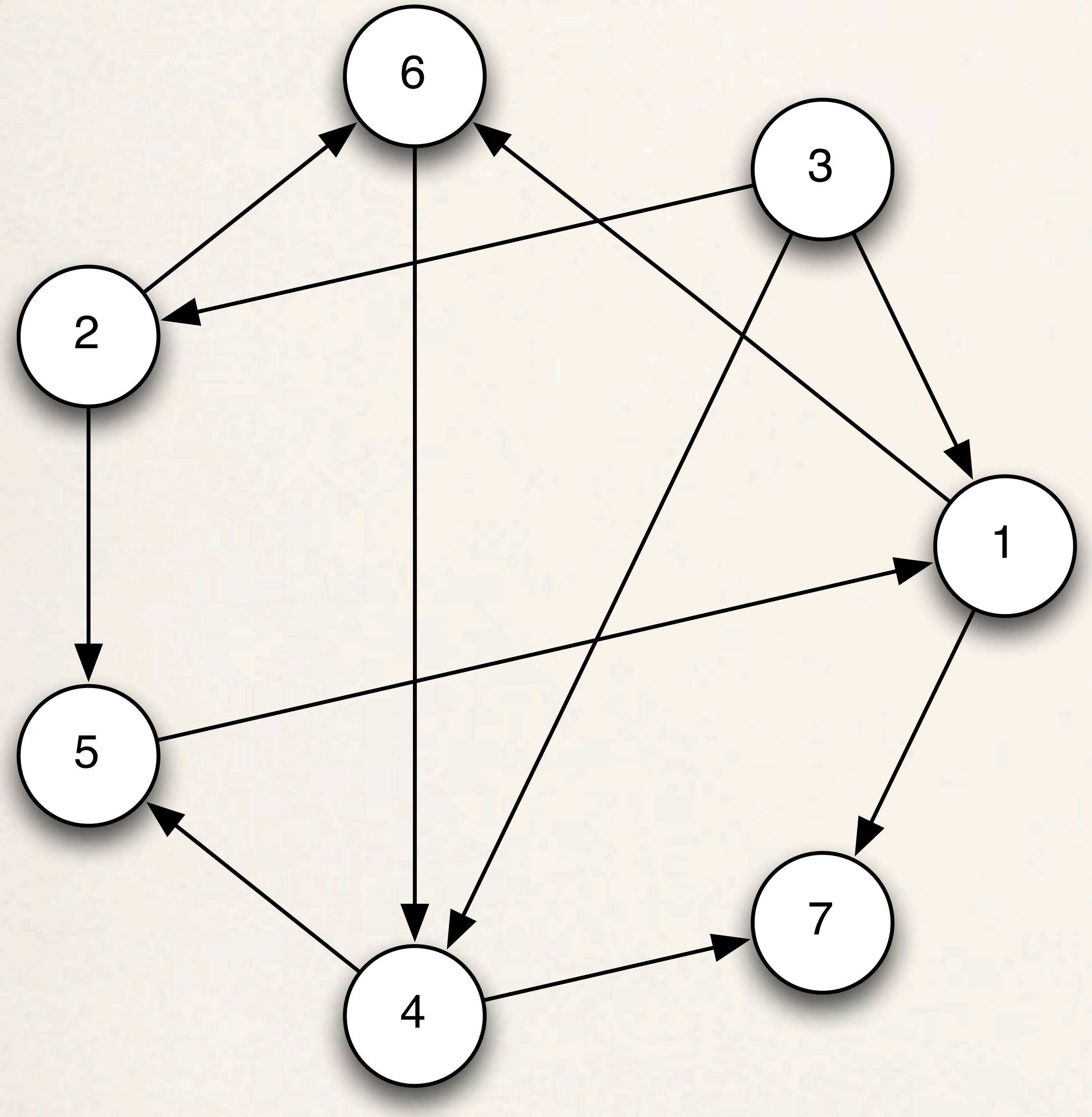
V	Adj.
1	1,2,5
2	1,3,5
3	2,4
4	3,5,6
5	1,2,4
6	4

Adjacency Matrix

V	1	2	3	4	5	6
1	1	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

Directed Graphs

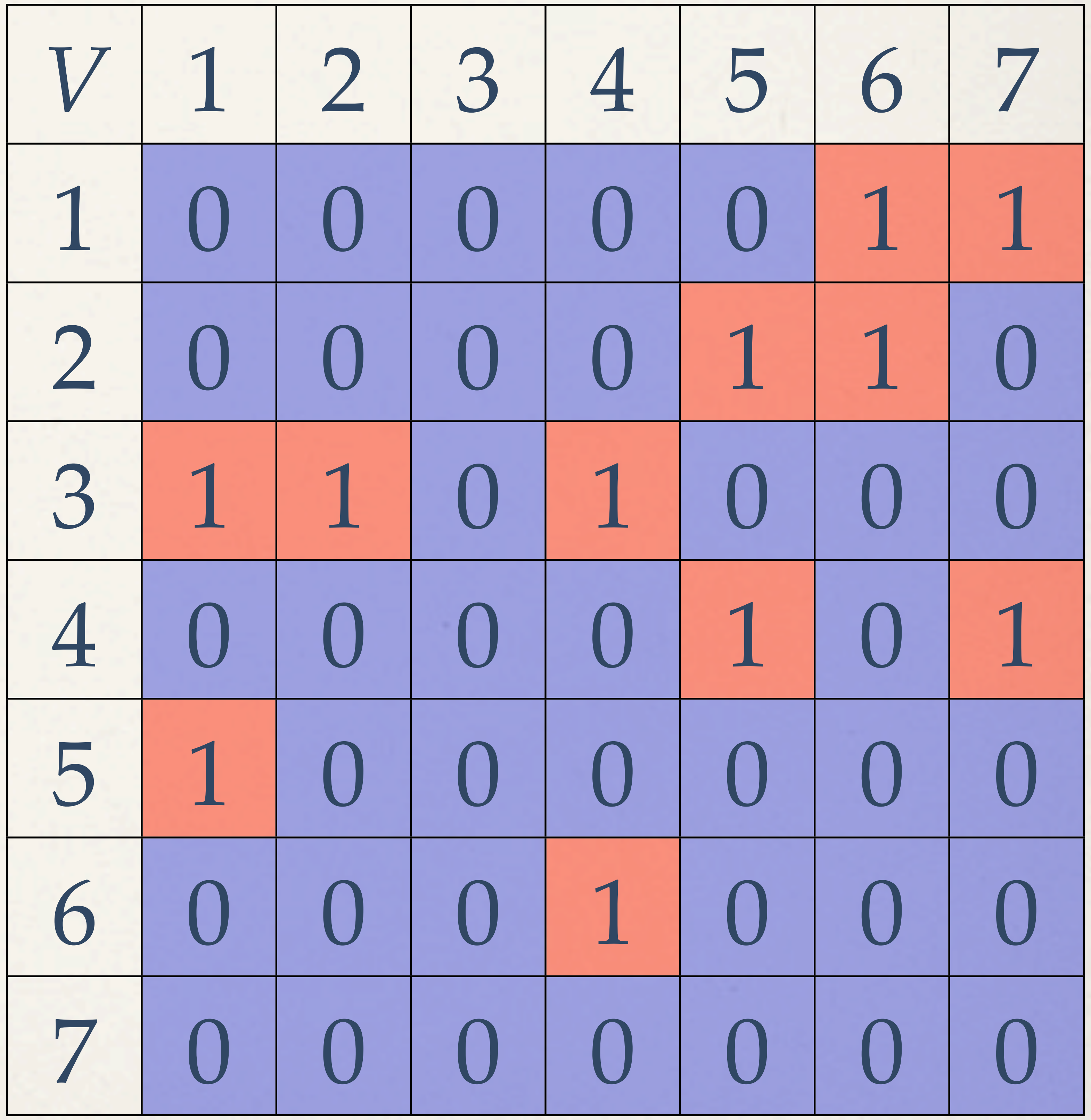
Graph



Adjacency List

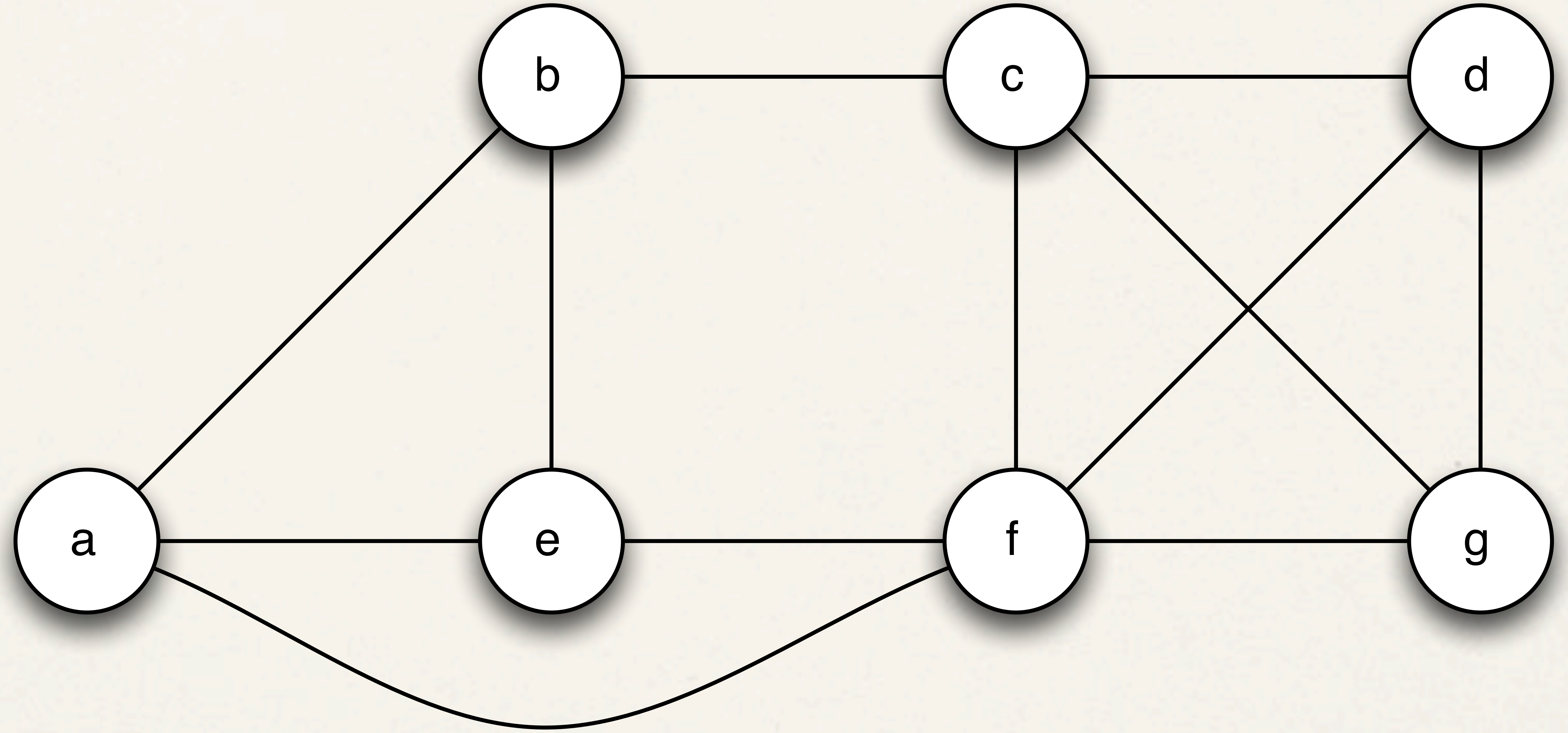
V	Term
1	6,7
2	5,6
3	1,2,4
4	5,7
5	1
6	4
7	

Adjacency Matrix

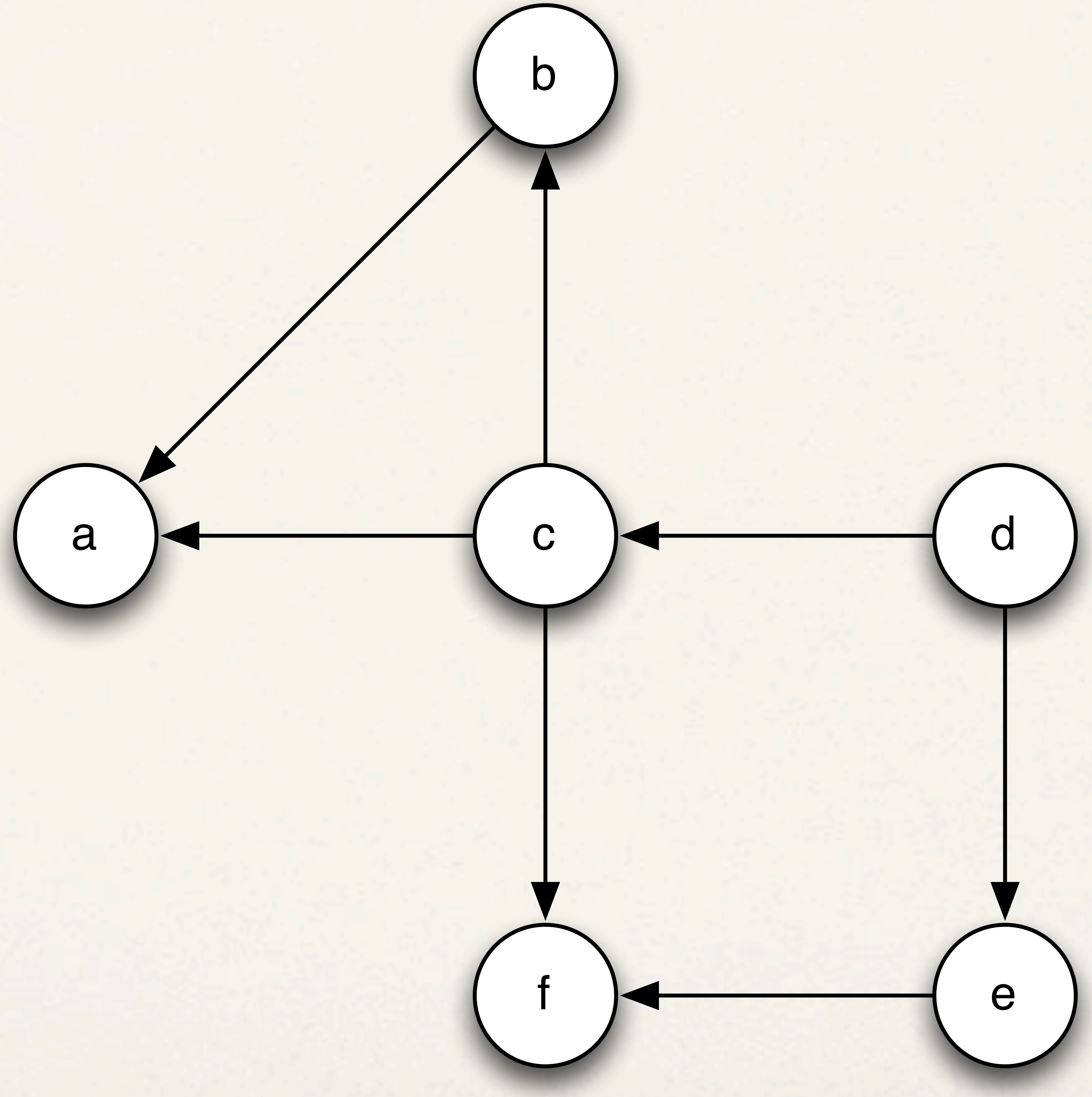


V	1	2	3	4	5	6	7
1	0	0	0	0	0	1	1
2	0	0	0	0	1	1	0
3	1	1	0	1	0	0	0
4	0	0	0	0	1	0	1
5	1	0	0	0	0	0	0
6	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0

Practice



Practice

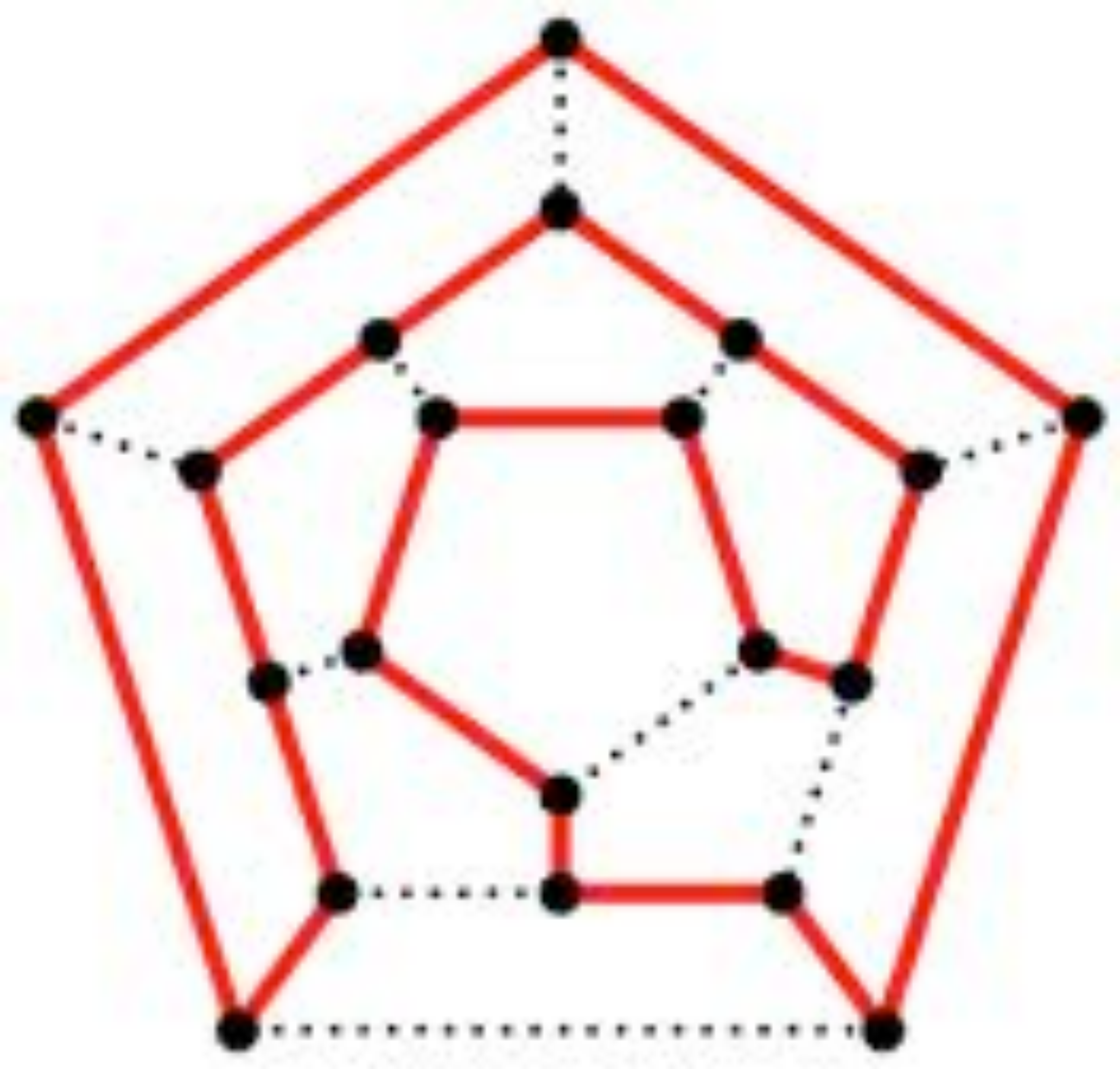


Performance of List vs Matrix

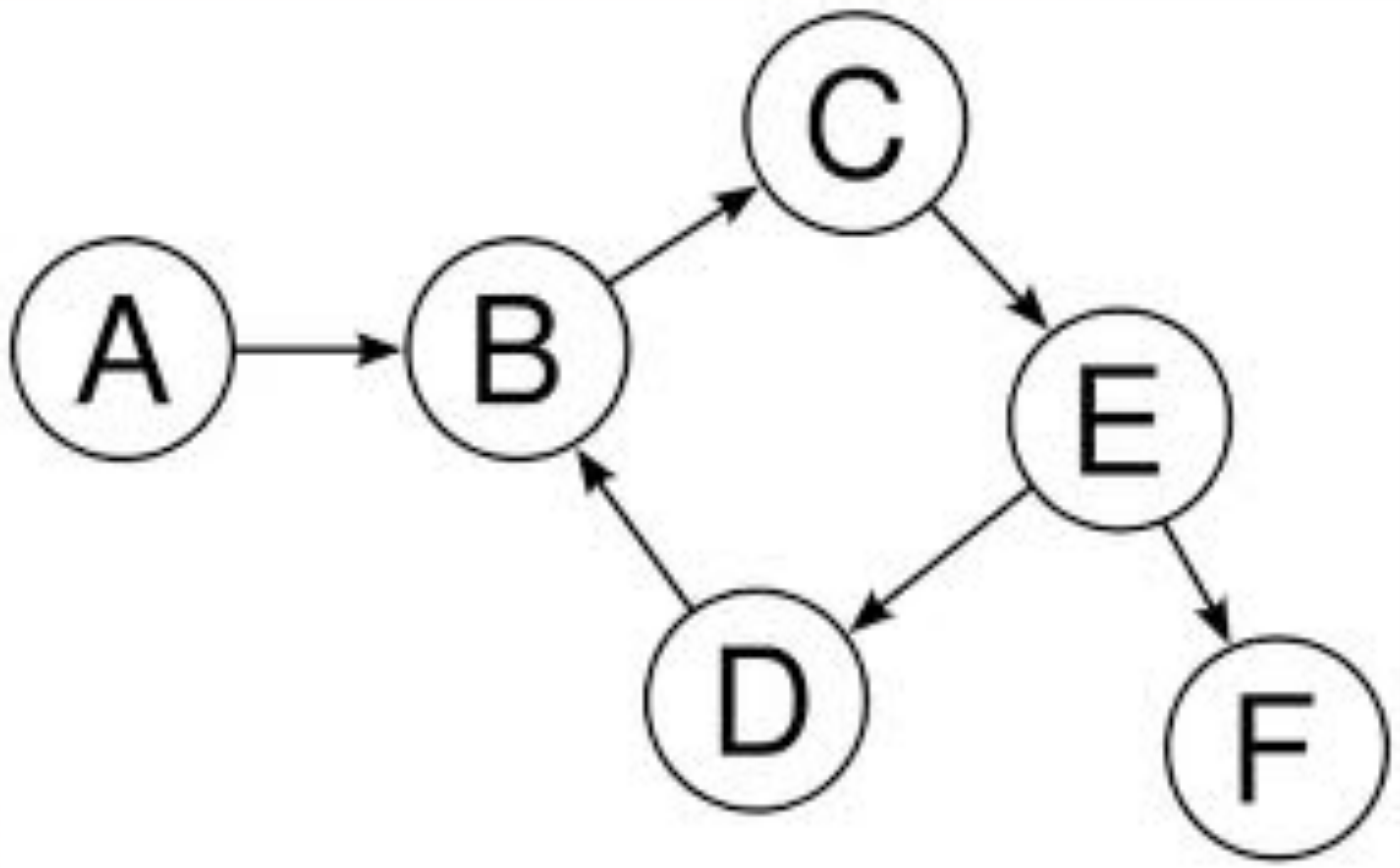
	Space	`getDegree(u)`	`isAdjacentTo(u,v)`	`getAdjacent(u)`
Adjacency List	$O(V+E)$	$O(\mathrm{D}(u))$	$O(\mathrm{D}(u))$	$O(\mathrm{D}(u))$
Adjacency Matrix	$O(V^2)$	$O(V)$	$O(1)$	$O(V)$

Paths

- A **path** (sometimes a **walk**) is a series of edges which starts at a vertex, travels from vertex to vertex along edges and ends at some vertex
  - Formally a path p is a set of edges s.t.
    $\boldsymbol{p}$**.start** $= e_0$.start
    $\boldsymbol{p}$**.end** $= e_n$.end
    e_n.end $= e_{n+1}$.start
- If p.start $= p$.end, then p is a **circuit** (**cycle**)
- If no edge is repeated, the path is **simple**

