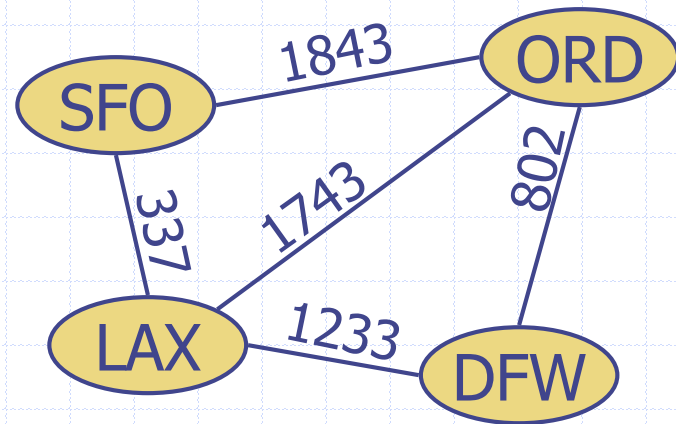


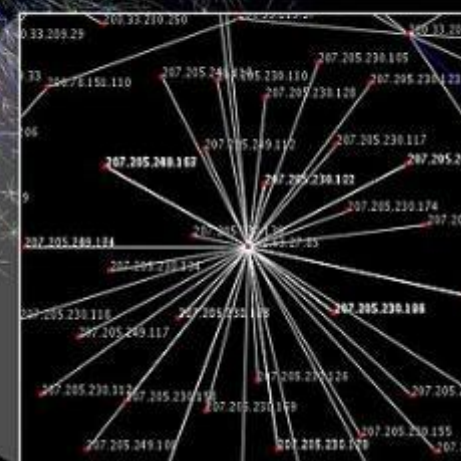
Graphs



Graphs

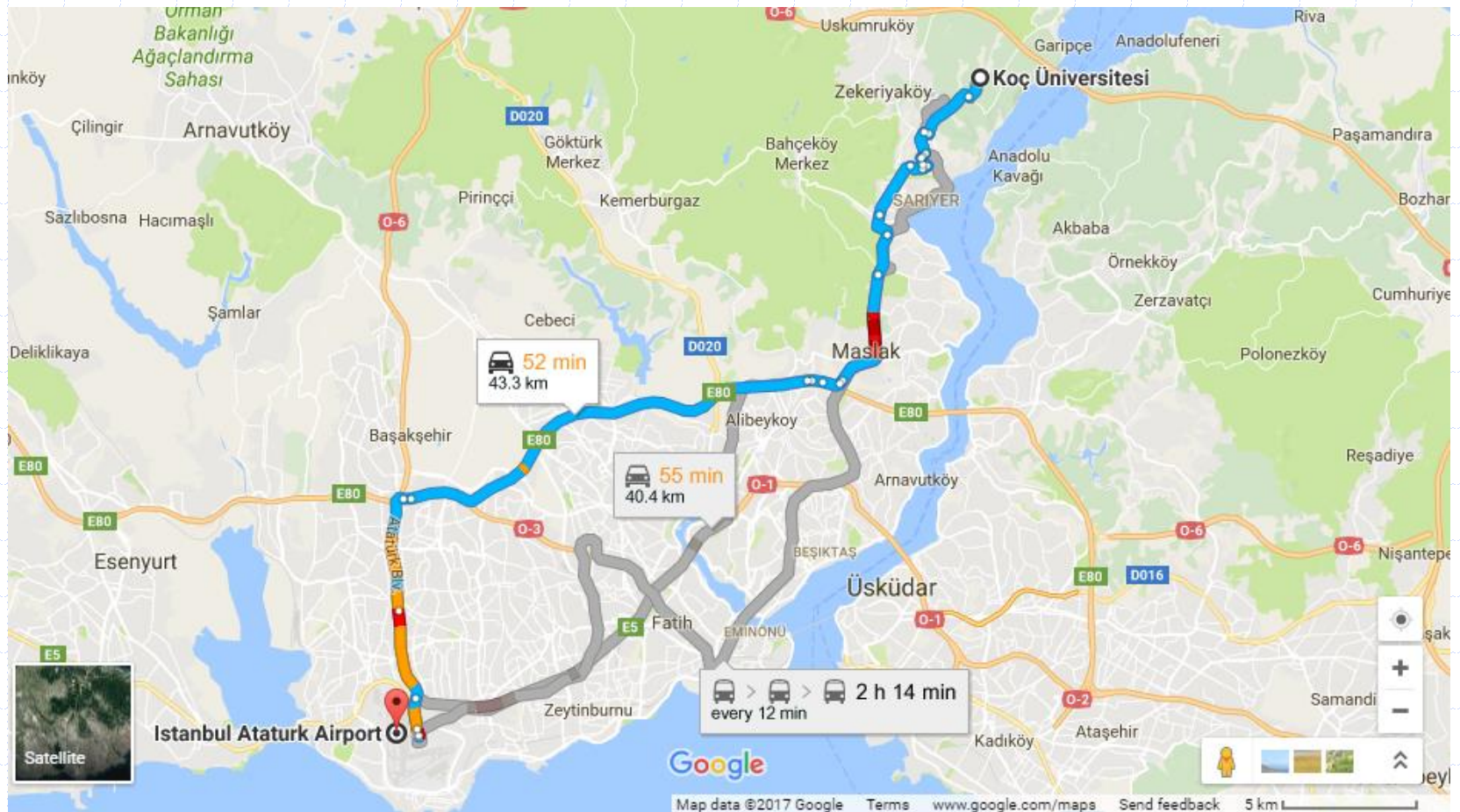
- ◆ A graph is a set of **vertices** connected by **edges**
- ◆ $G = (V, E)$
 - V: The set of vertices
 - E: The set of edges, where each edge is a pair (v, w) s.t. $v, w \in V$





The graph of the internet

Navigation