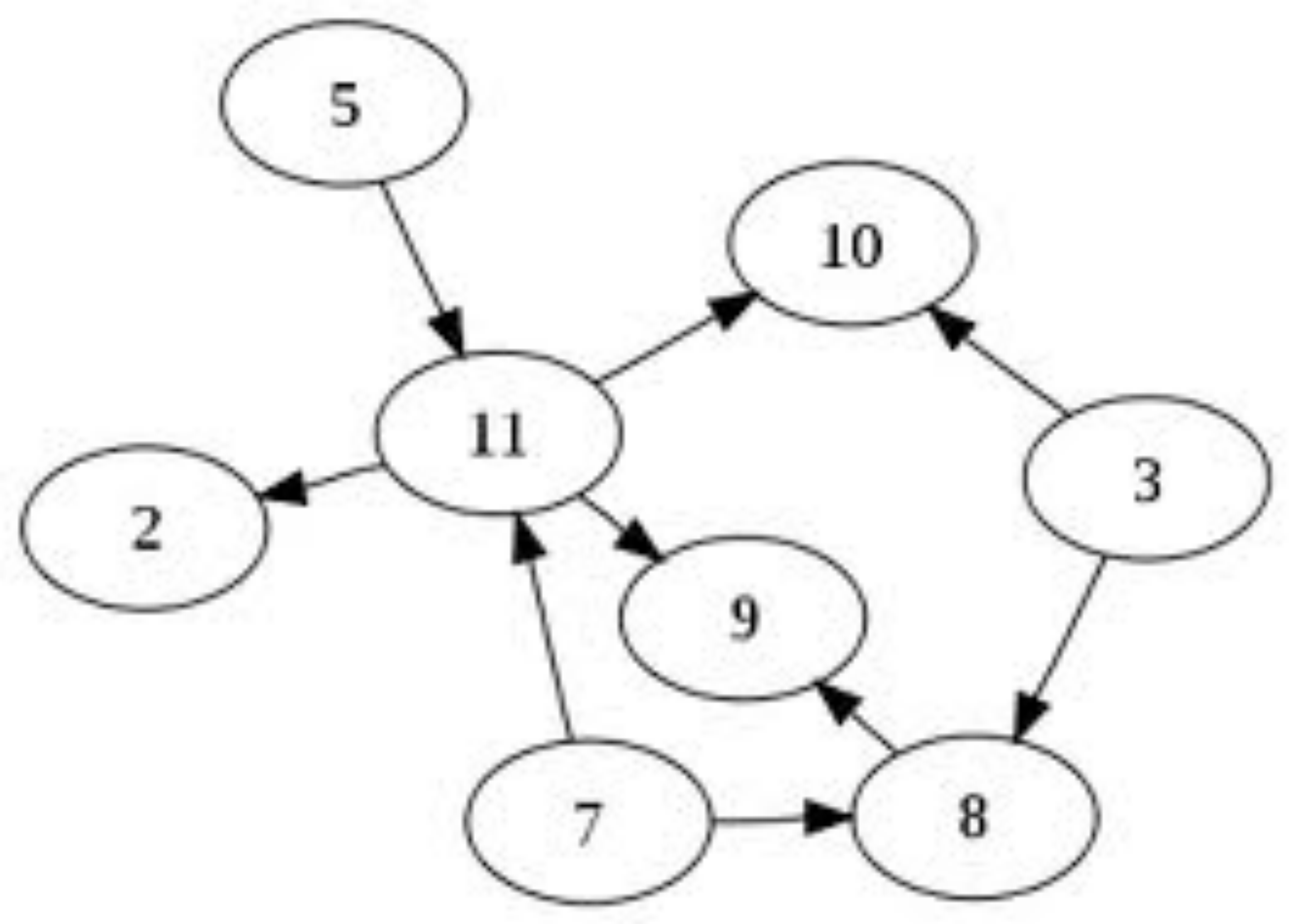


Cyclic vs Acyclic

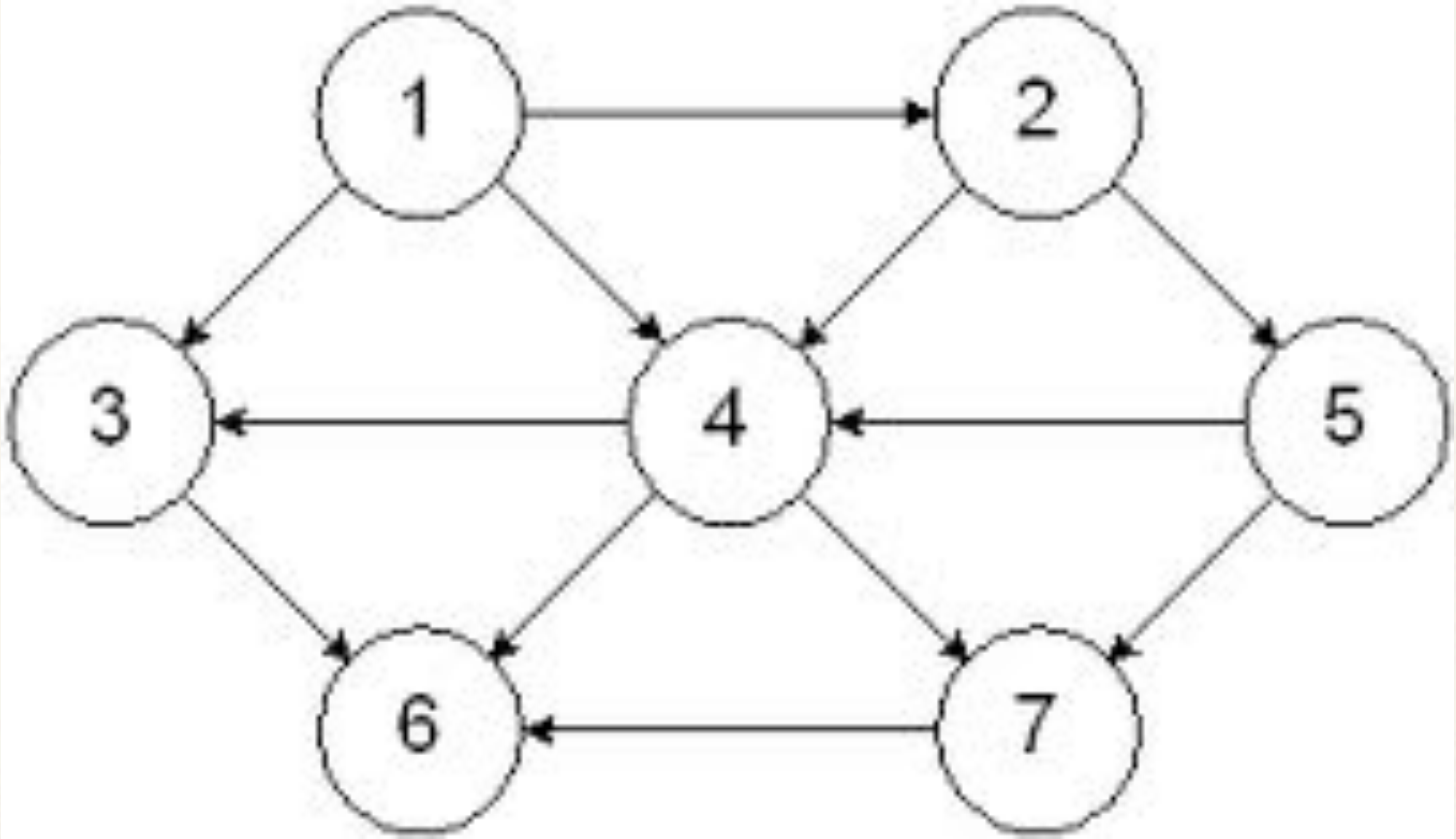


- Directed graphs can be **cyclic** or **acyclic**
  - Cyclic: contains a cycle
  - Acyclic: does not contain any cycles

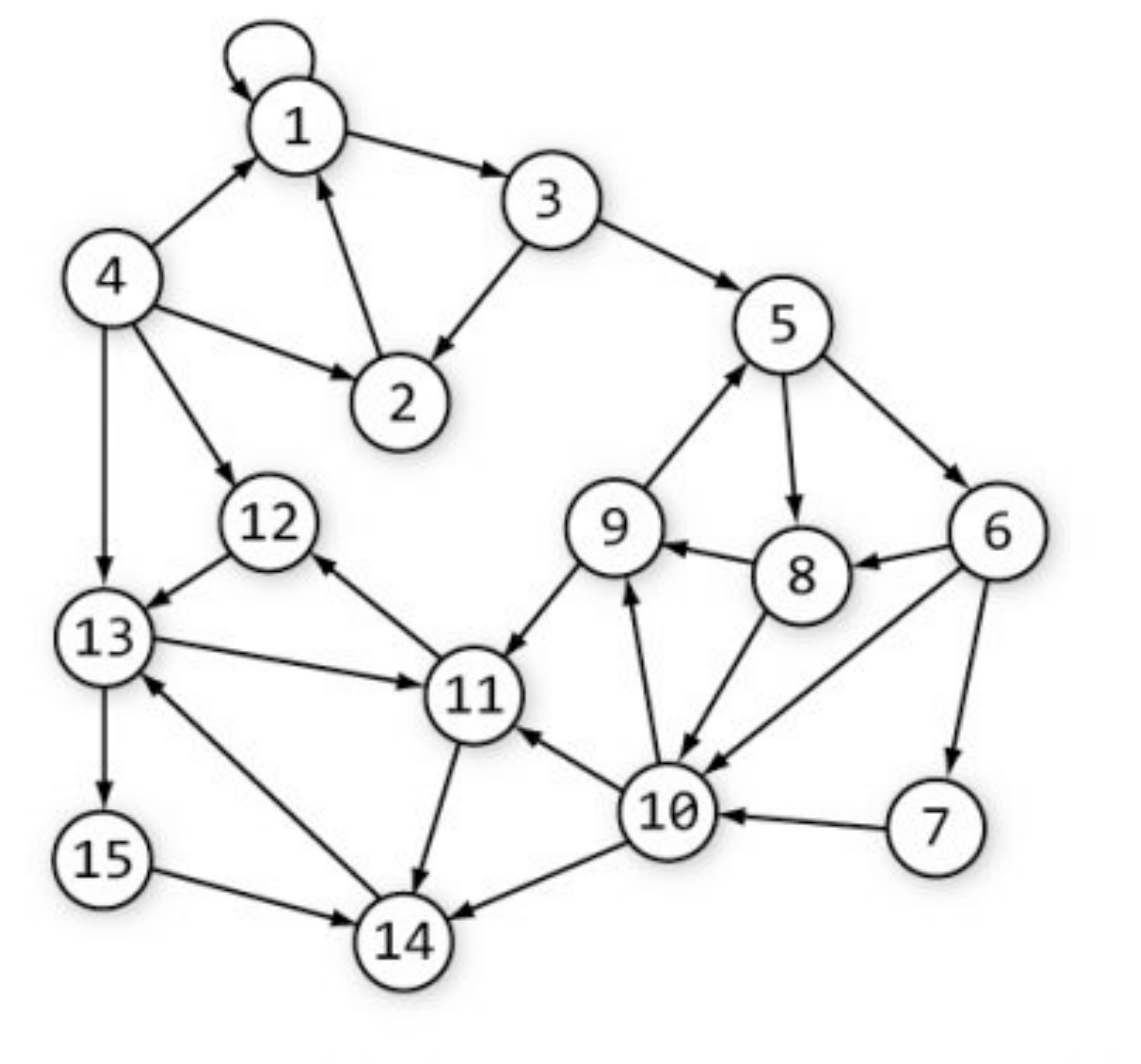
Cycle Practice



Cycle Practice



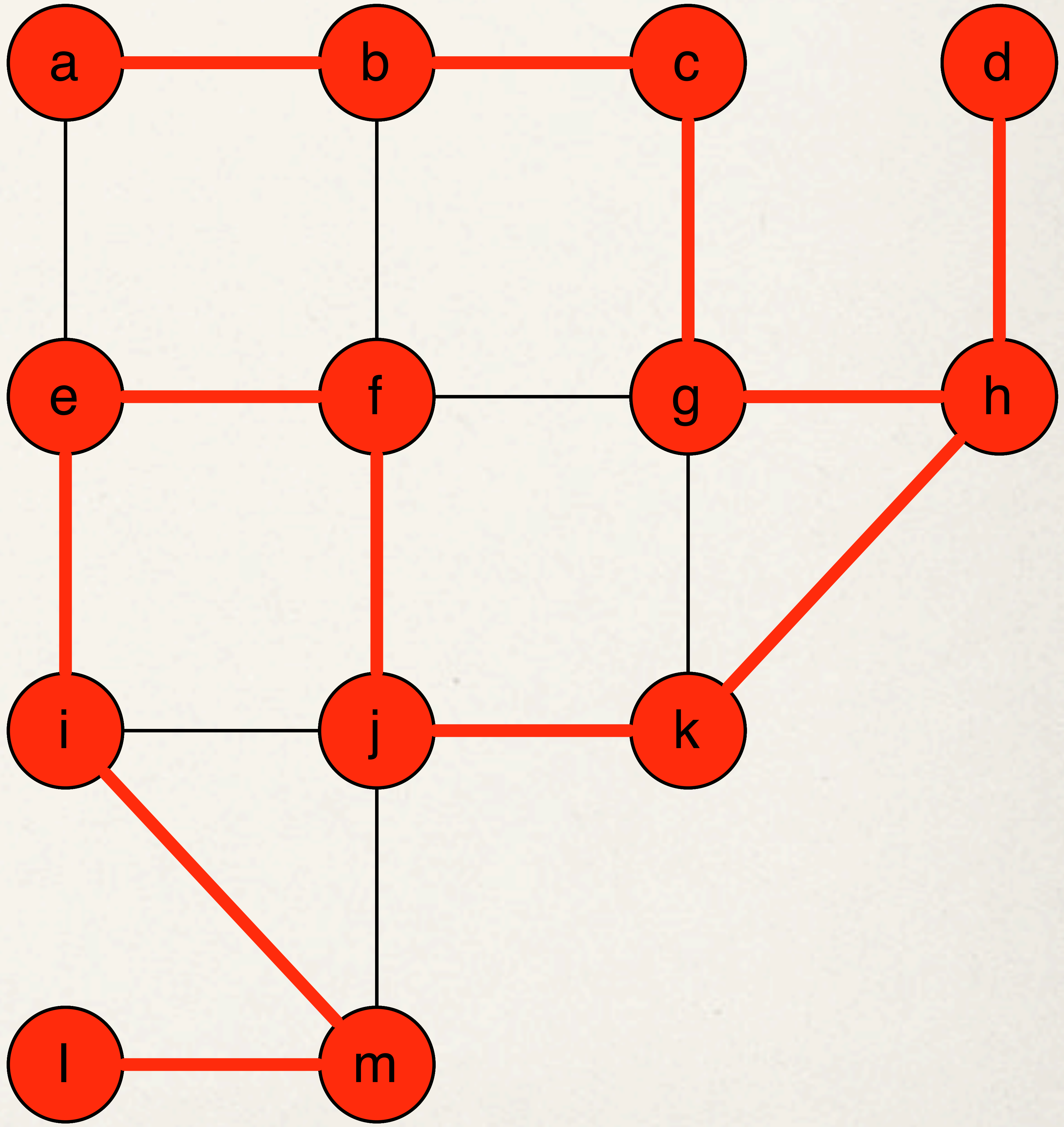
How many cycles are there?



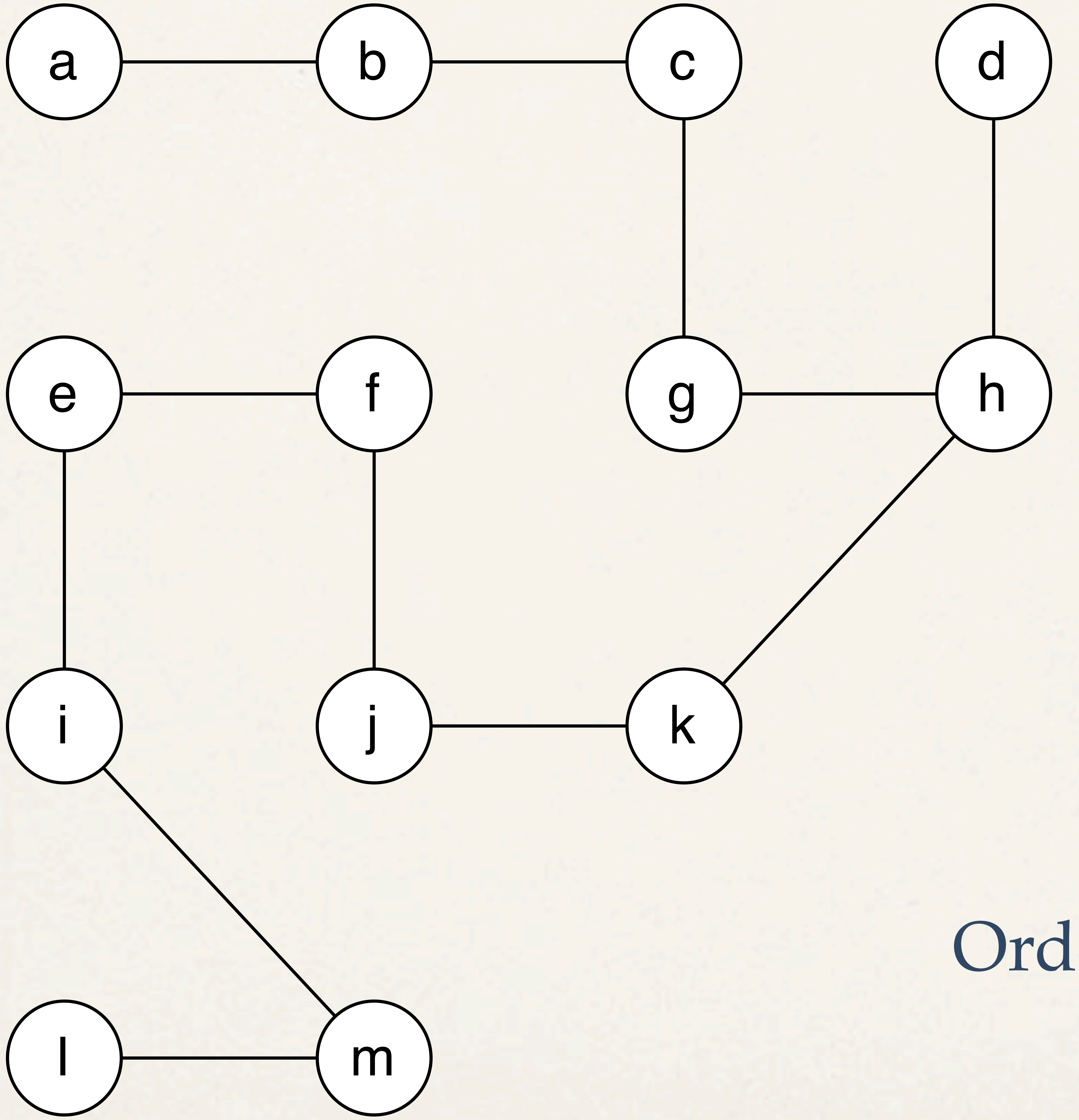
Graph Search

Depth-First Search (DFS)

- **Visit** some start vertex
- **Follow** an edge to a vertex which hasn't been explored
    - **Visit** that vertex
    - **Follow** an edge from that vertex to another unexplored vertex
    - If there are no edges to choose from, **backtrack** to the previous vertex



DFS

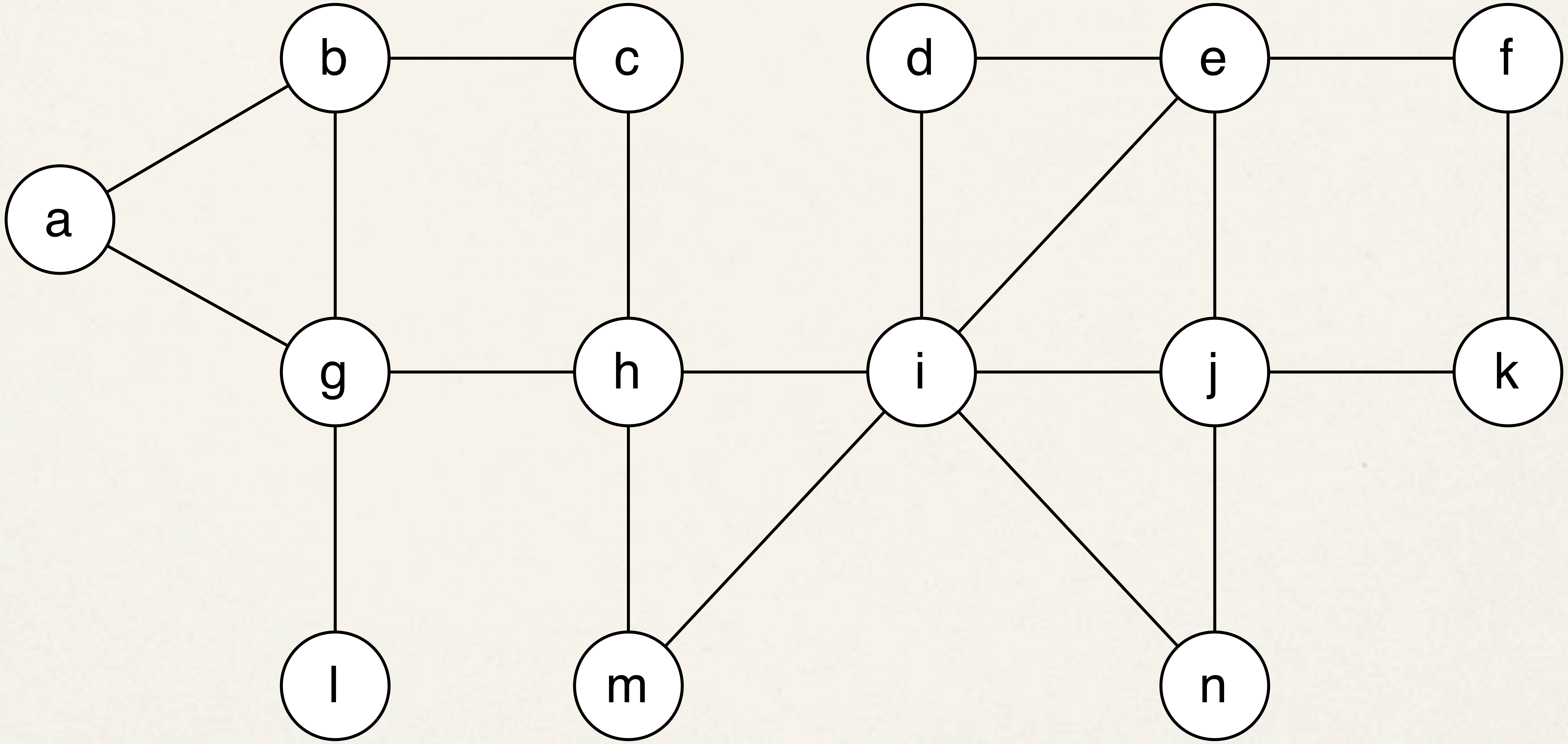


Order visited: a, b, c, g, h, d, k, j, f, e, i, m, l

A Recursive DFS Algorithm

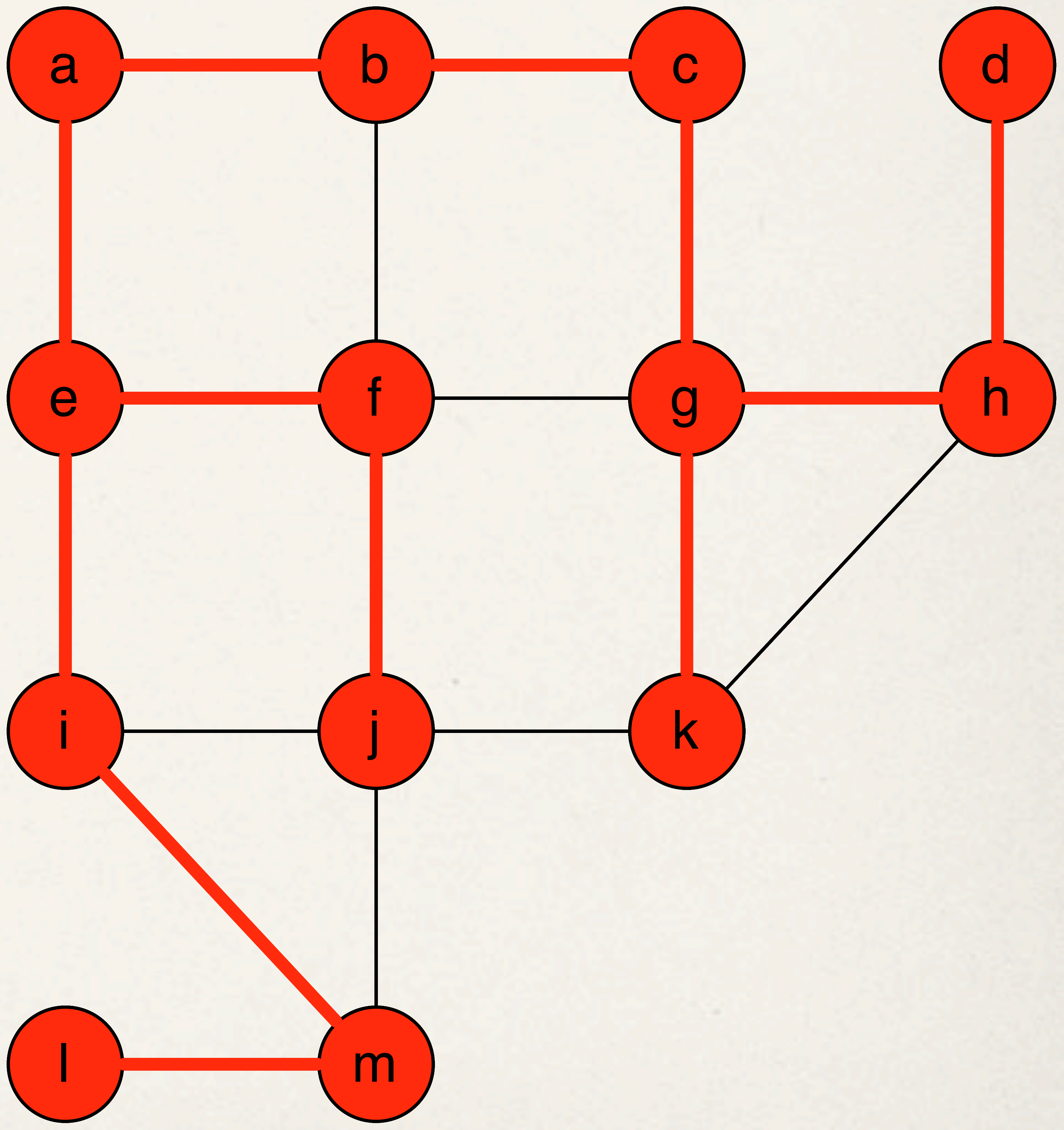
```
procedure DFS(G: connected graph with vertices $v_1, v_2, ..., v_n$)
T := tree consisting only of the vertex $v_1$f
visit($v_1$)

procedure visit(v: vertex of G)
for each vertex w adjacent to v and not yet in T
    add vertex w and edge {v, w} to T
    visit(w)
```

Practice

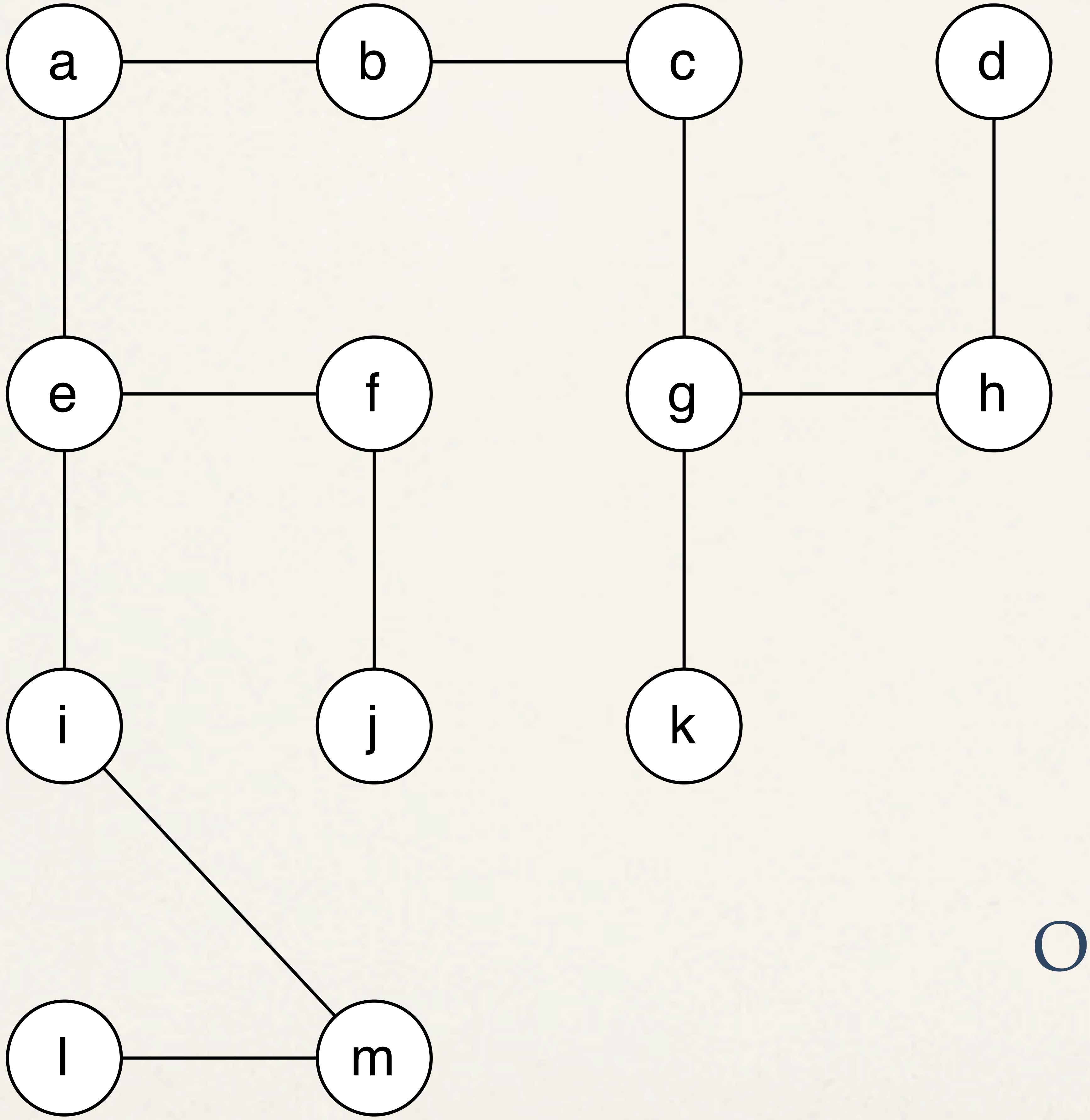


Breadth-First Search

- **Visit** some start vertex
- **Visit** all neighbors of the start vertex
- **Visit** all those neighbors' neighbors
- Repeat until all vertices visited



BFS



Order visited: a, b, e, c, f, i, g, j, m, h, k, l, d

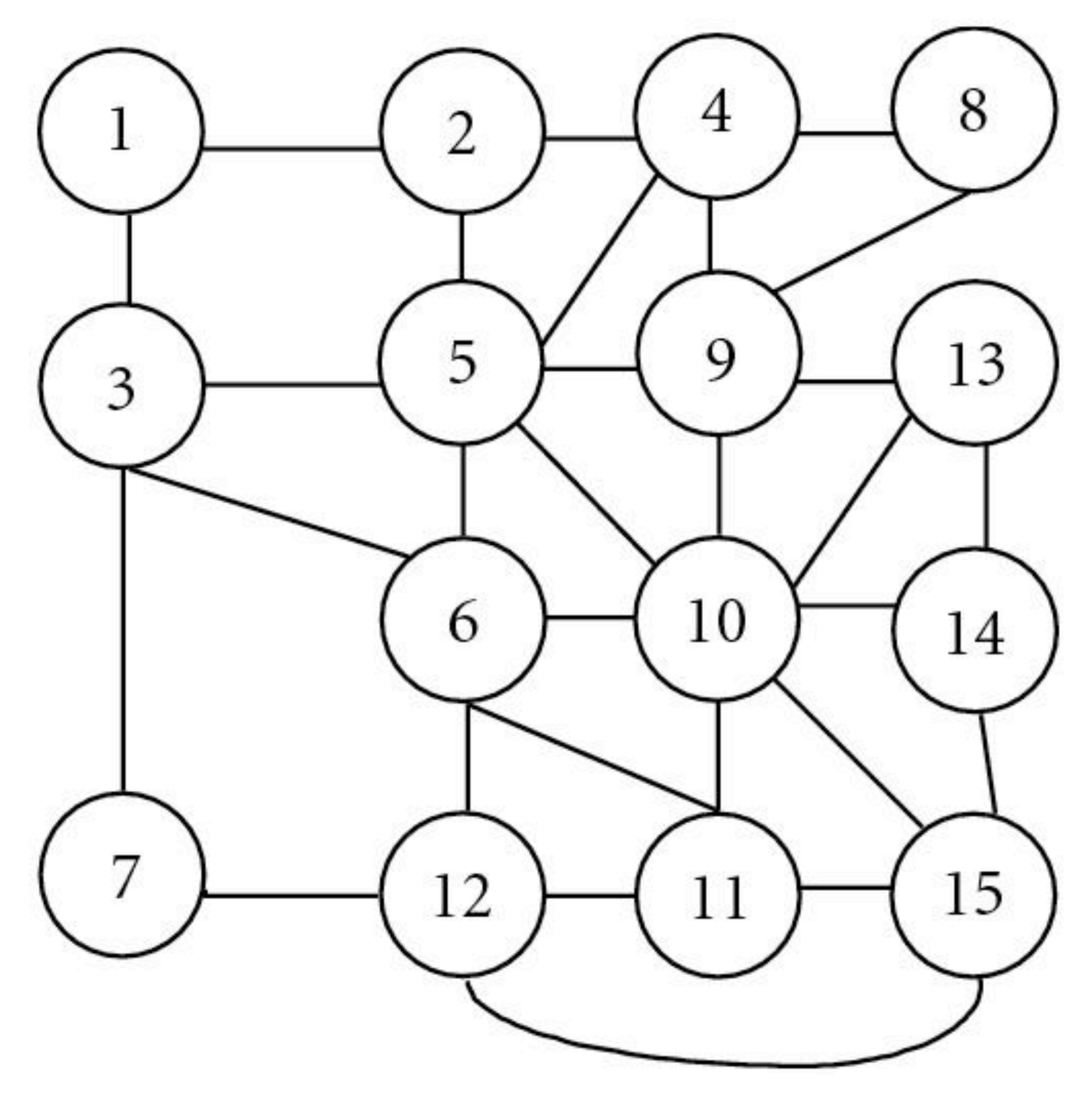
A (non-recursive) BFS Algorithm

```
**procedure** *BFS* (*G*: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of vertex $v_1$
L := empty list
put $v_1$ in the list L of unprocessed vertices
while L is not empty
    remove the first vertex, v, from L
    for each neighbor w of v
        if w is not in L and not in T then
            add w to the end of the list L
            add w and edge {v, w} to T
```

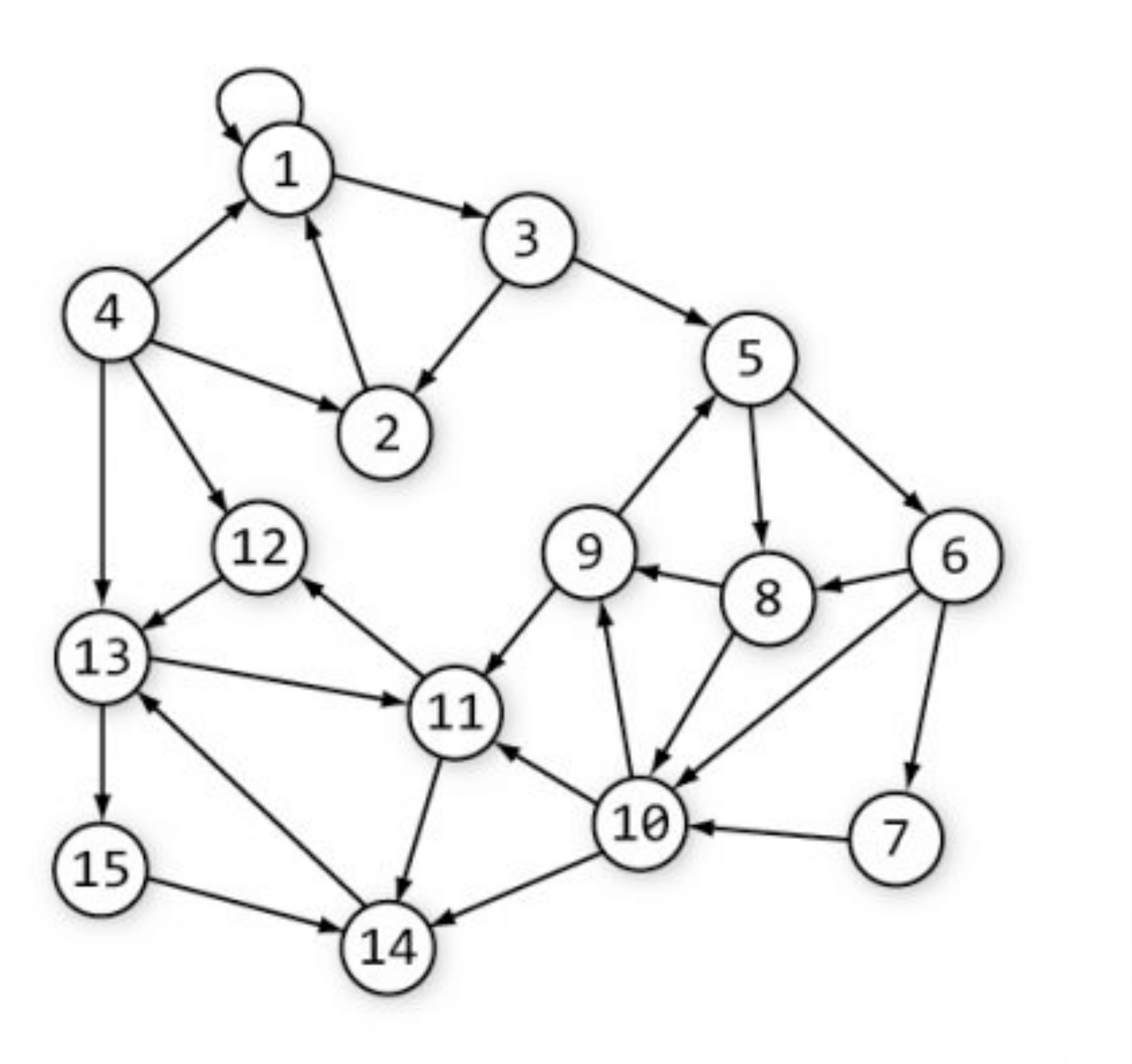
Shortest-Path Problems

Shortest Path

- The **length** of a path is the number of edges in it
- Many problems try to find the **shortest path** between two vertices
  - Flights with the fewest stopovers
  - Driving directions with few instructions
- *Can* use DFS or BFS

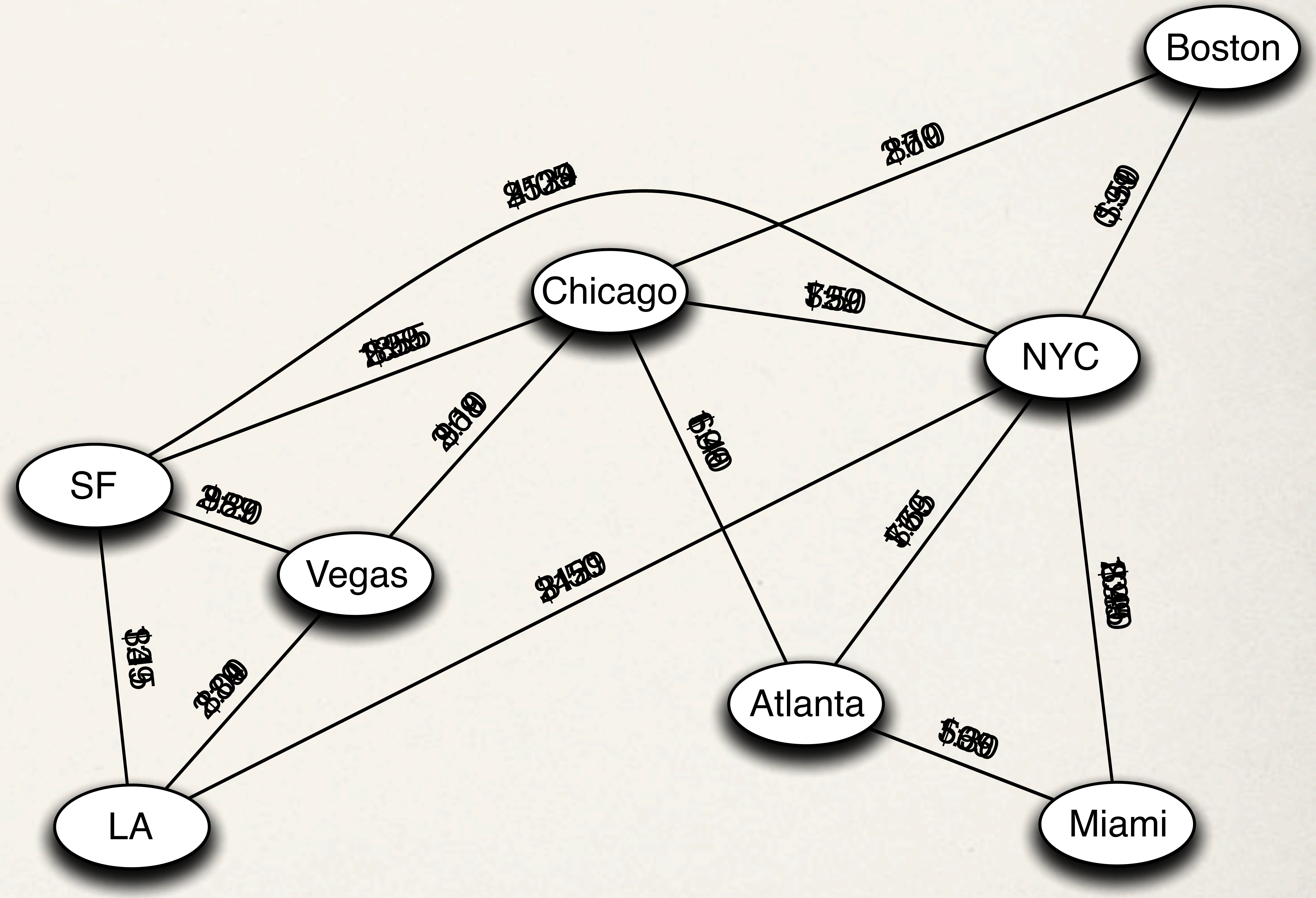


Practice



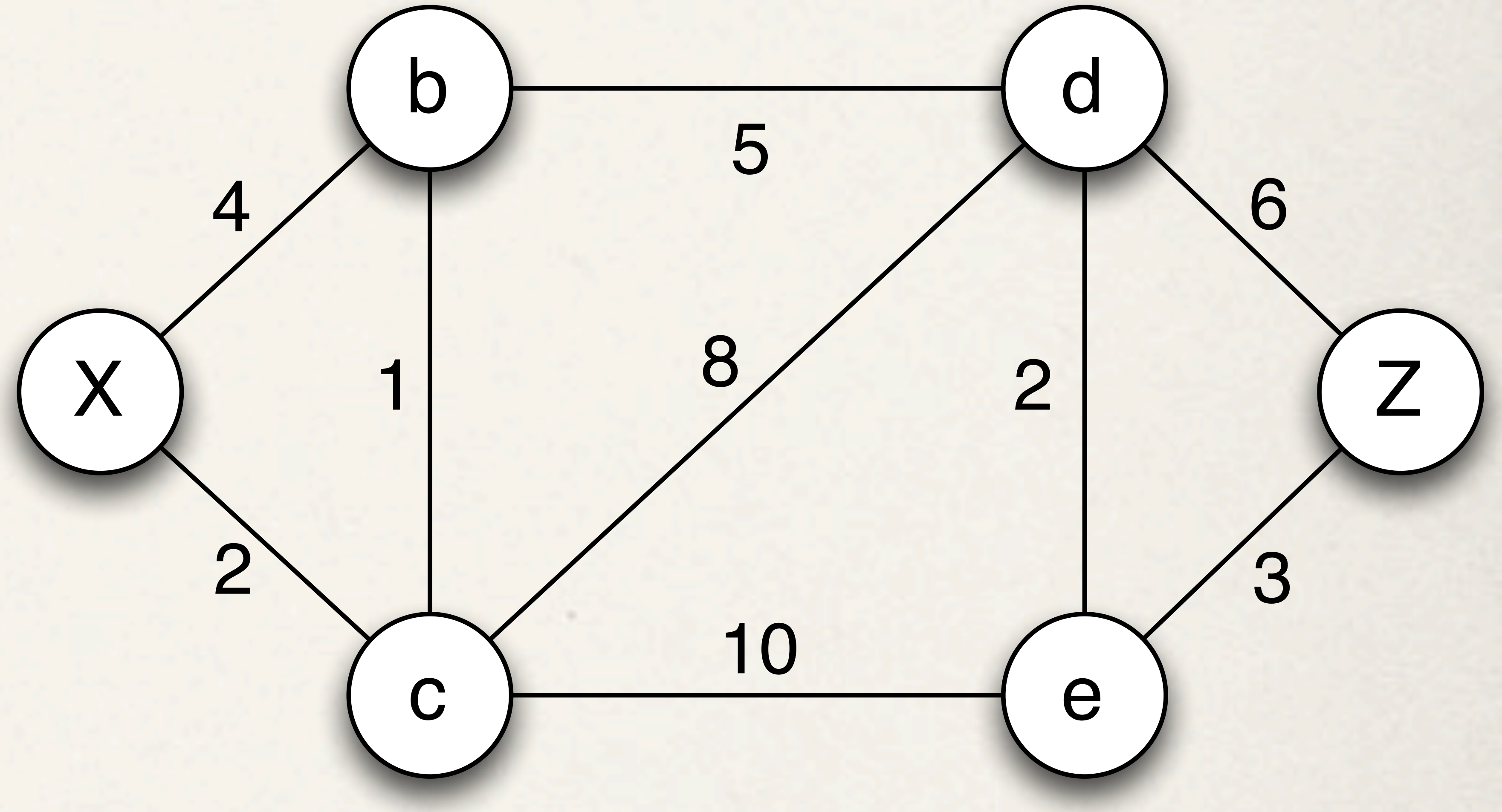
Weighted Graphs

- Add a **weight** to every edge
  - weights are normally “costs”
- Length of a path is the sum of the weights of each edge

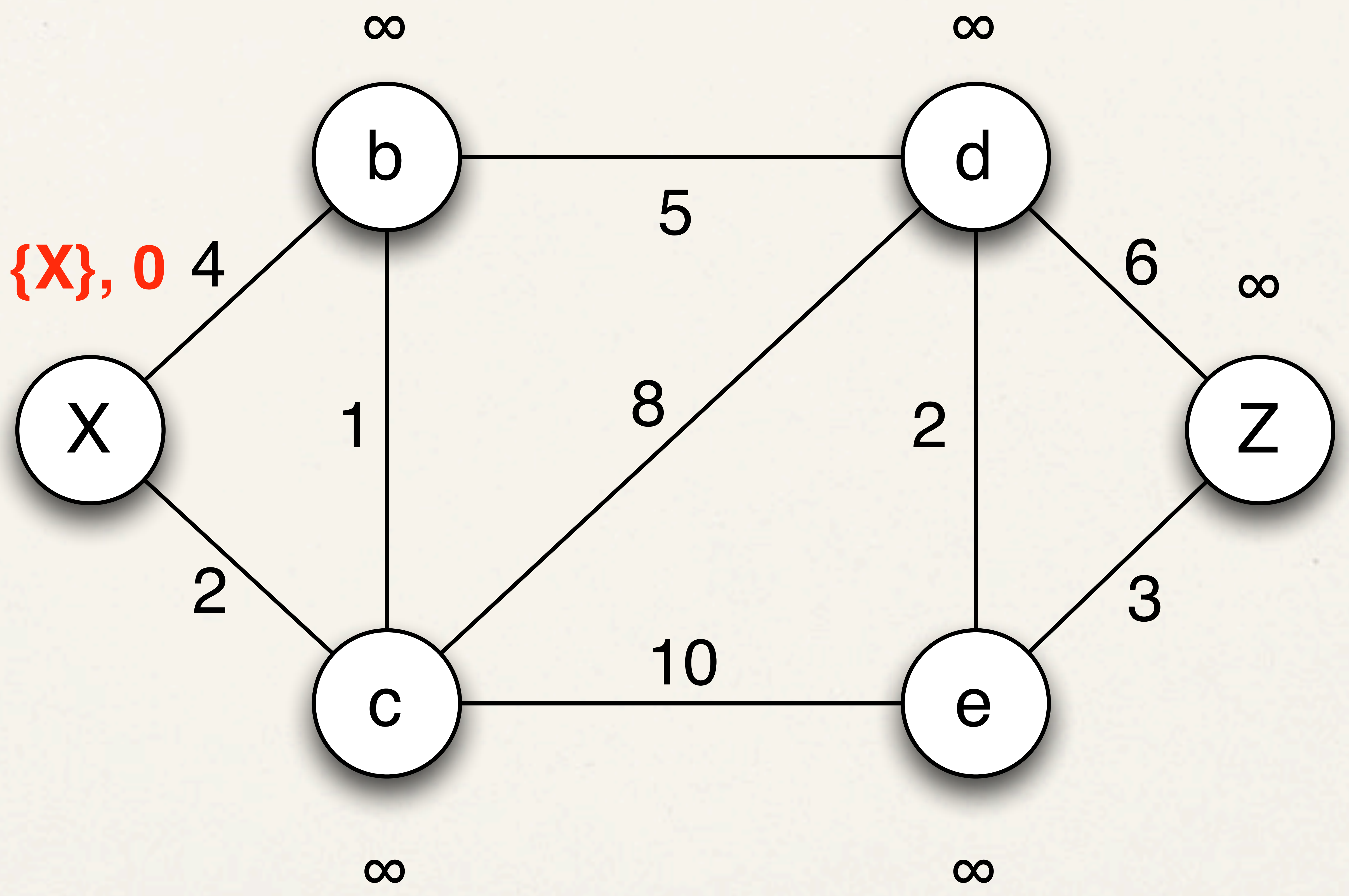


Dijkstra's Algorithm

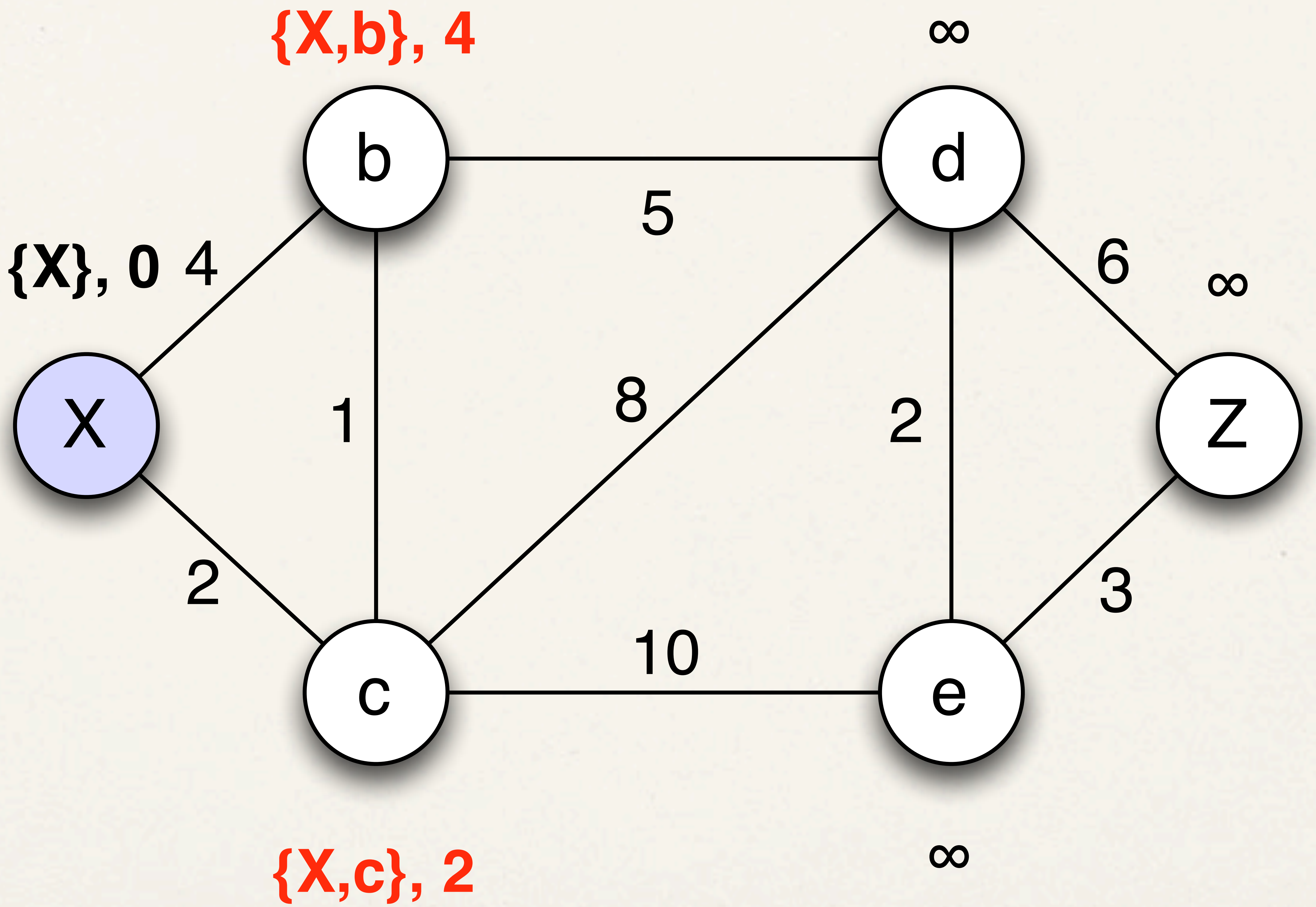
- Finds shortest path between X and Z in $O(n^2)$ time
- **Greedy algorithm:**
  - Start with $C = \{X\}$
  - Search the neighbors of C to find next closest node to X not in C
    - Add it to C
    - Repeat until Z is the next closest node to X



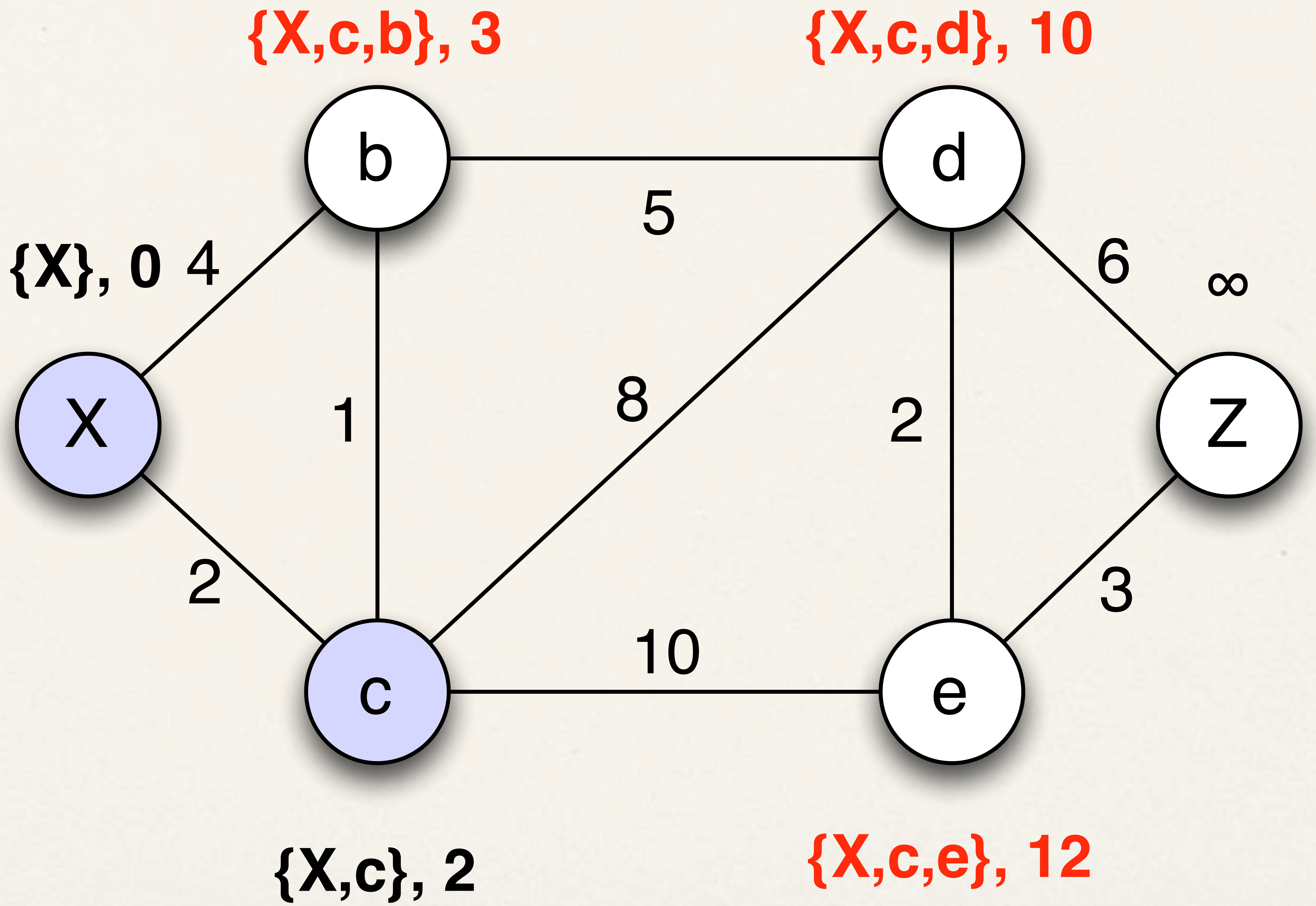
Dijkstra’s Algorithm



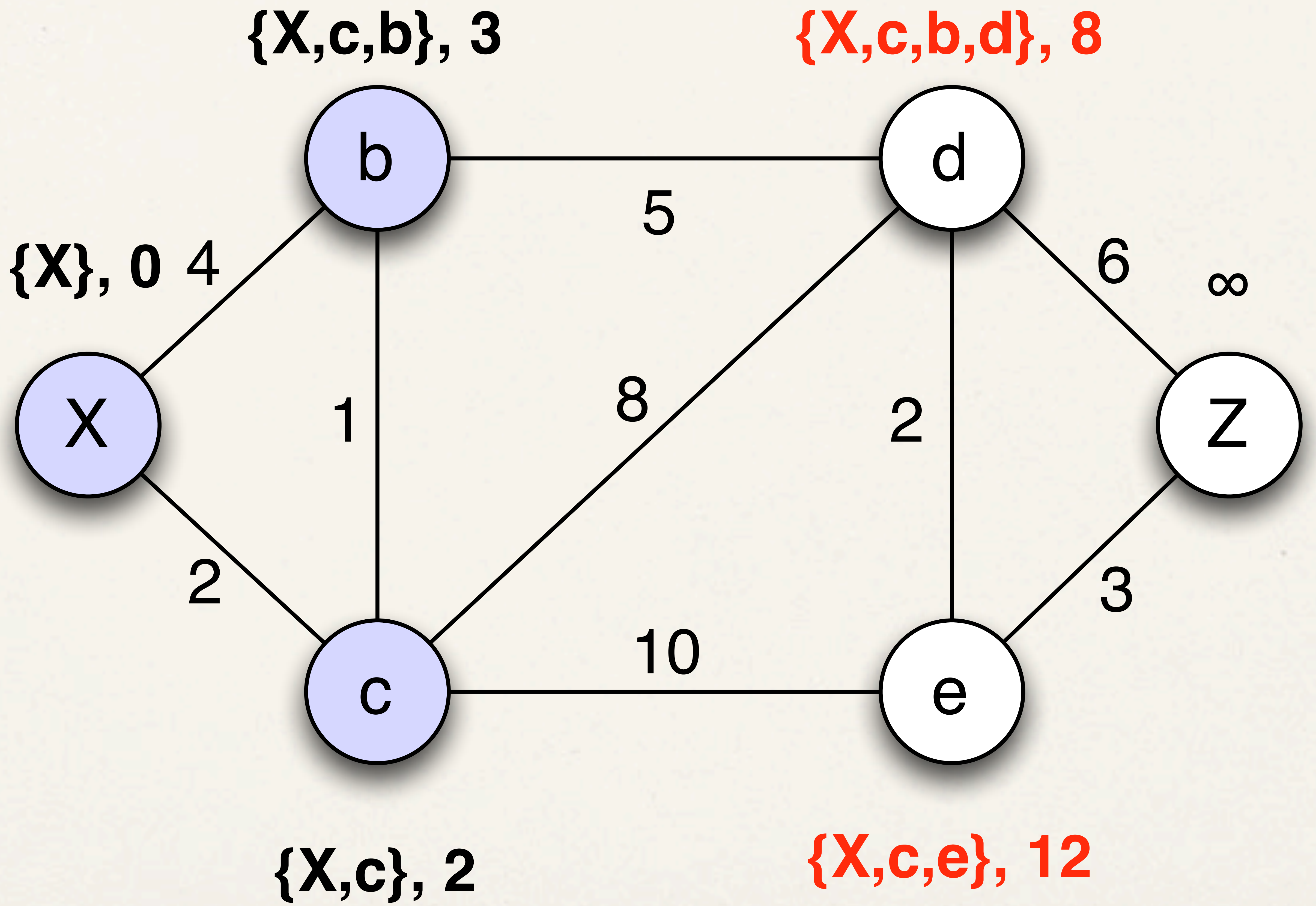
Dijkstra's Algorithm



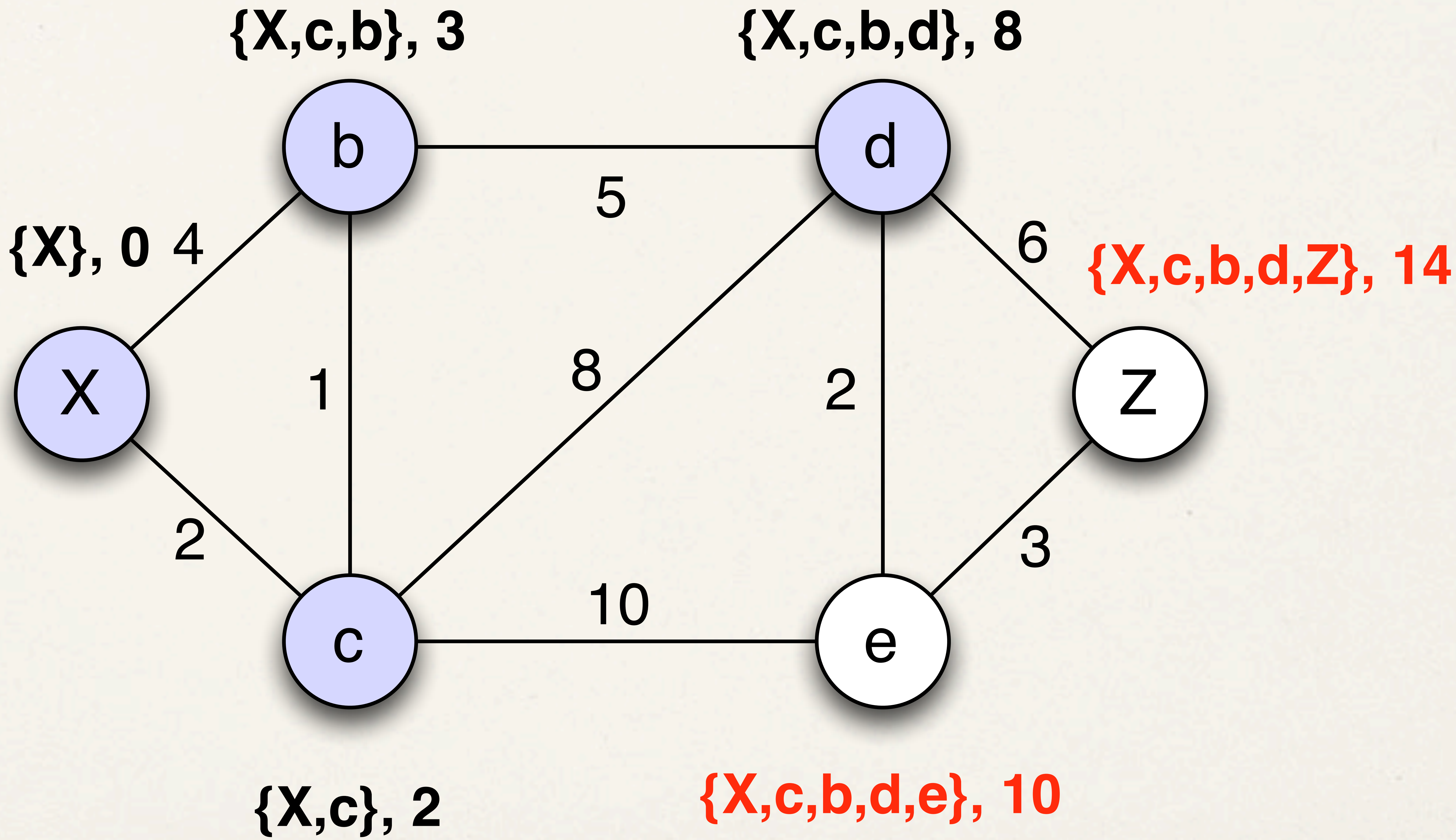
Dijkstra's Algorithm



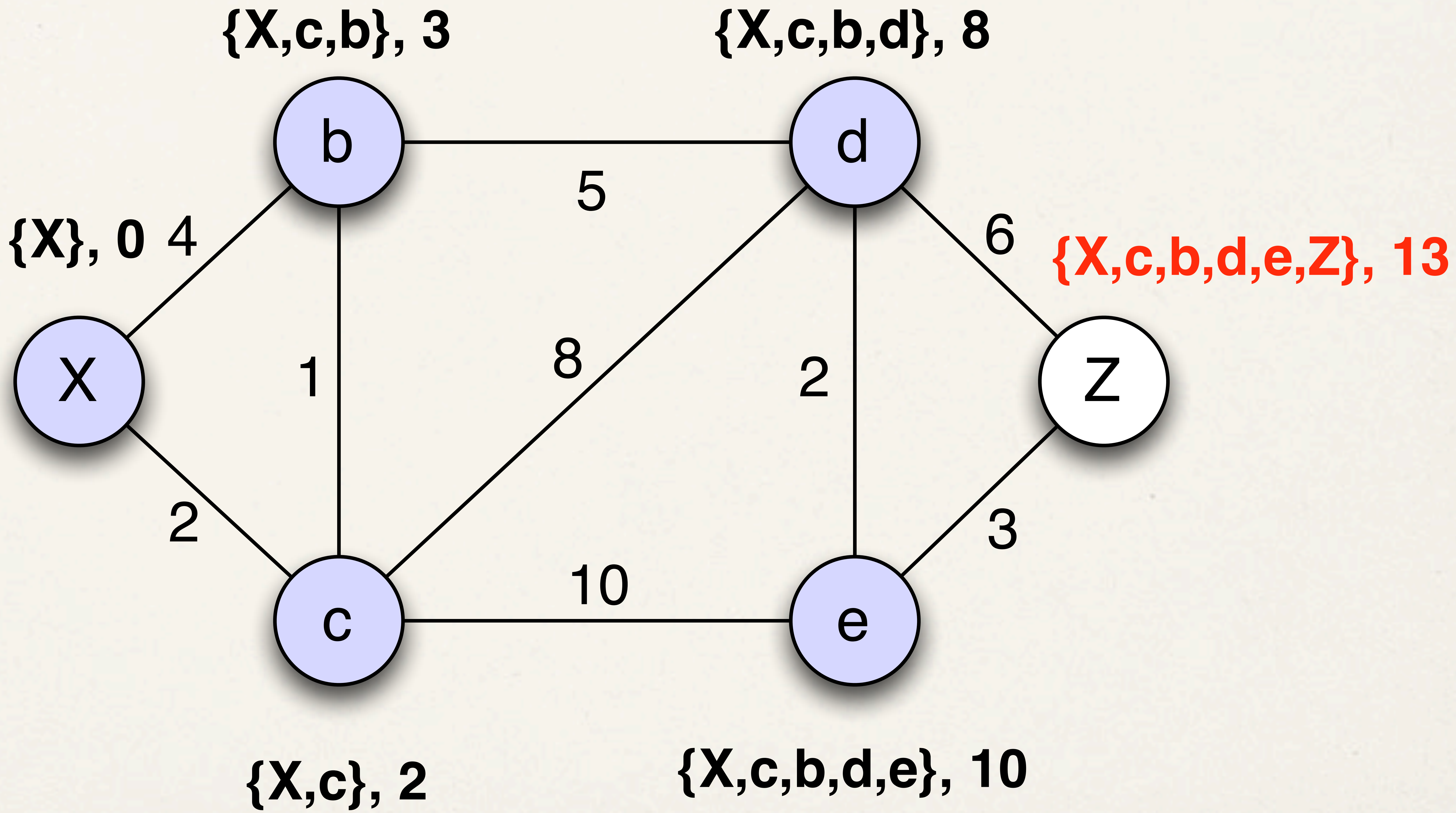
Dijkstra's Algorithm



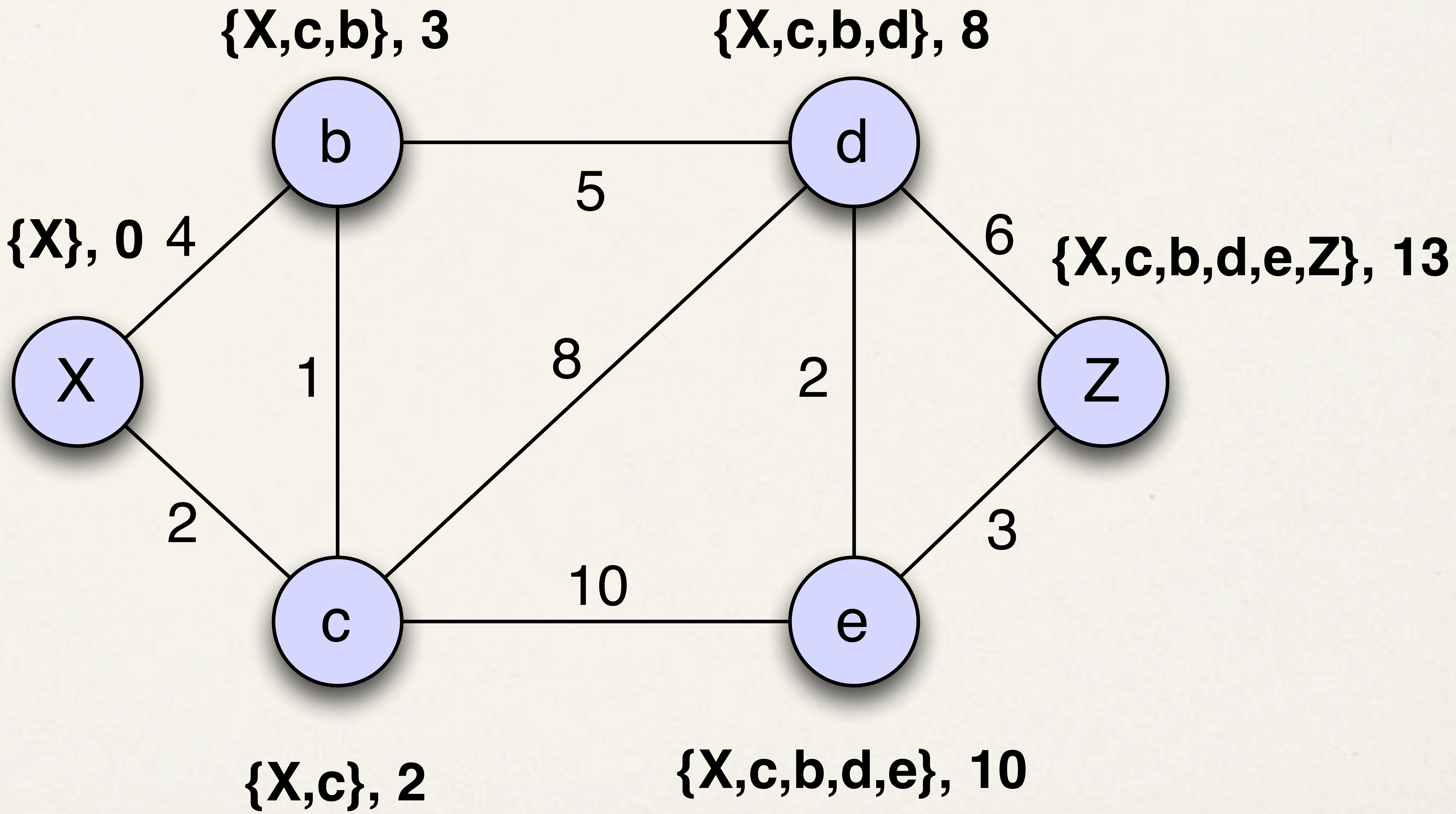
Dijkstra's Algorithm



Dijkstra's Algorithm

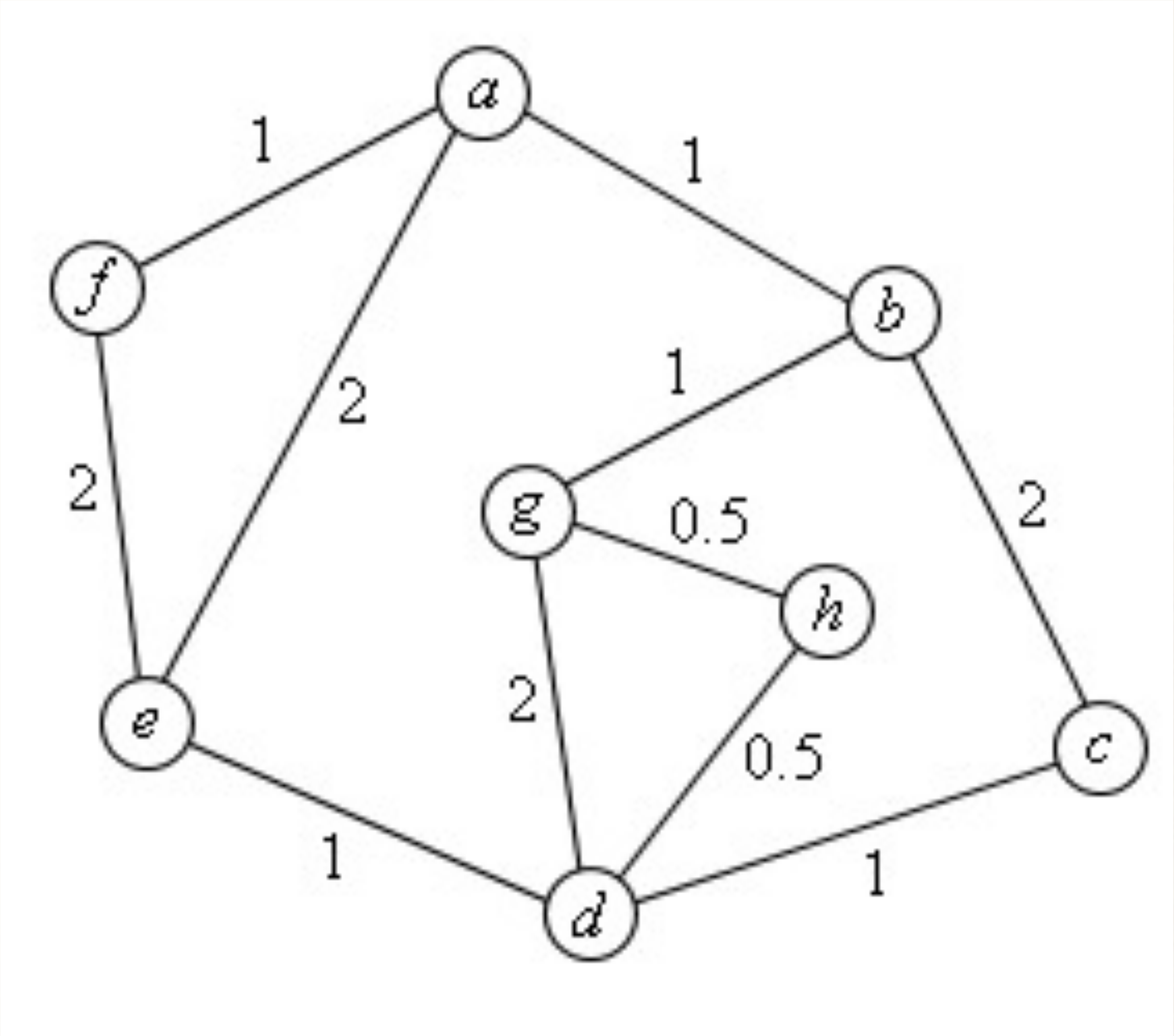


Dijkstra's Algorithm



Practice

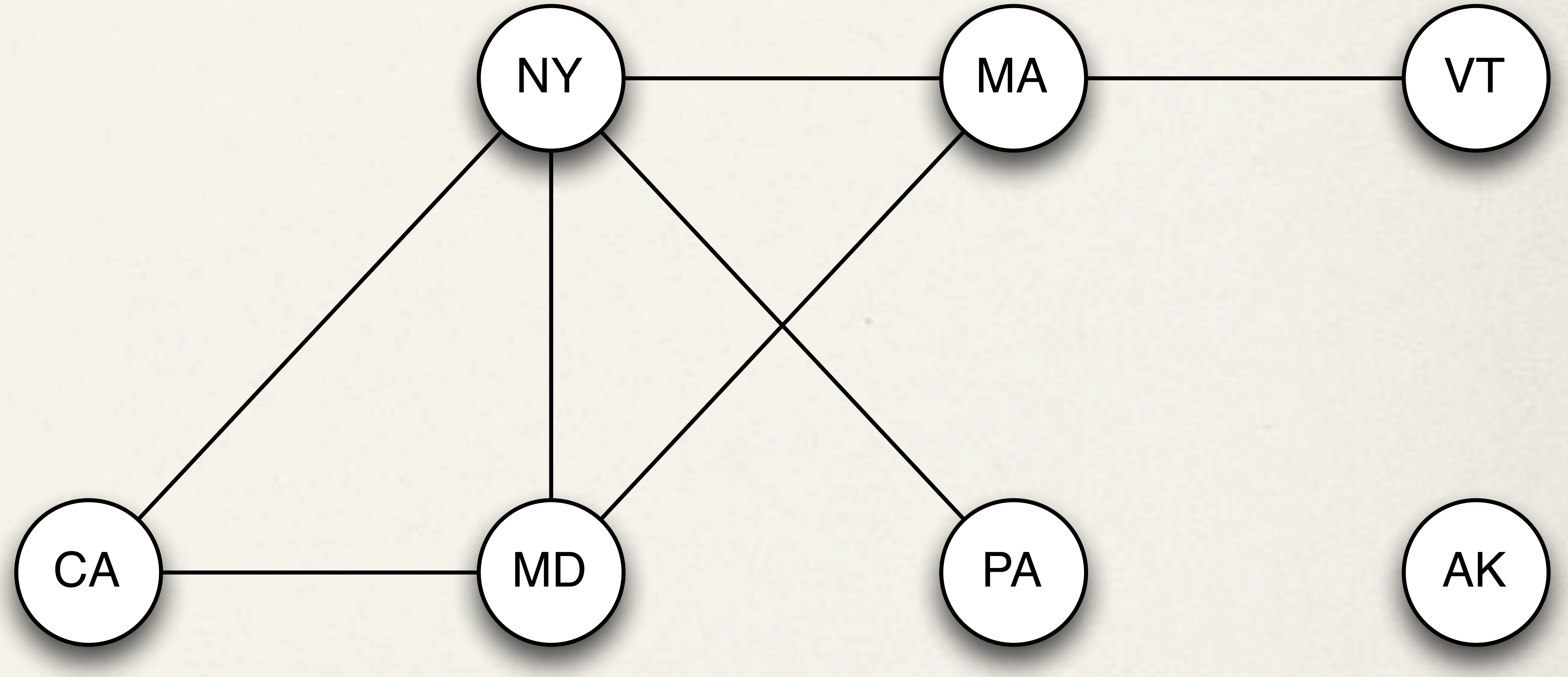
- Find the shortest path from *g* to *f* using Djikstra's Algorithm



Connectivity

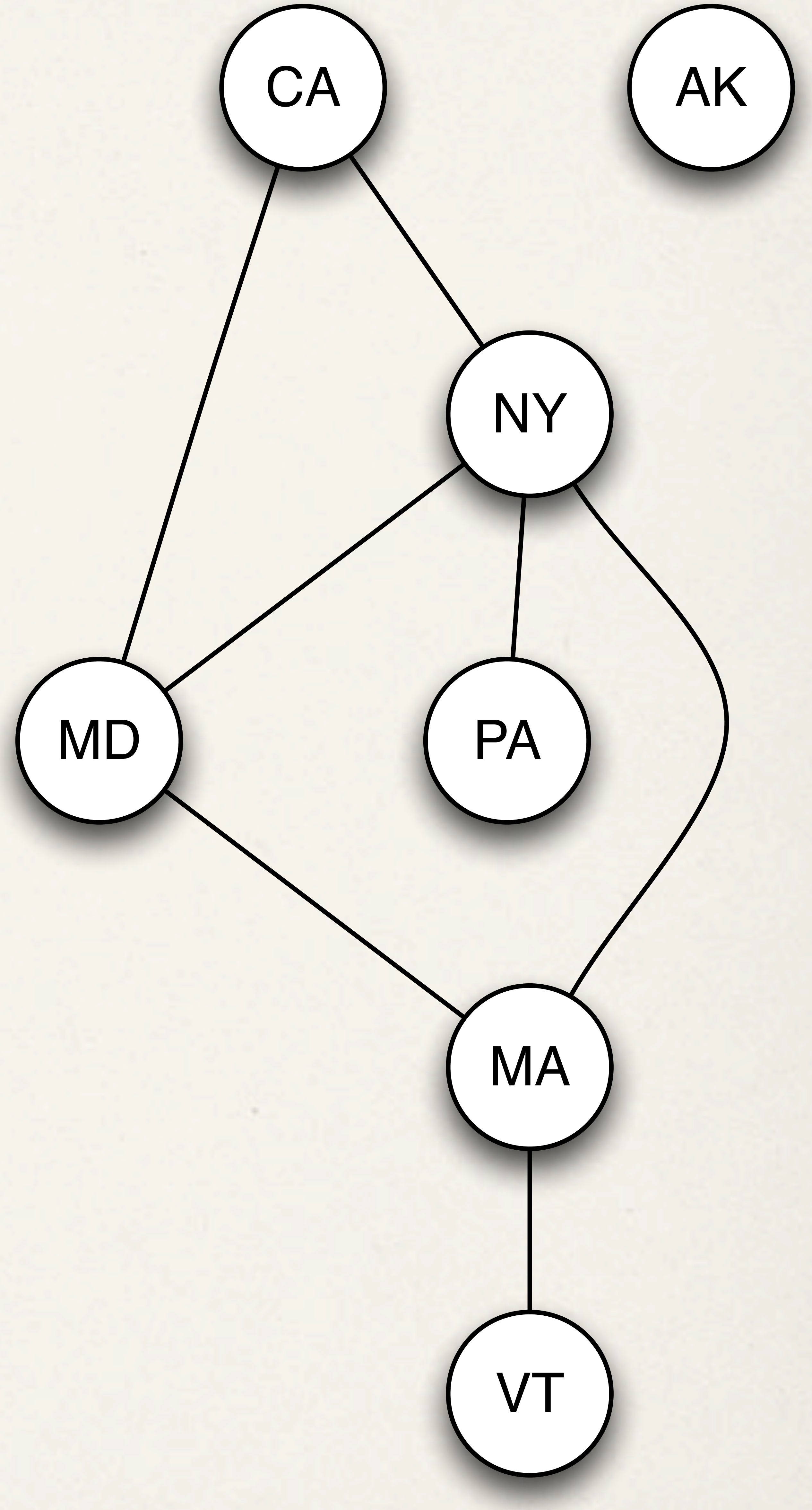
Network Reliability

- What would happen if my router in NY went offline? If CA got knocked out?
- I often want there to always be a path available between all the nodes in my graph

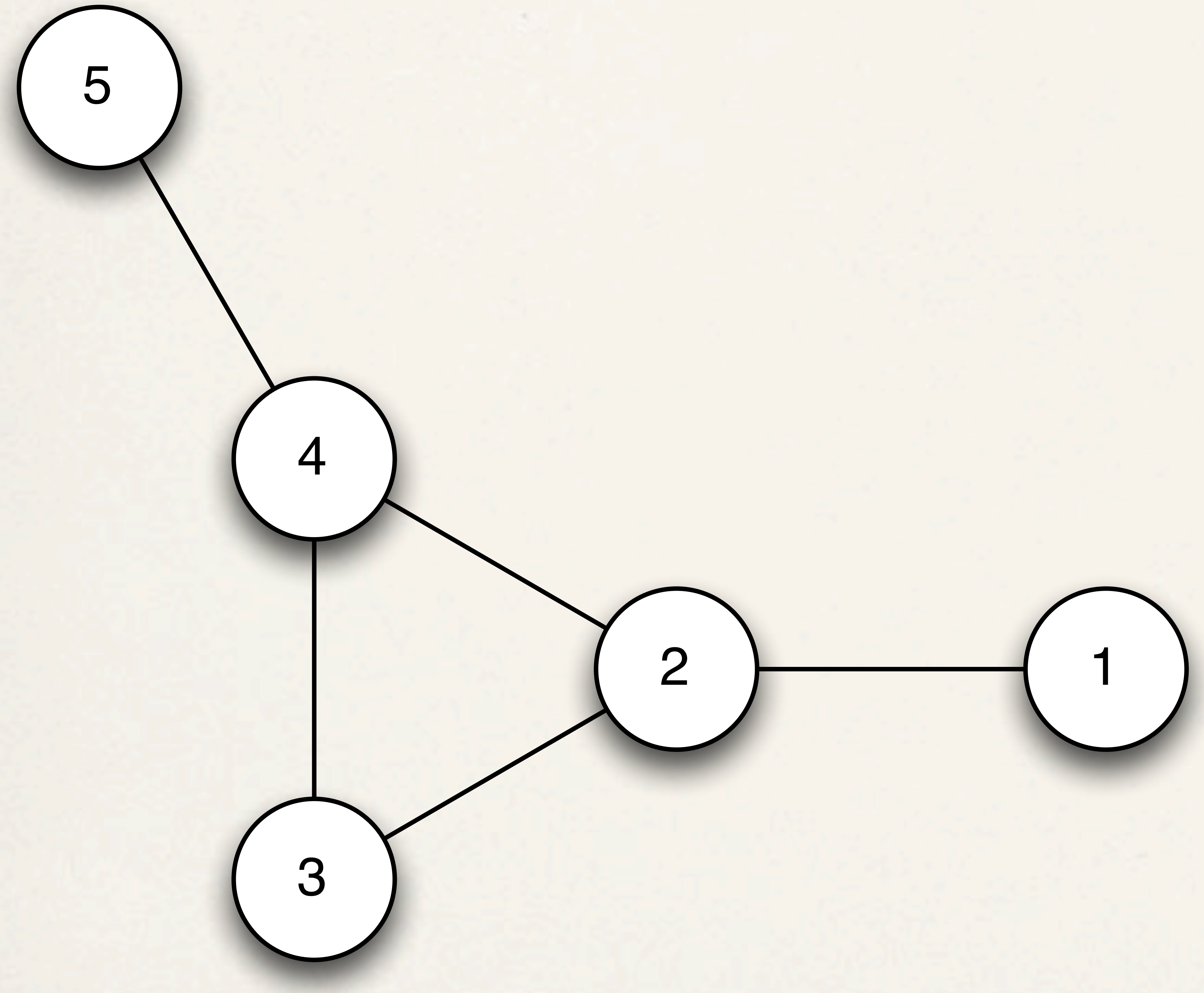


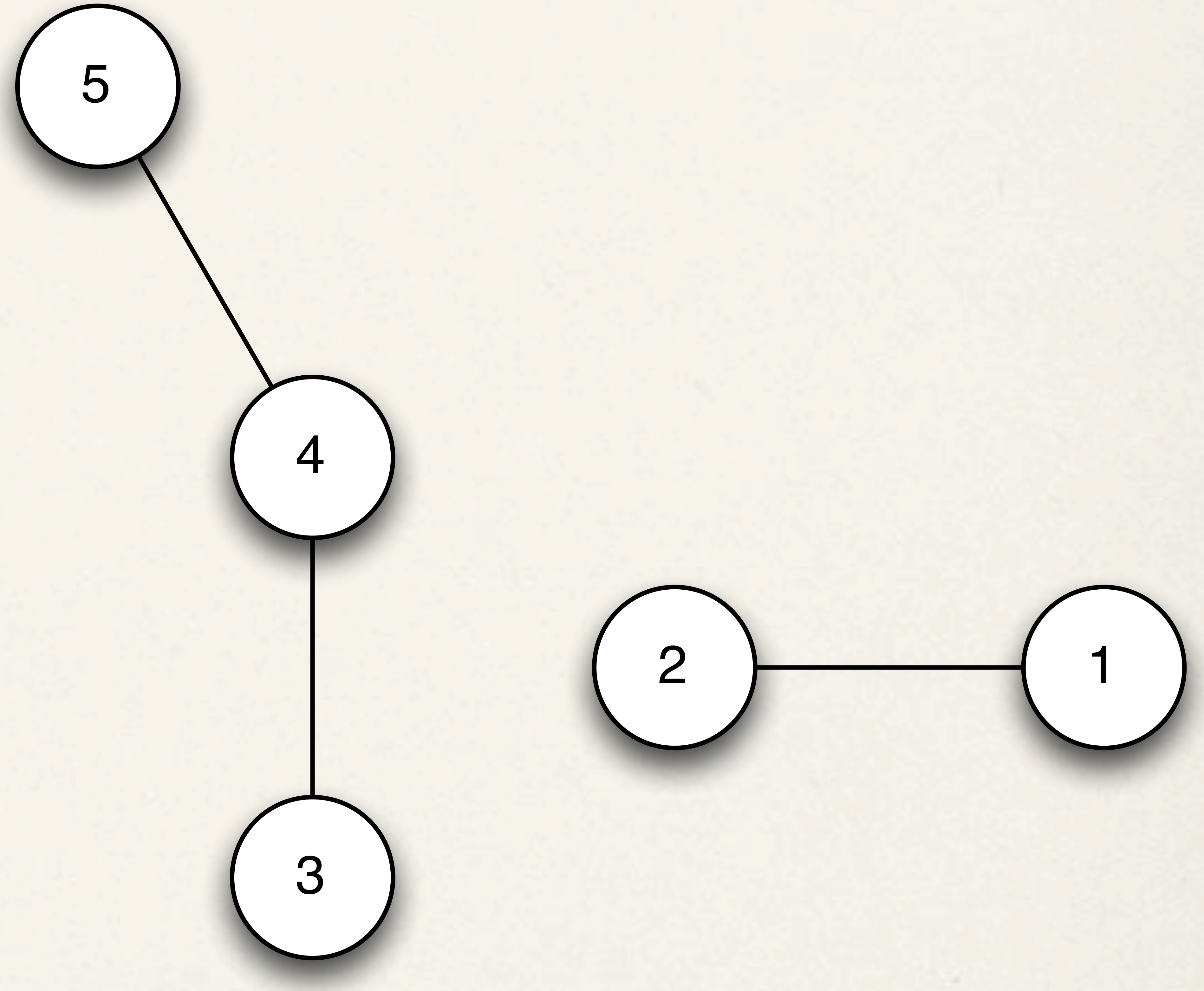
Connectedness

- Two nodes are **connected** if there exists a path between them
    - Otherwise they are **disconnected**
- If every pair of nodes in a graph is connected, then the whole graph is **connected**
    - Otherwise it is **disconnected**
- If a graph is connected, then there exists a simple path between every pair of vertices in the graph



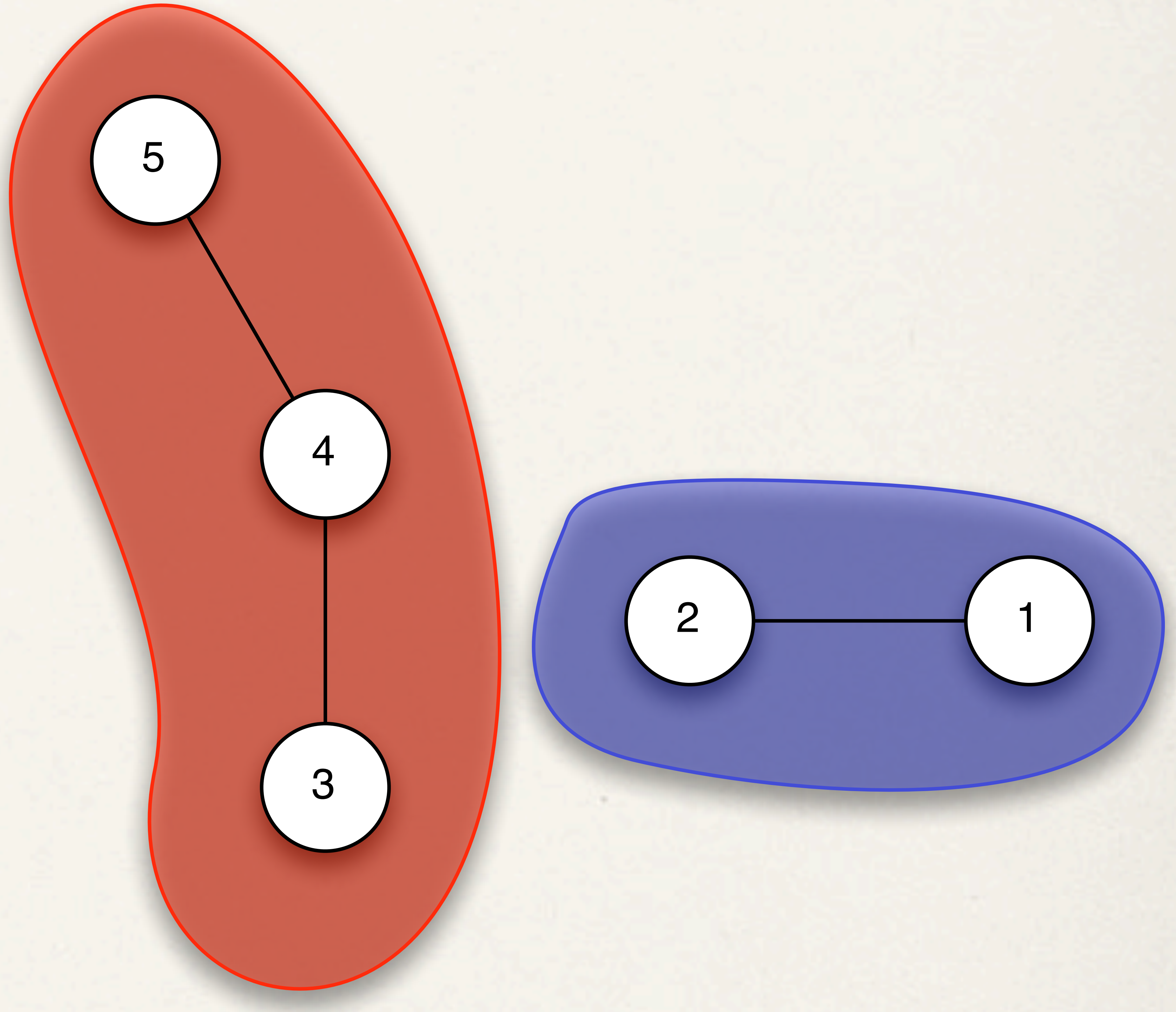
Connected vs Disconnected



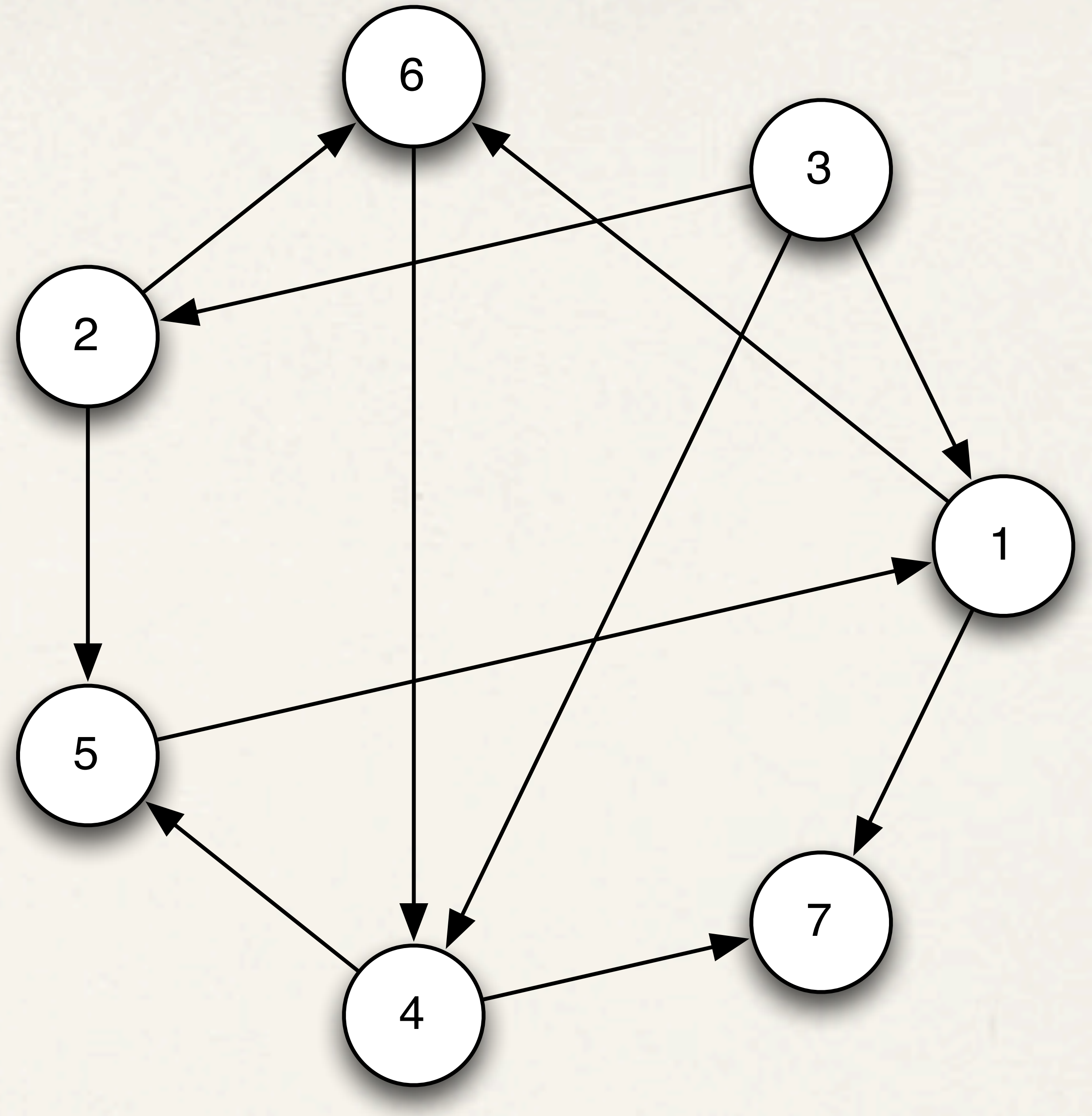


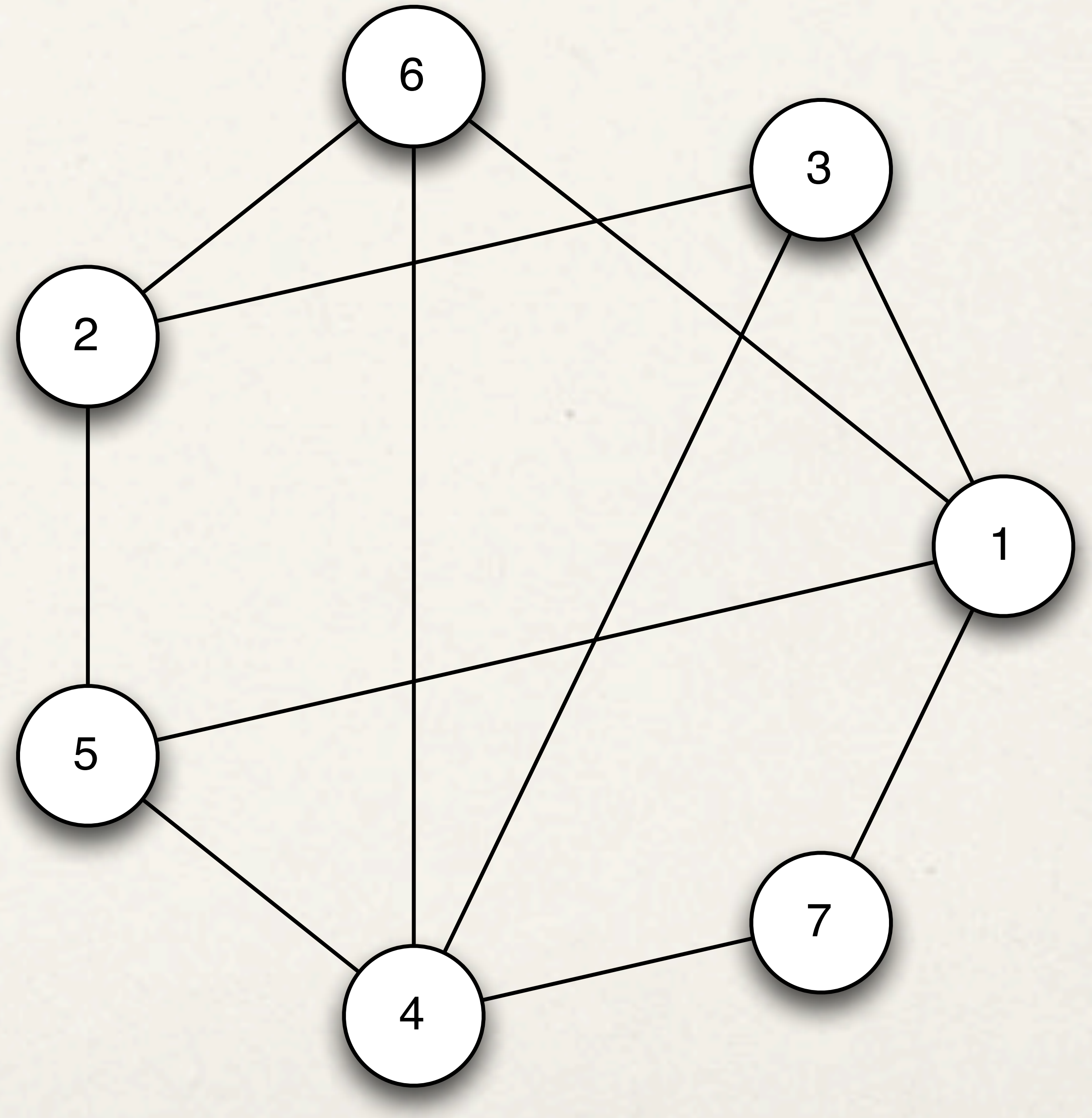
Connected Components

- A part of a graph which is disconnected from all other parts is called a **connected component**
- Formally, a connected subgraph of *G* which is not a proper subgraph of any other connected subgraph of *G* is a **connected component**
- A connected graph has 1 connected component
- A disconnected graph has 2+



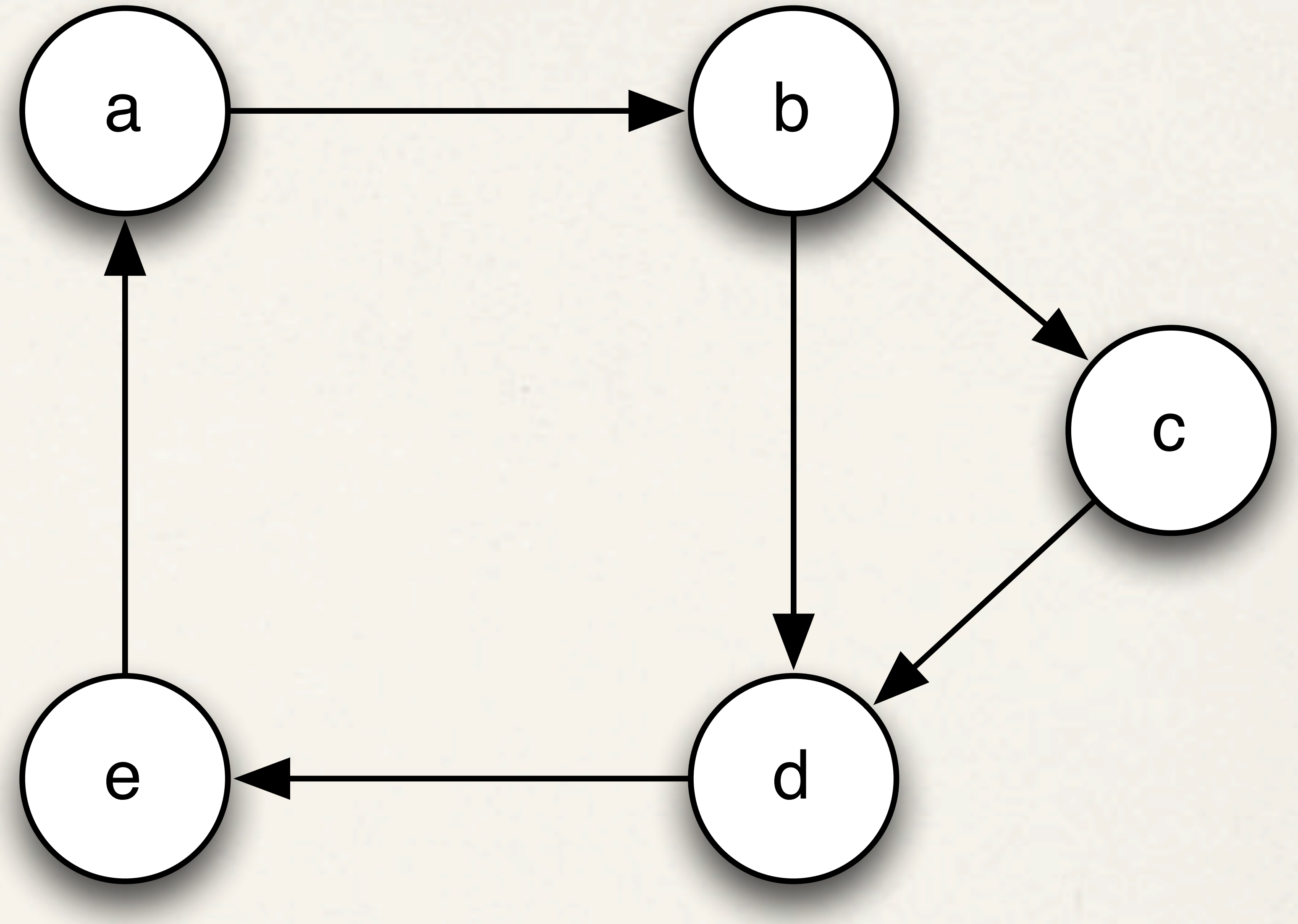
Directed Graphs



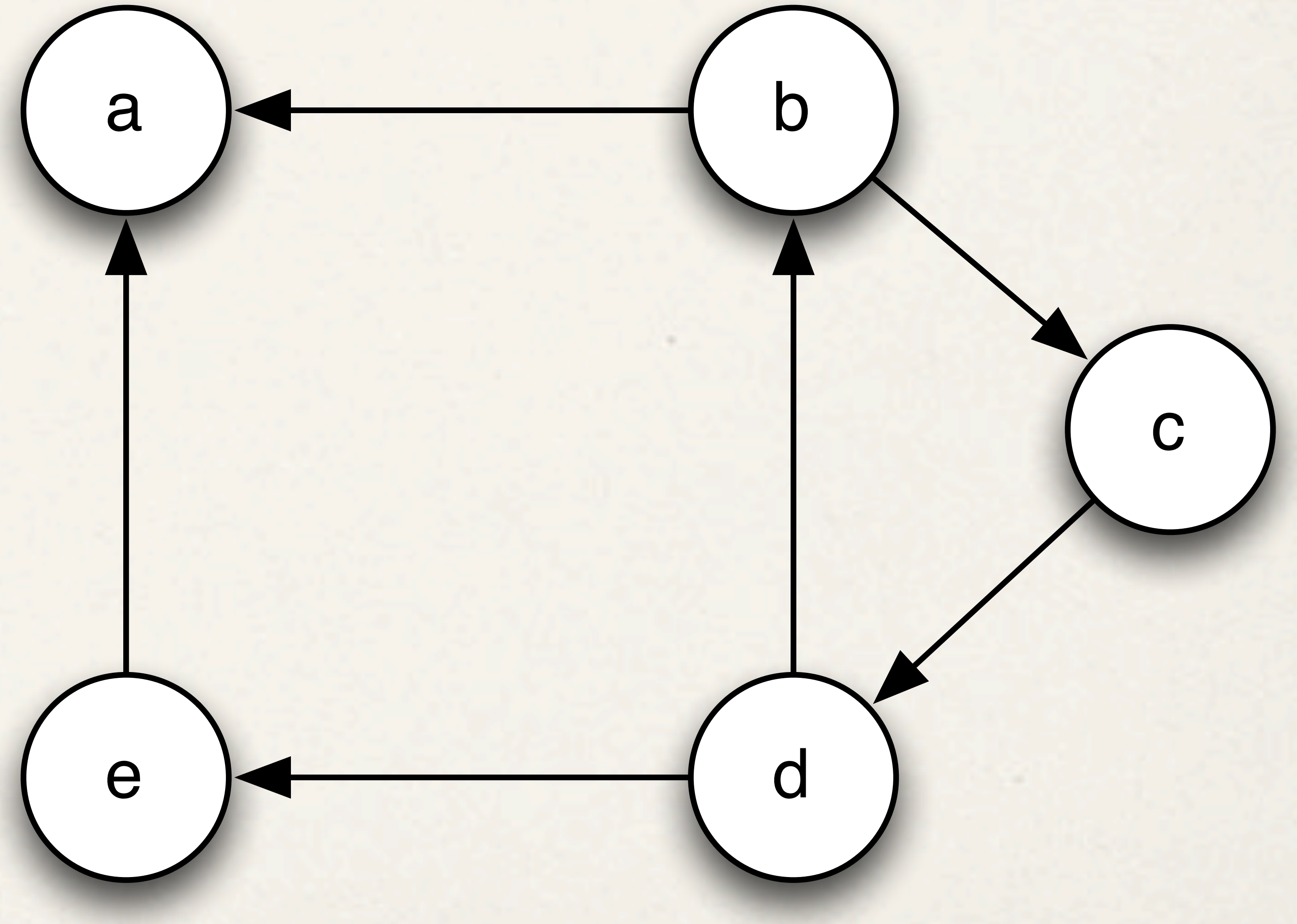


- The **underlying undirected graph** of a digraph is the same graph, minus directions
- A digraph is **weakly connected** if the underlying undirected graph is connected
    - i.e. if the digraph is "in one piece"

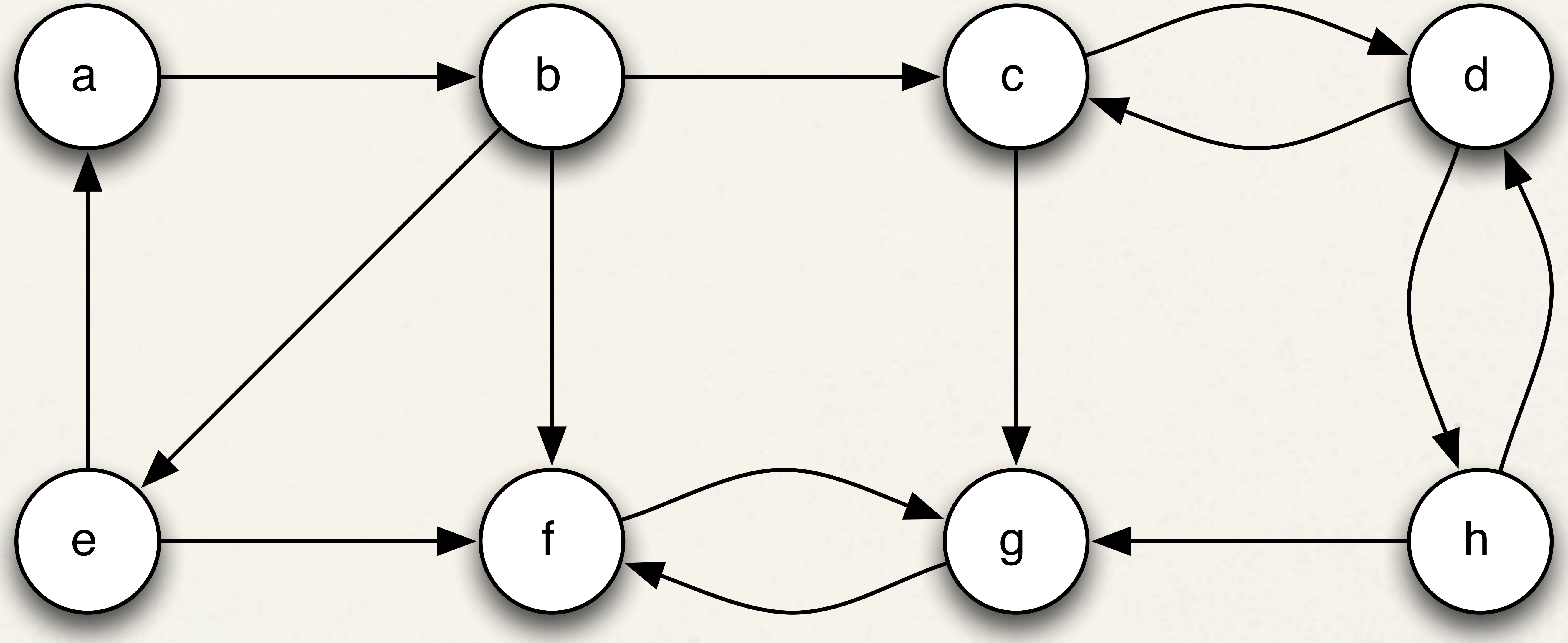
Strongly Connected



- A digraph G is **strongly connected** if for every pair of vertices $a,b \in G$, there exists a path from a to b and from b to a
    - Note: this includes the pair a,a



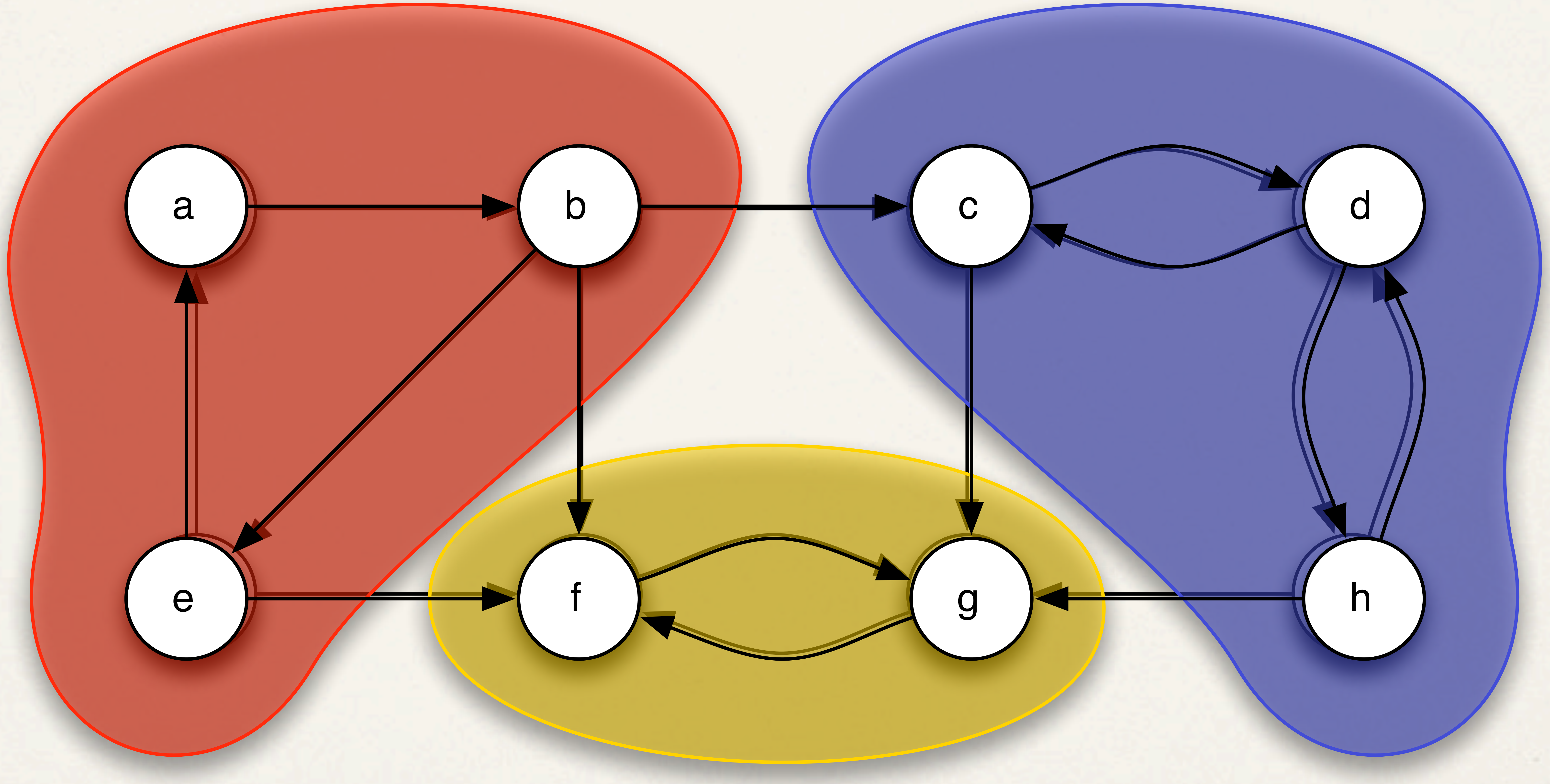
Practice



Strongly Connected Components

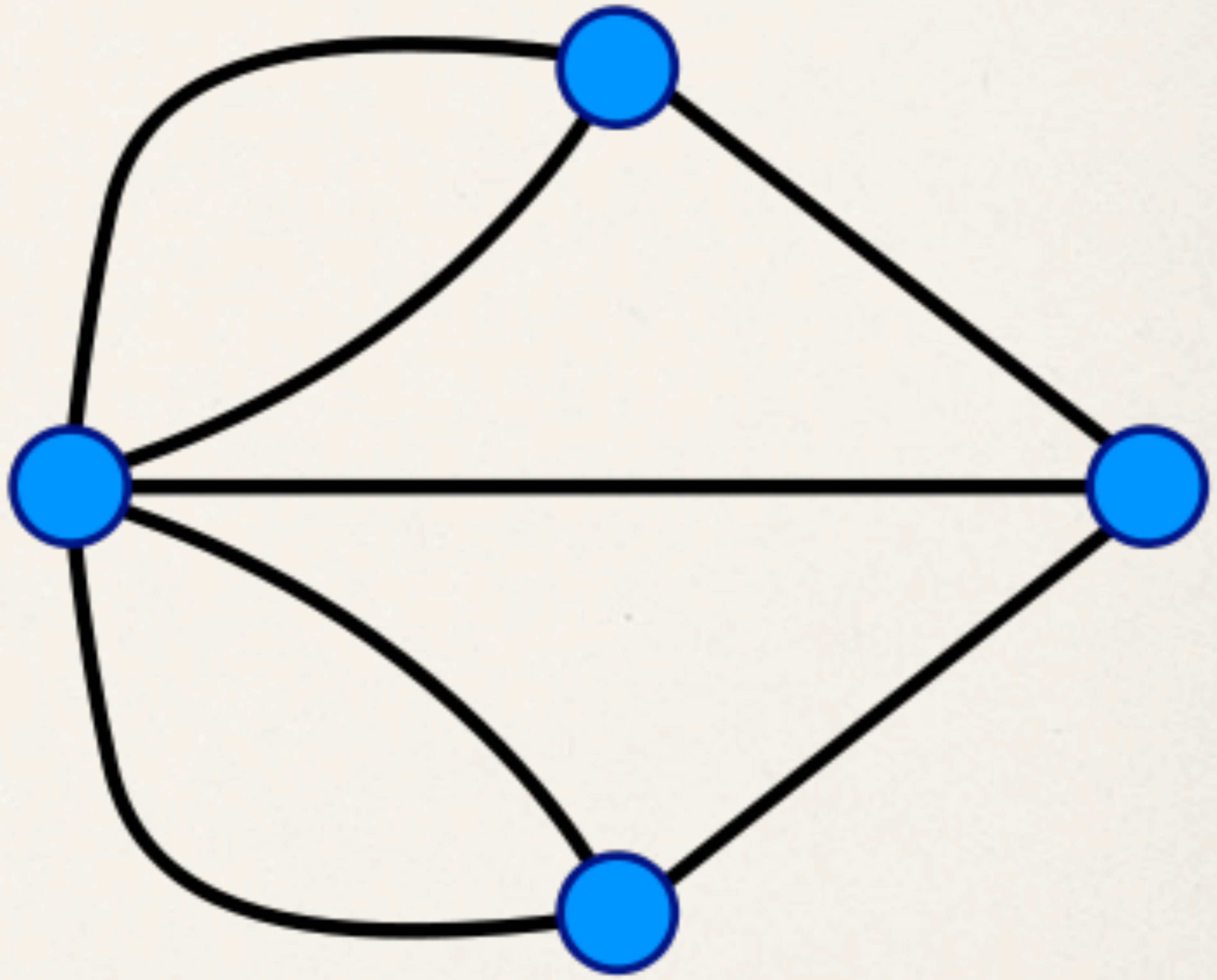
- Analogous to connected components
  - Maximal subgraphs of G which are strongly connected
    i.e. they are not contained within any other such strongly connected subgraphs
- We can **compress** a digraph into a directed acyclic graph by reducing it to a graph of its strongly connected components
  - Treat all nodes in a SCC as interchangeable
  - Model how to hop from one SCC to another

Strongly Connected Components



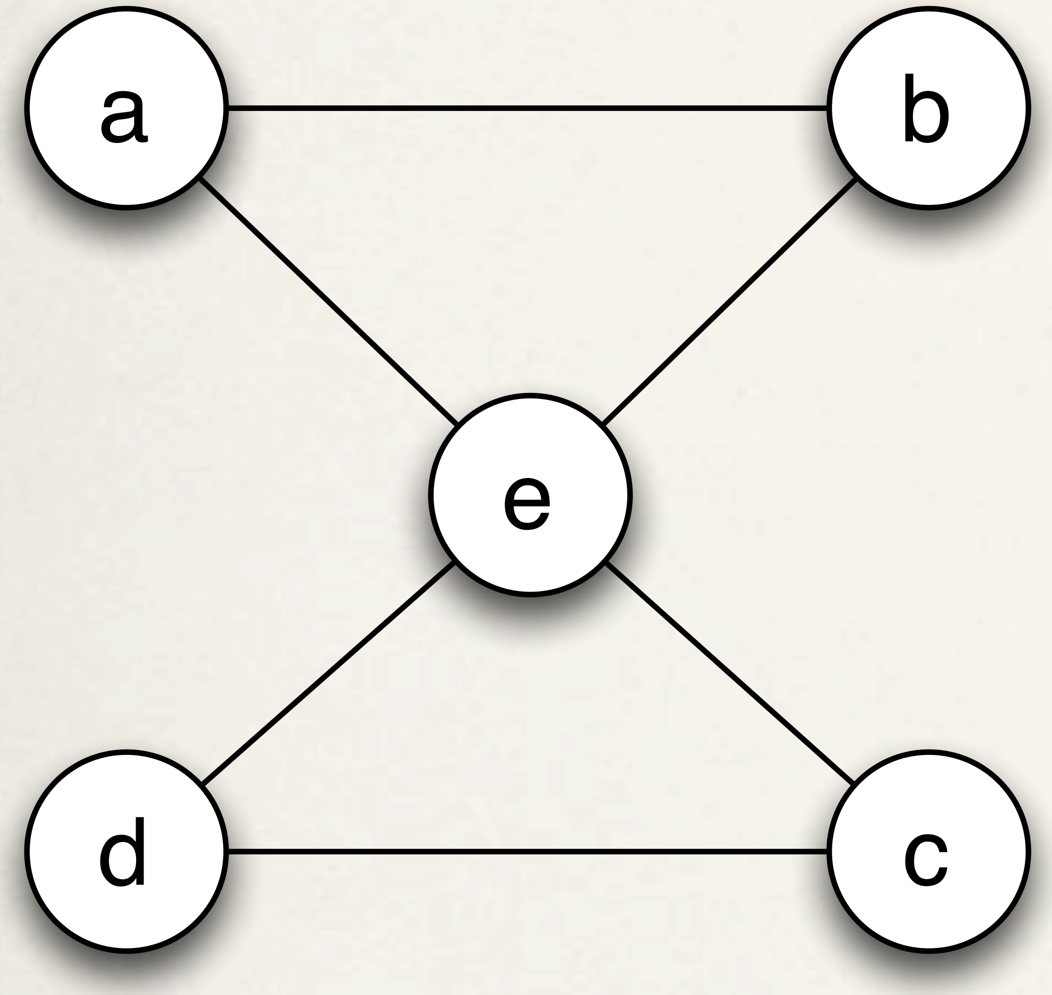
Euler and Hamilton Paths/Cycles

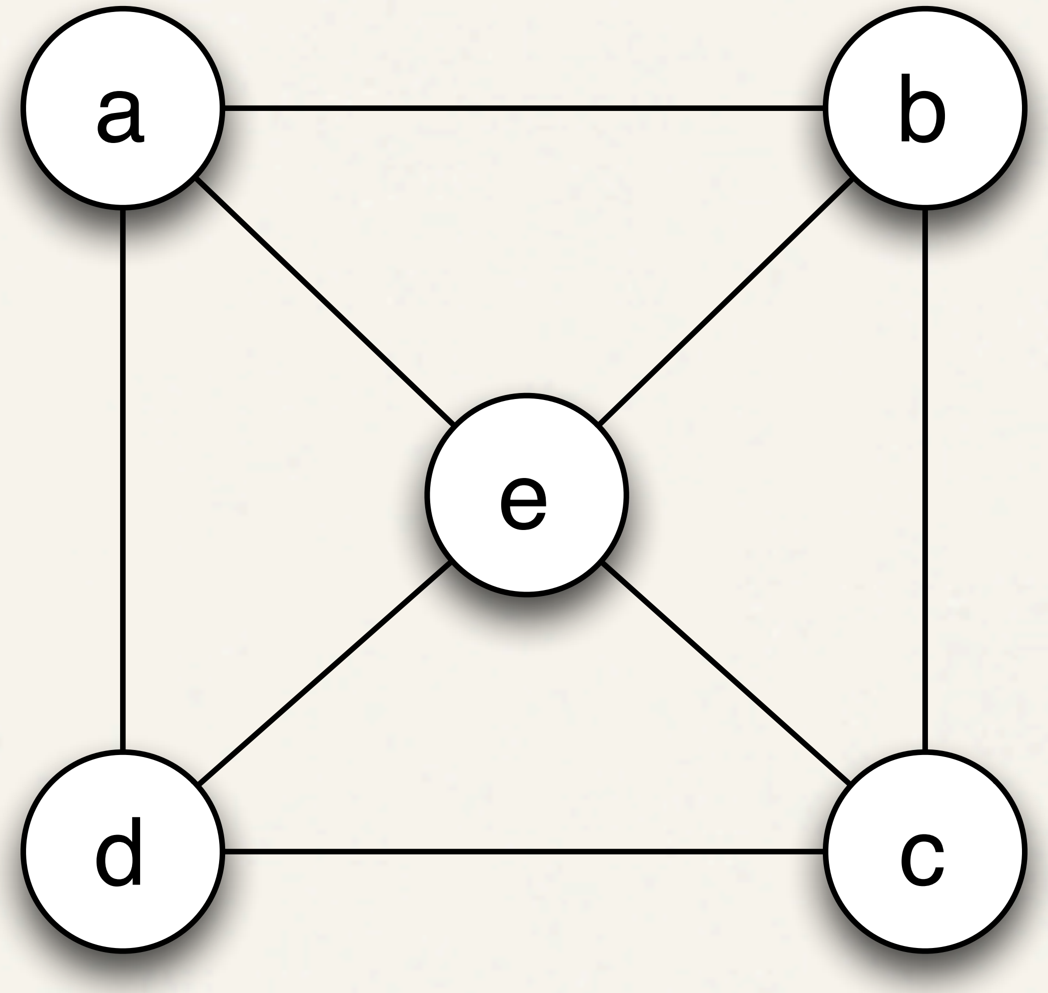
Euler Paths and Circuits

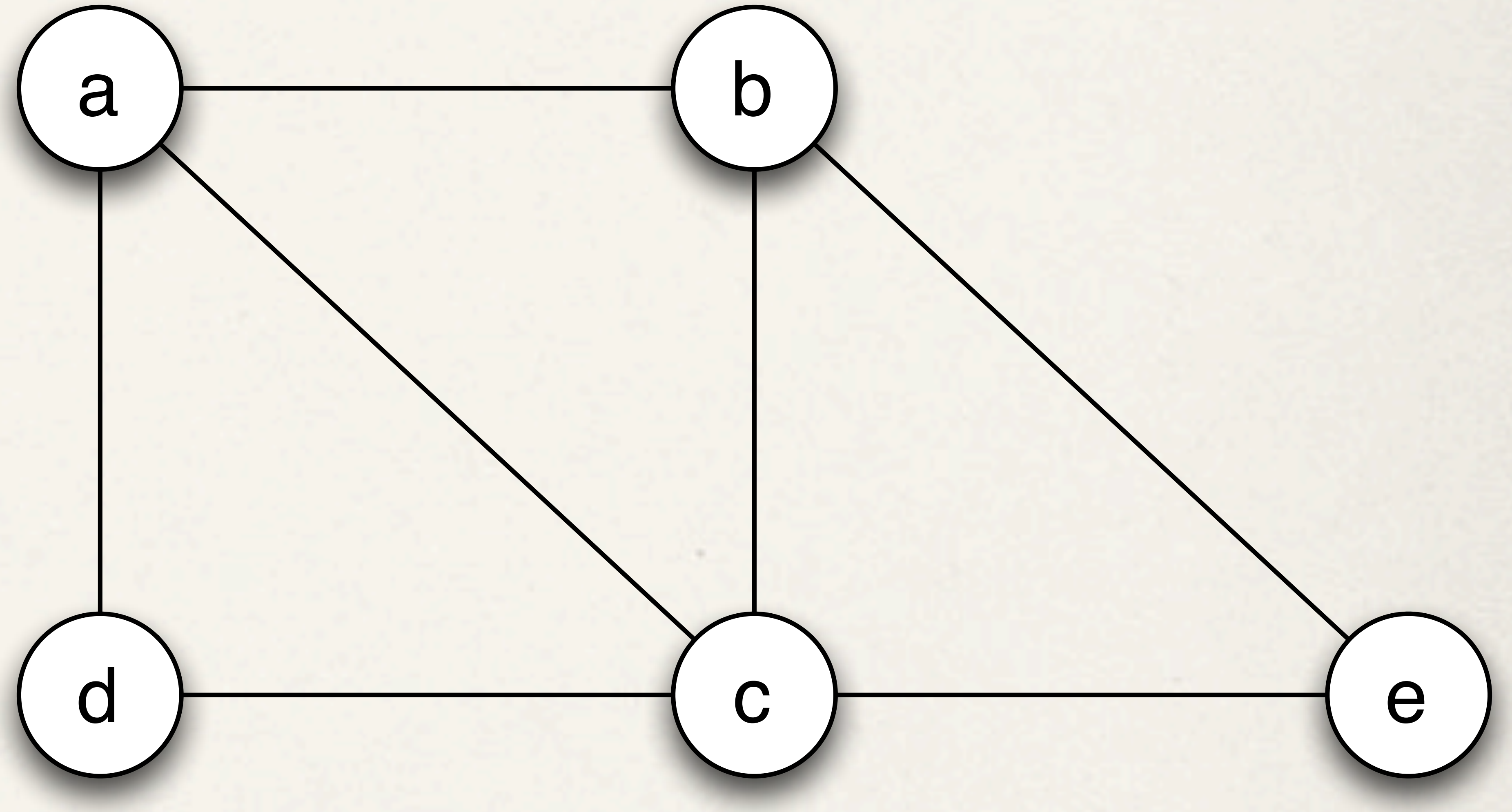


- **Euler Path**: path in *G* which uses every edge exactly once
  - can visit a node more than once
- **Euler Circuit**: Euler path which is also a circuit

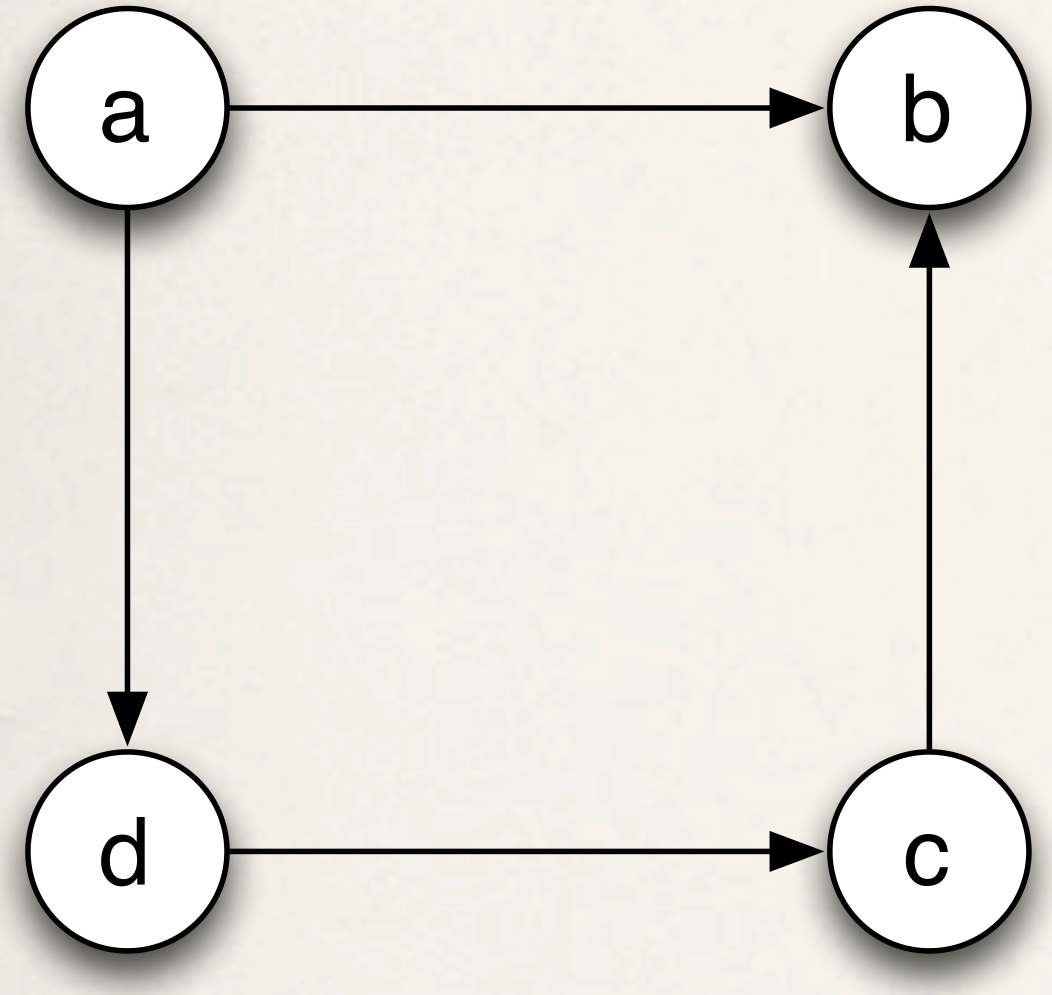
Practice

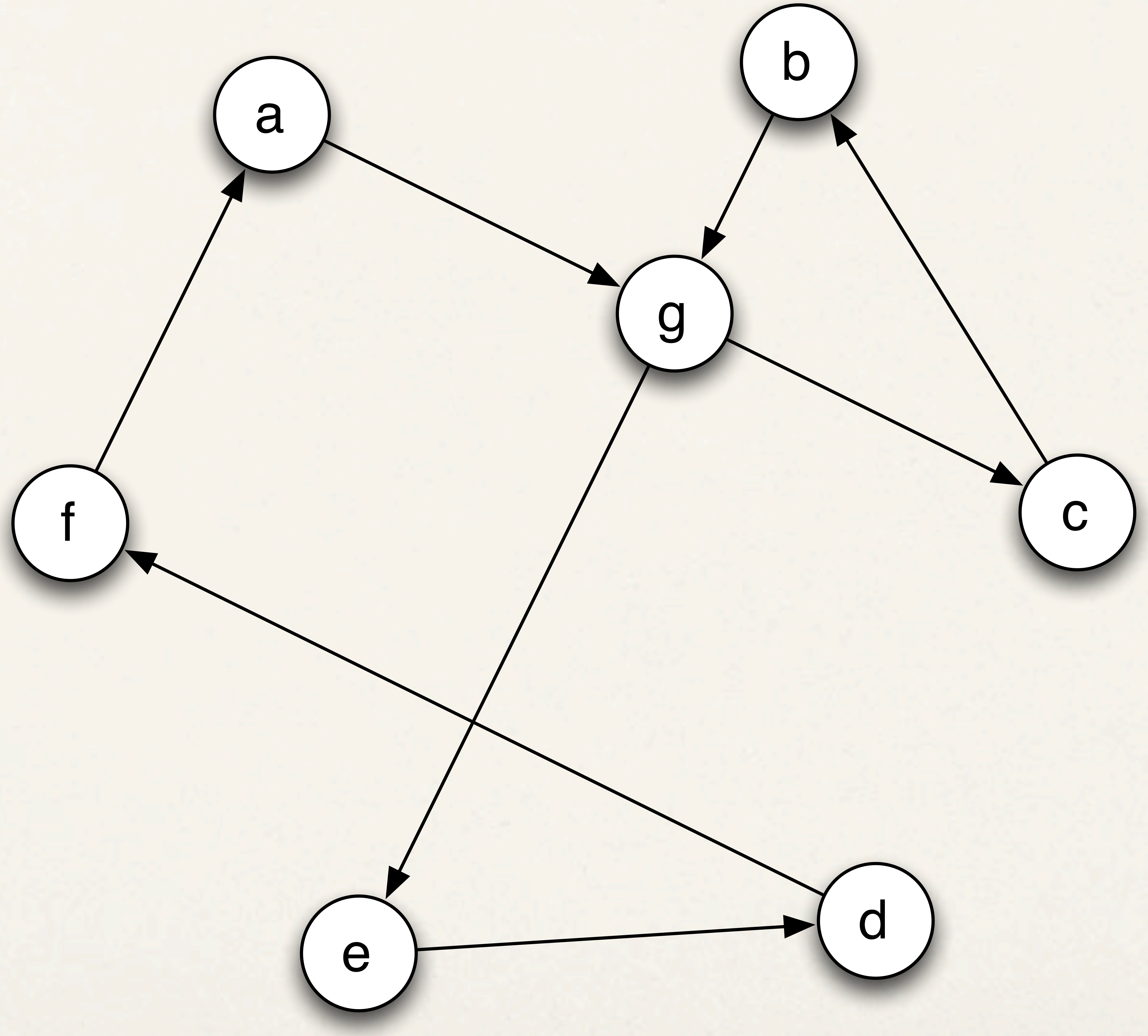


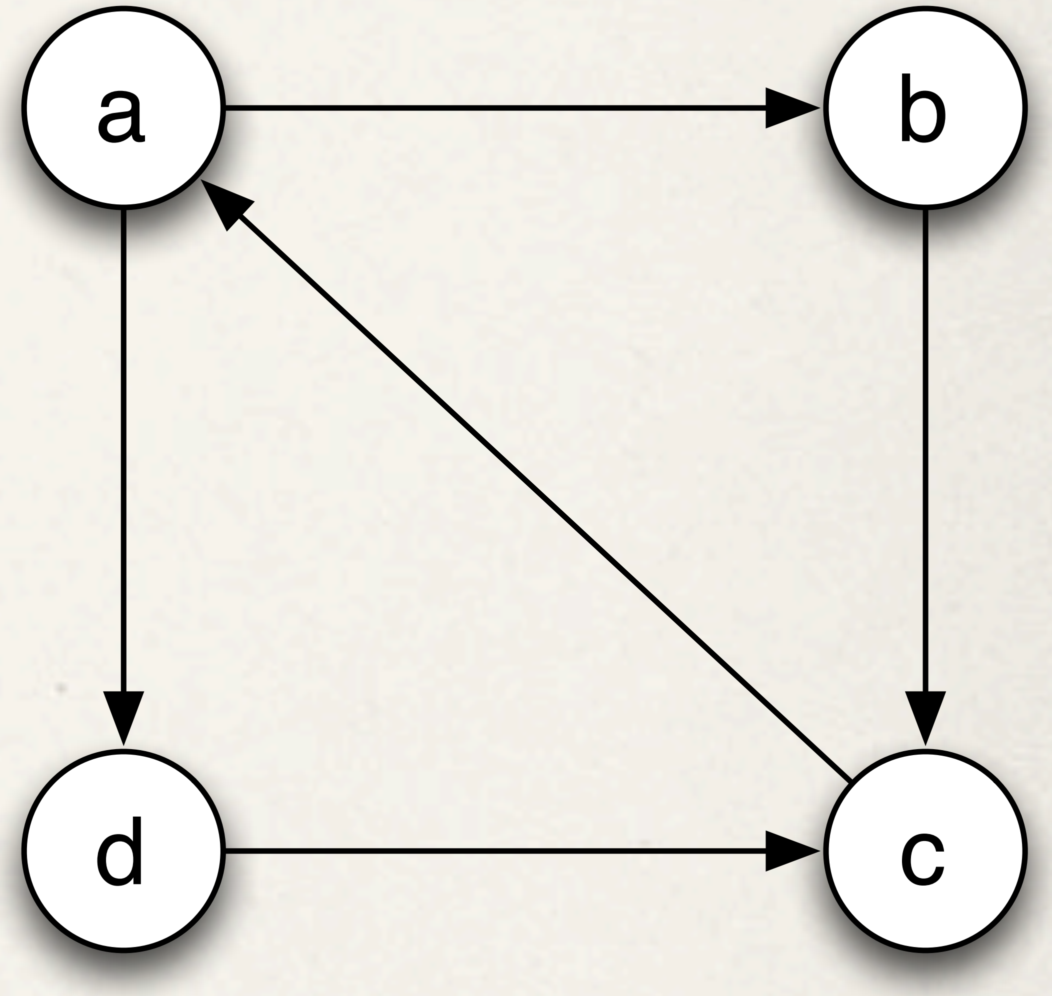




Practice

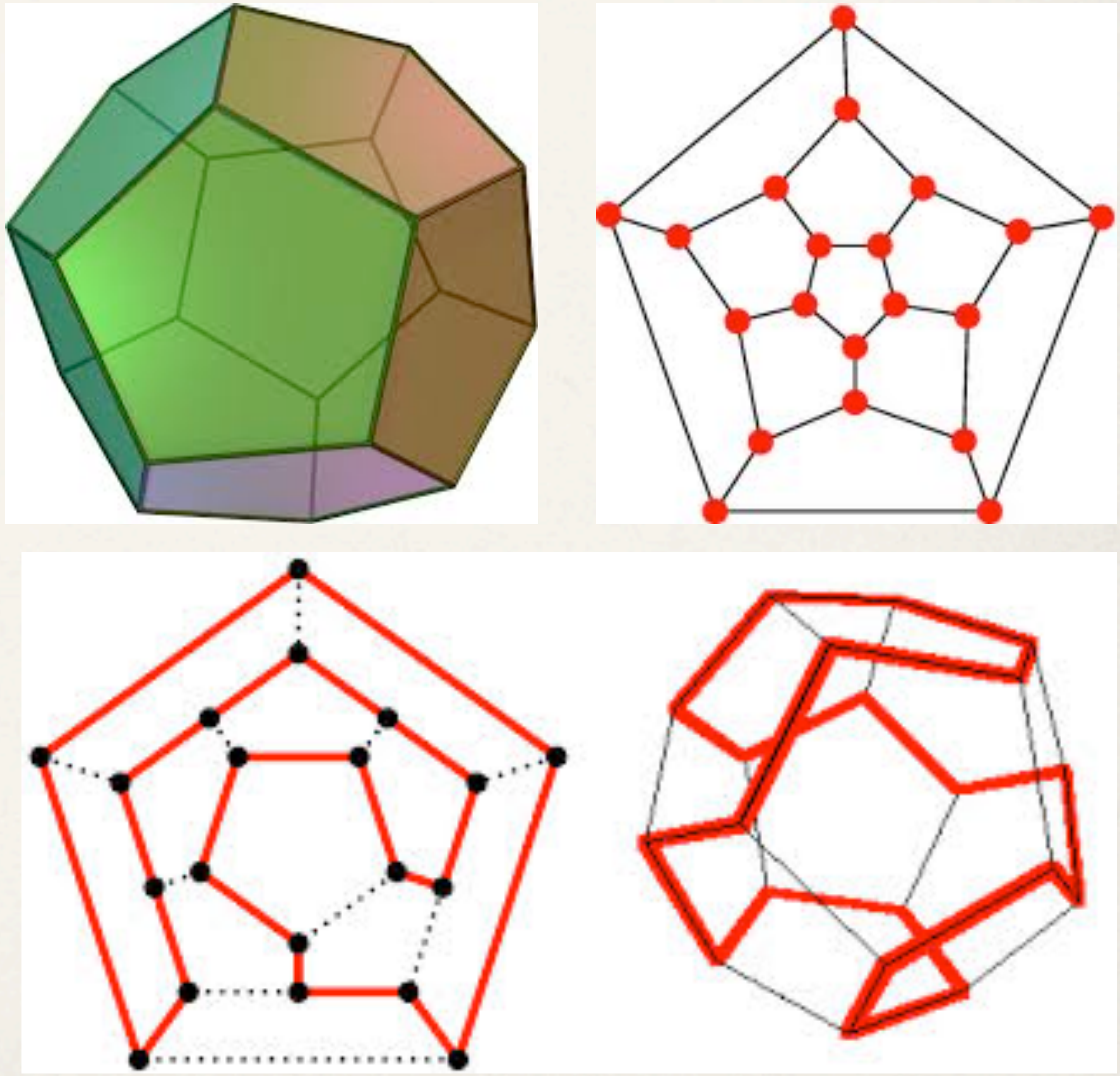




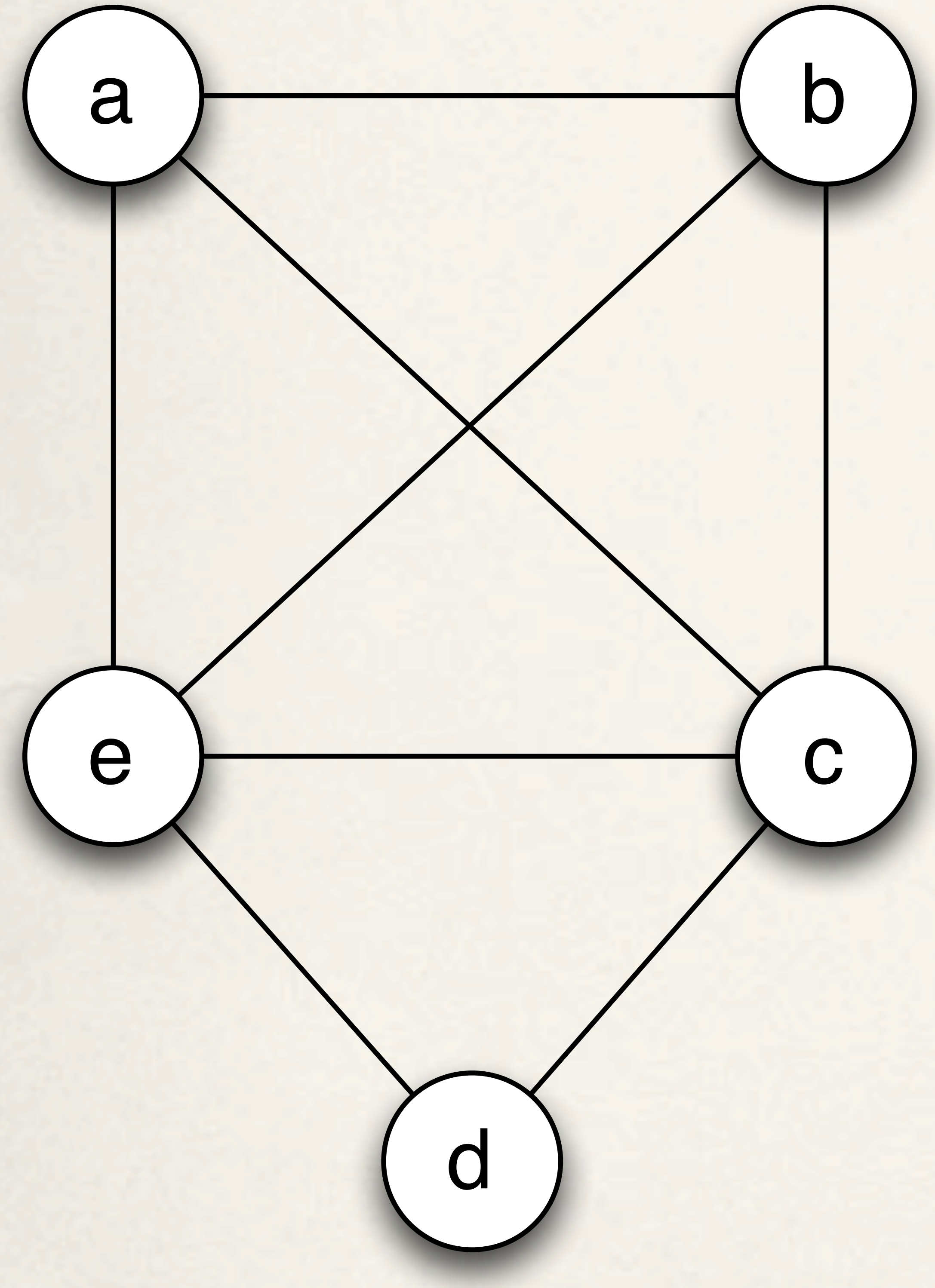


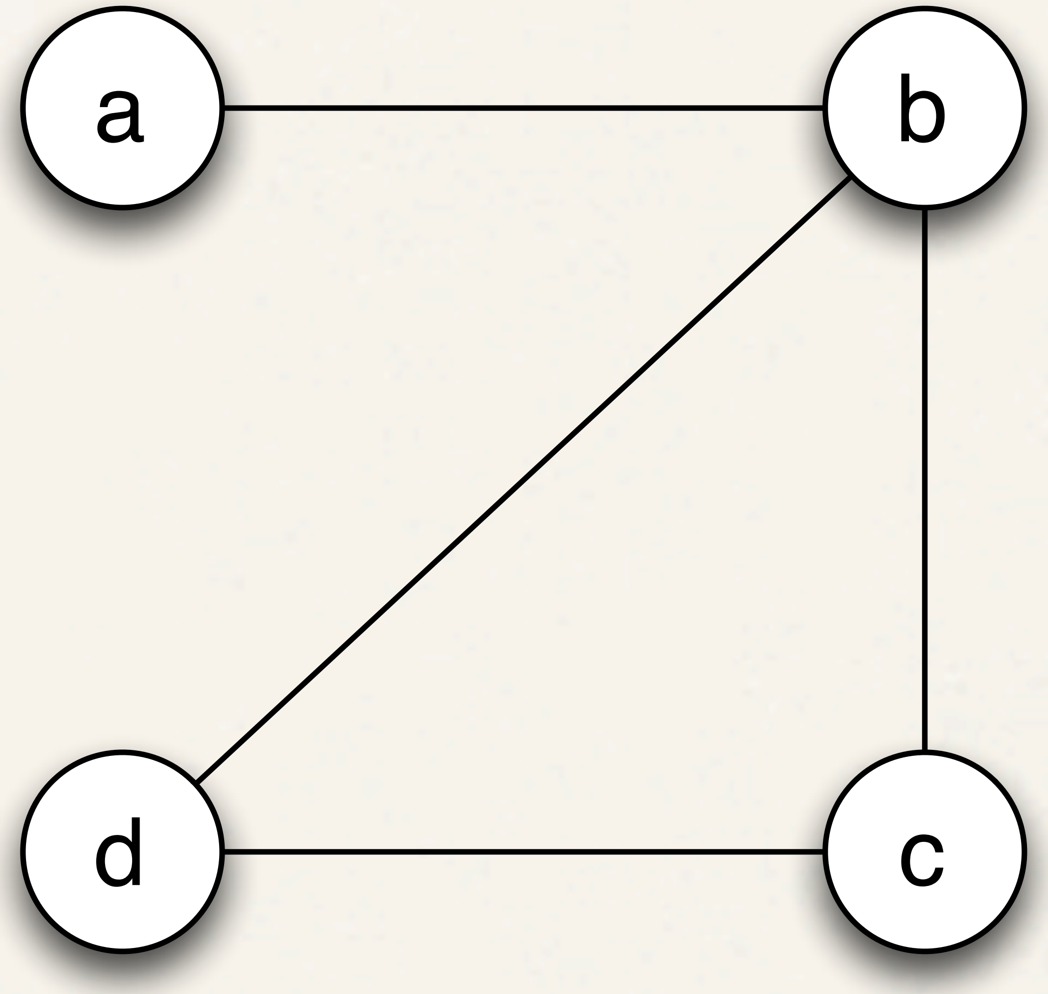
Hamilton Paths and Circuits

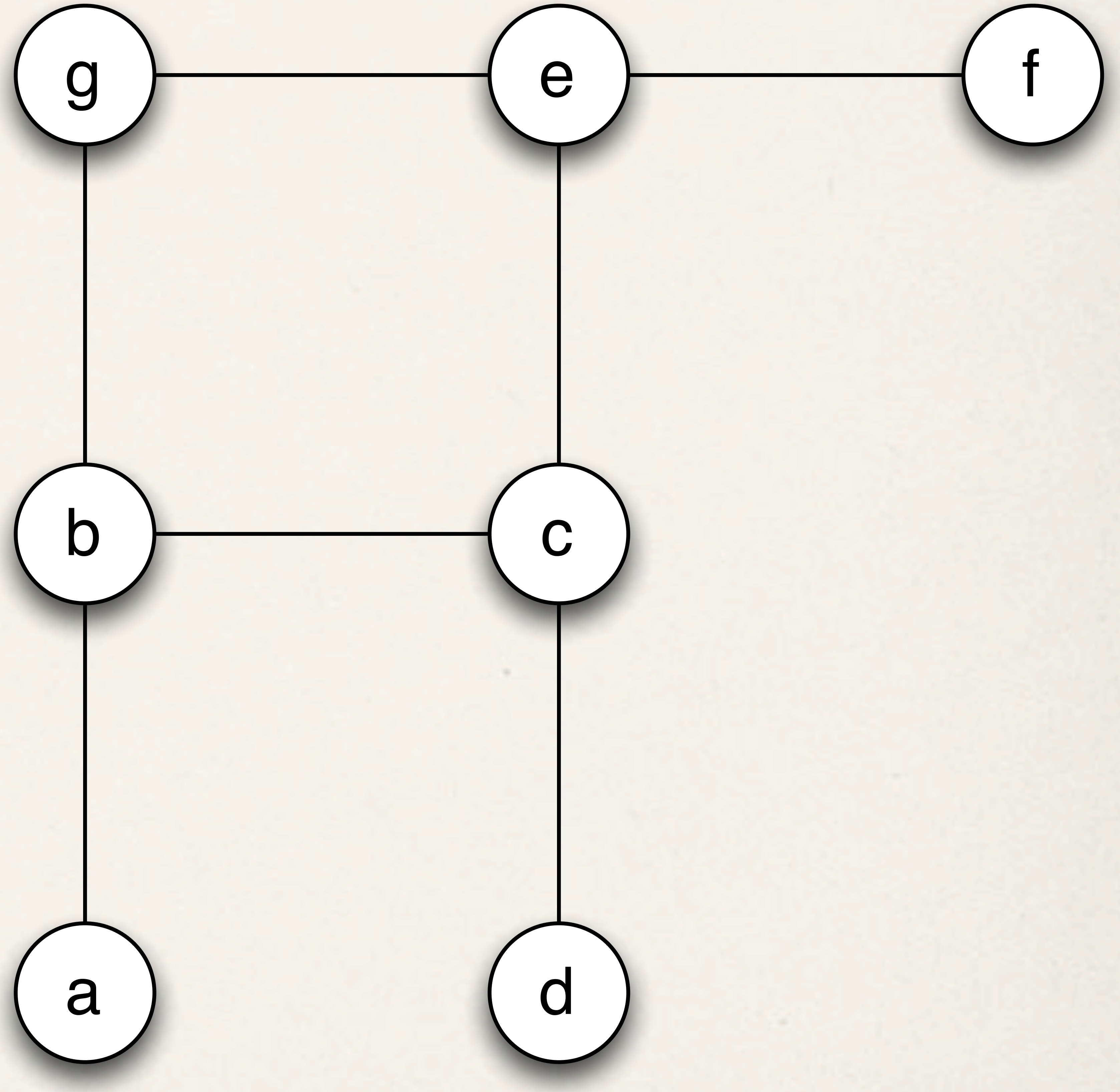
- **Hamilton Path**: path visiting every node exactly once
  - each edge can only be used once
- **Hamilton Circuit**: hamilton path which is also a circuit



Practice

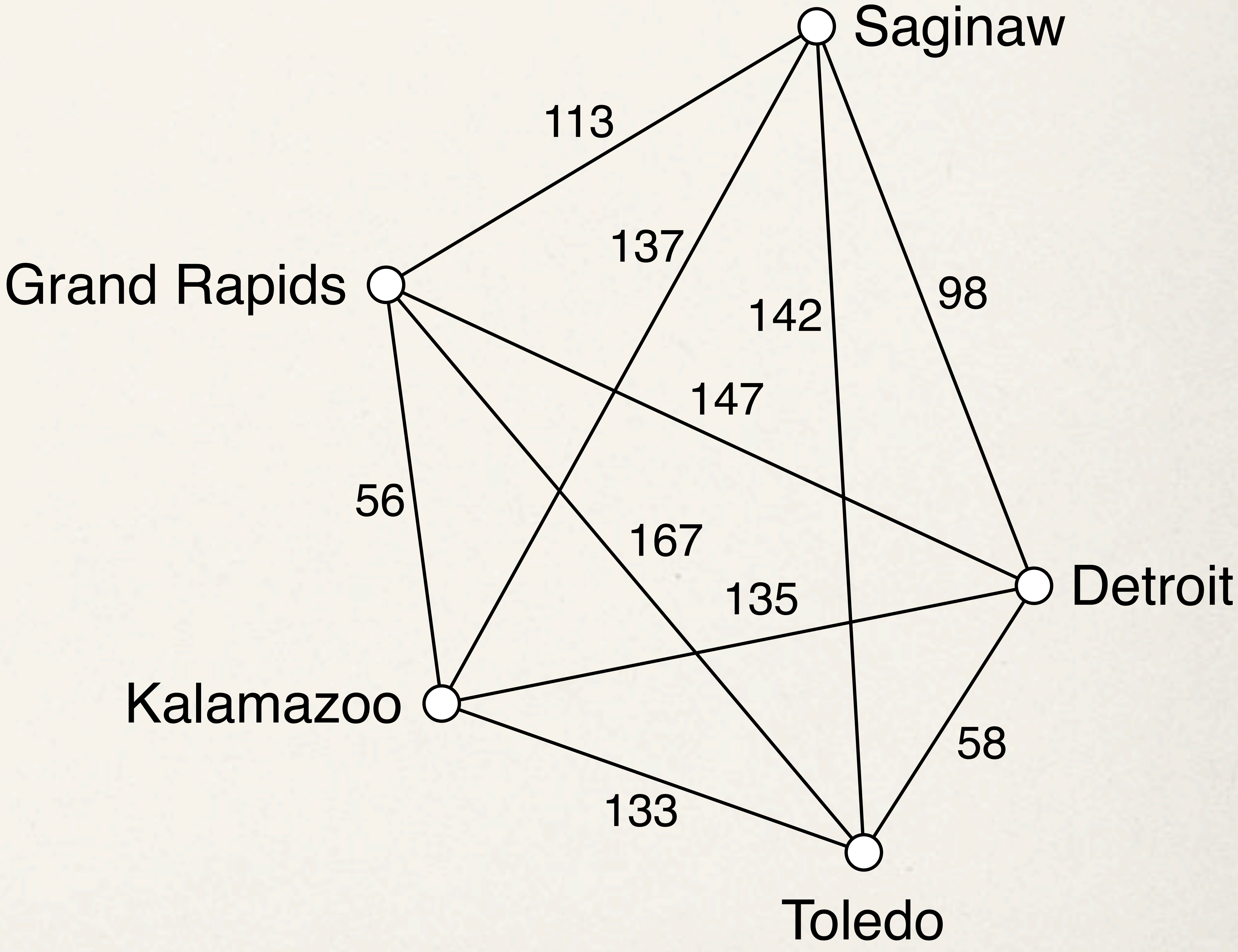






Traveling Salesperson Problem

- **Traveling Salesperson Problem (TSP)** or **Traveling Salesman Problem**
  - Find the shortest hamilton cycle
  - i.e. visiting every node once and returning to the start



Traveling Salesperson Problem

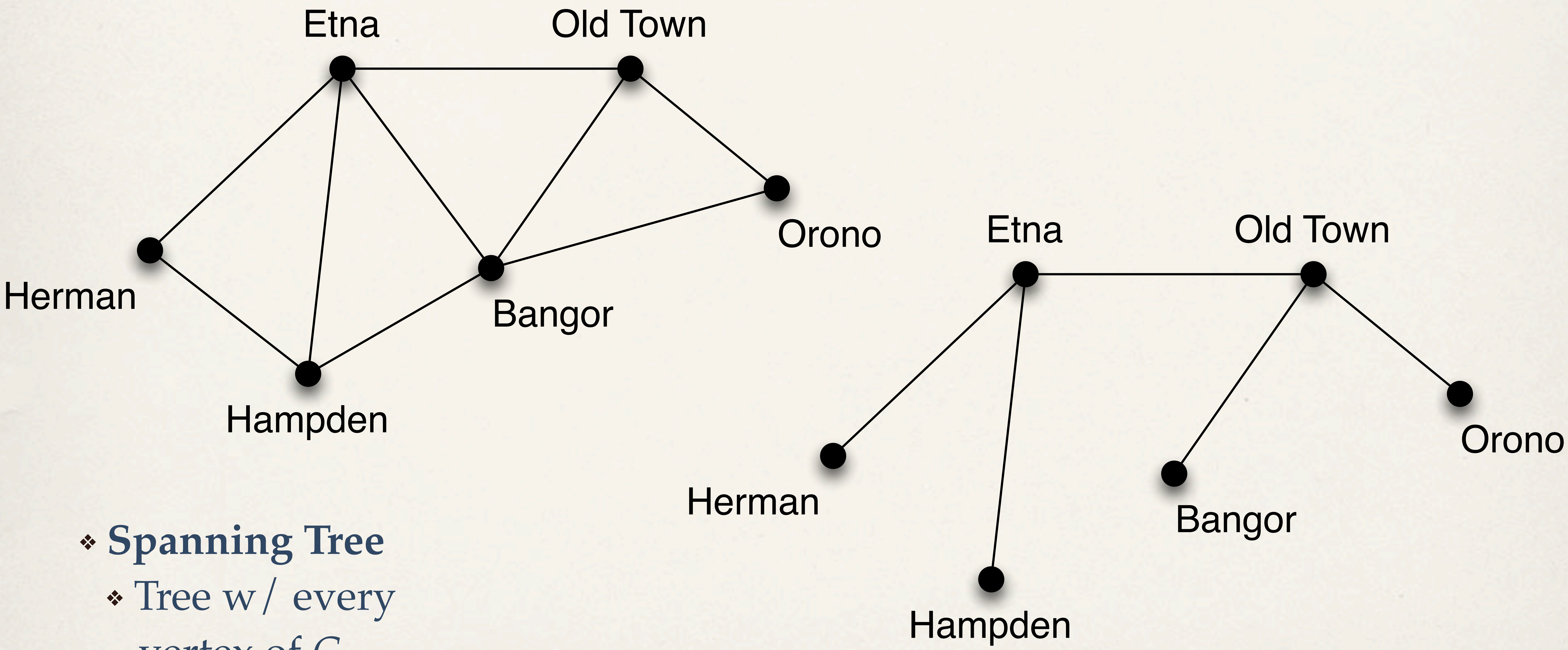
- This is an **NP-hard problem**
  - There are $(n - 1)!/2$ Hamilton cycles
  - $O(n!)$ to compute exhaustively
  - FedEx pays lots and lots of money for improvements on this problem
    - Often okay to find an **approximate** solution

Route	Total
D - T - GR - S - K - D	610
D - T - GR - K - S - D	516
D - T - K - S - GR - D	588
D - T - K - GR - S - D	458
D - T - S - K - GR - D	540
D - T - S - GR - K - D	504
D - S - T - GR - K - D	598
D - S - T - K - GR - D	576
D - S - K - GR - T - D	682
D - S - GR - T - K - D	646
D - GR - S - T - K - D	670
D - GR - T - S - K - D	728

Spanning Trees

Section 9.5

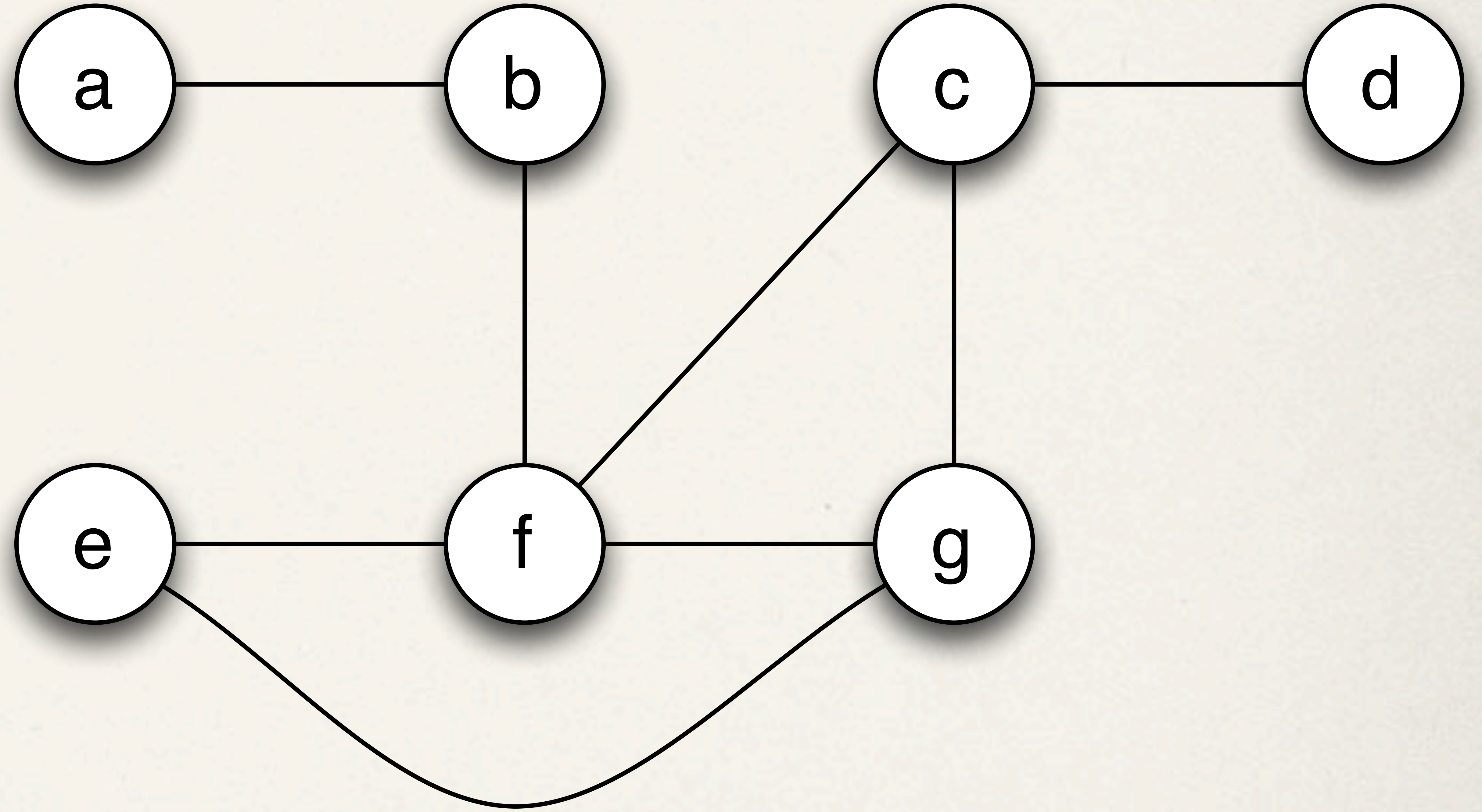
Spanning Tree



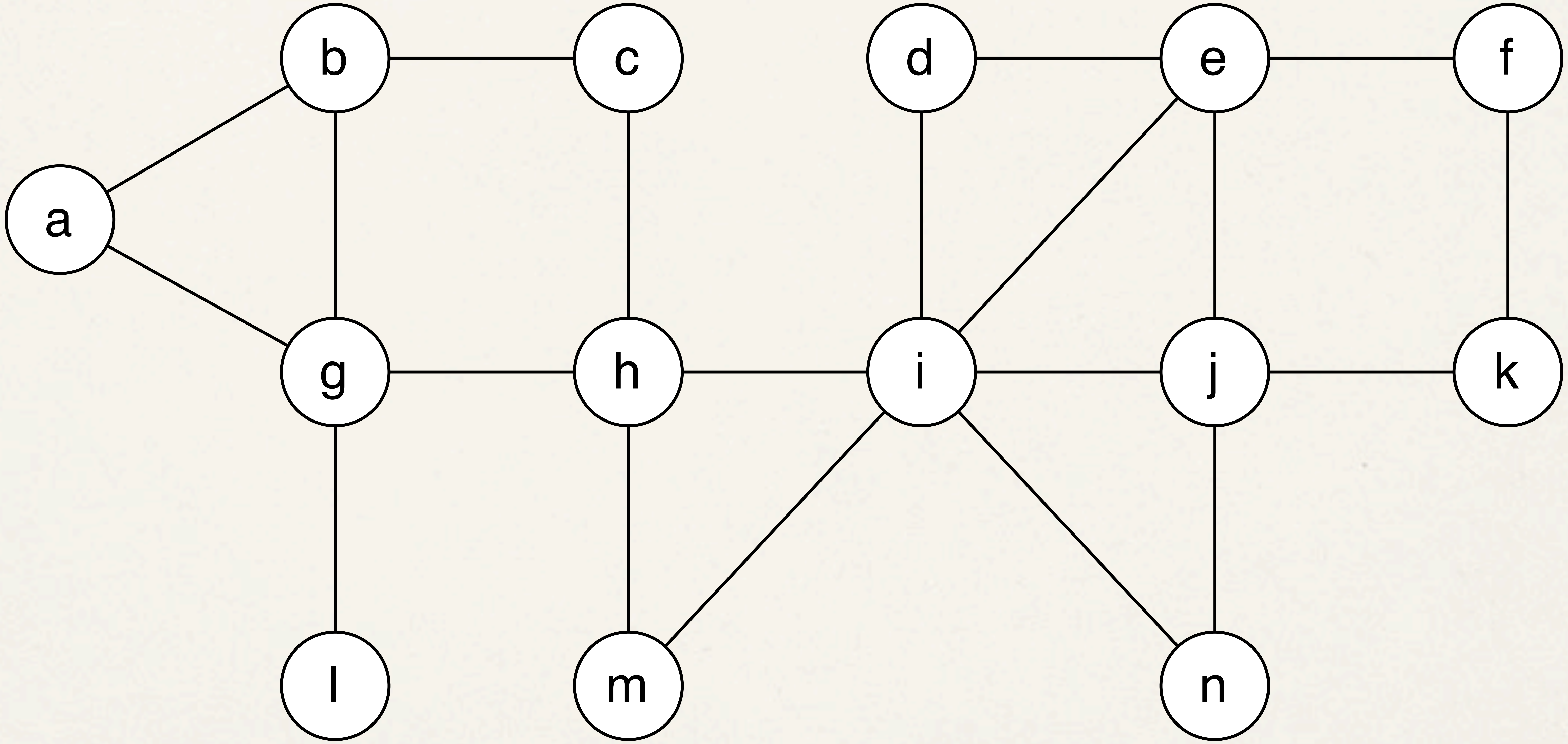
- **Spanning Tree**
  - Tree w/ every vertex of G

Finding Spanning Trees

- Trees are acyclic, graphs aren't
- Remove edges to break up cycles
    - Keep graph connected

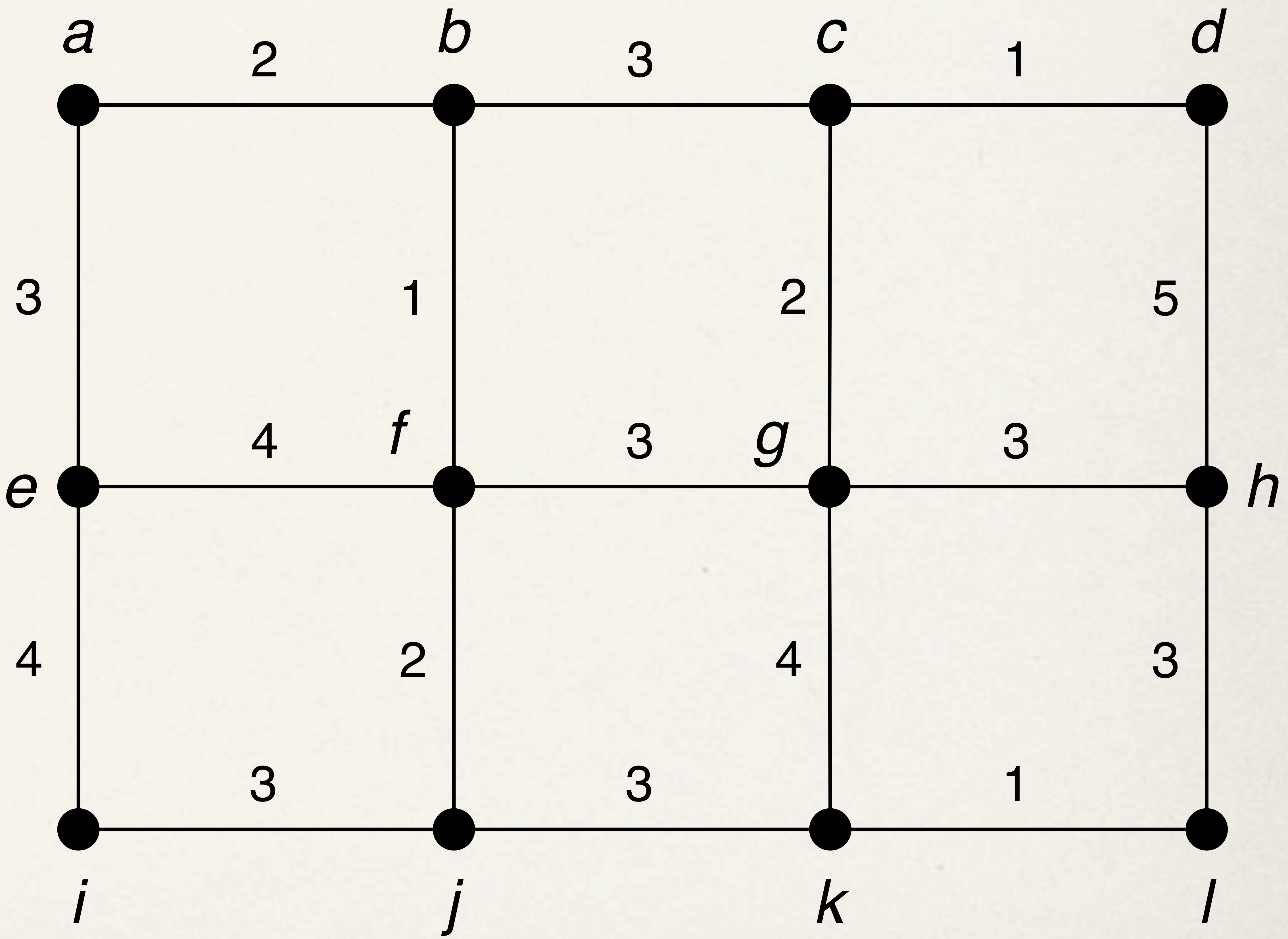


Practice



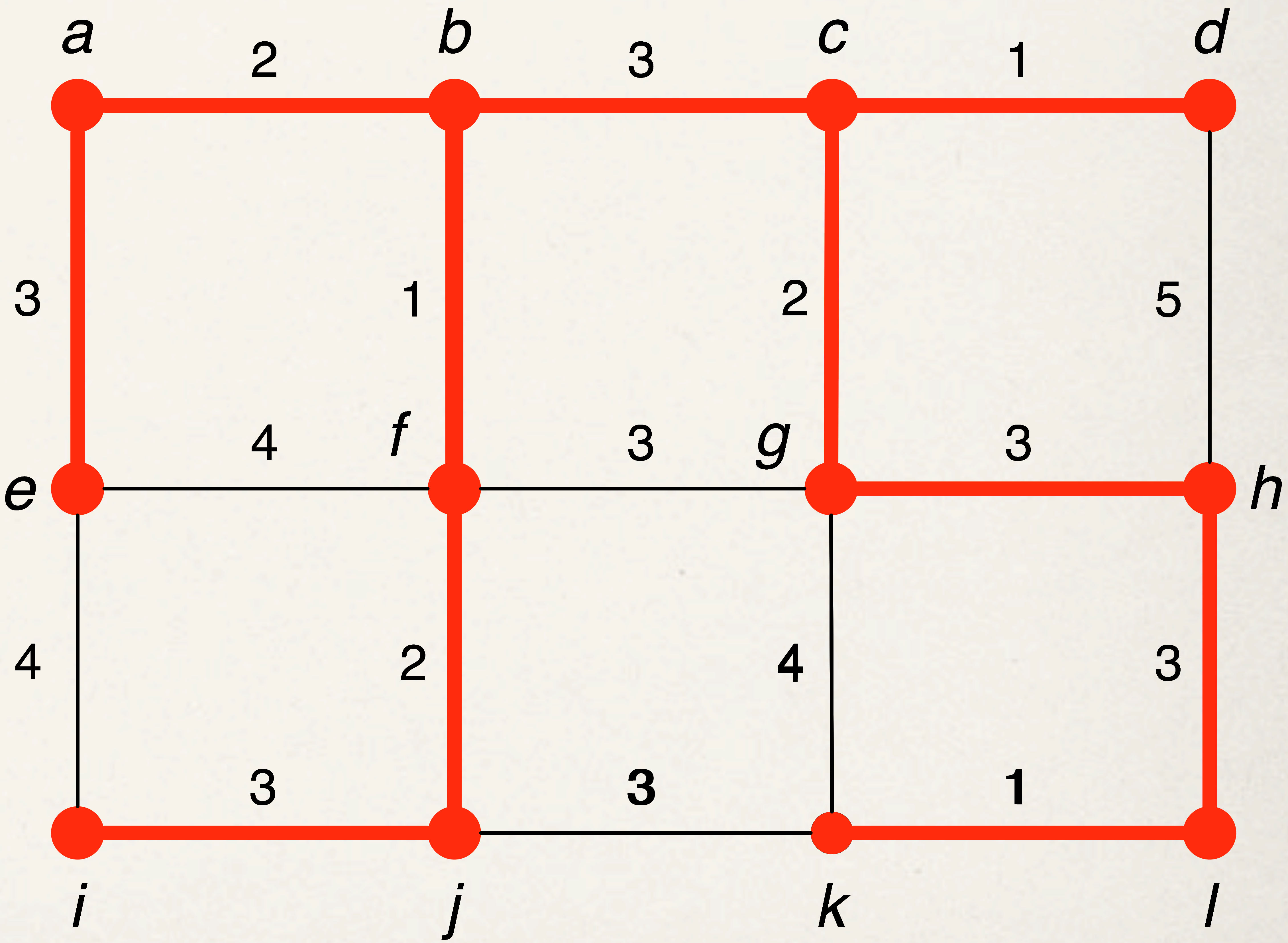
Minimum Spanning Trees

- Weighted graphs
- Find spanning tree with the smallest possible sum of edge weights
- Applications:
  - Making connected graphs



Prim's Algorithm

```
procedure Prim (G)
T := a minimum weight edge
for i := 1 to n - 2
    e := an edge of minimum weight,
         incident to a vertex in T,
         not forming a cycle in T
    T := T with e added
return T
```



Kruskal's Algorithm

```
procedure Kruskal (G)
T := an empty graph
for i := 1 to n - 1
   e := any edge of minimum weight in G,
         not forming a cycle if added to T
   T := T with e added
return T
```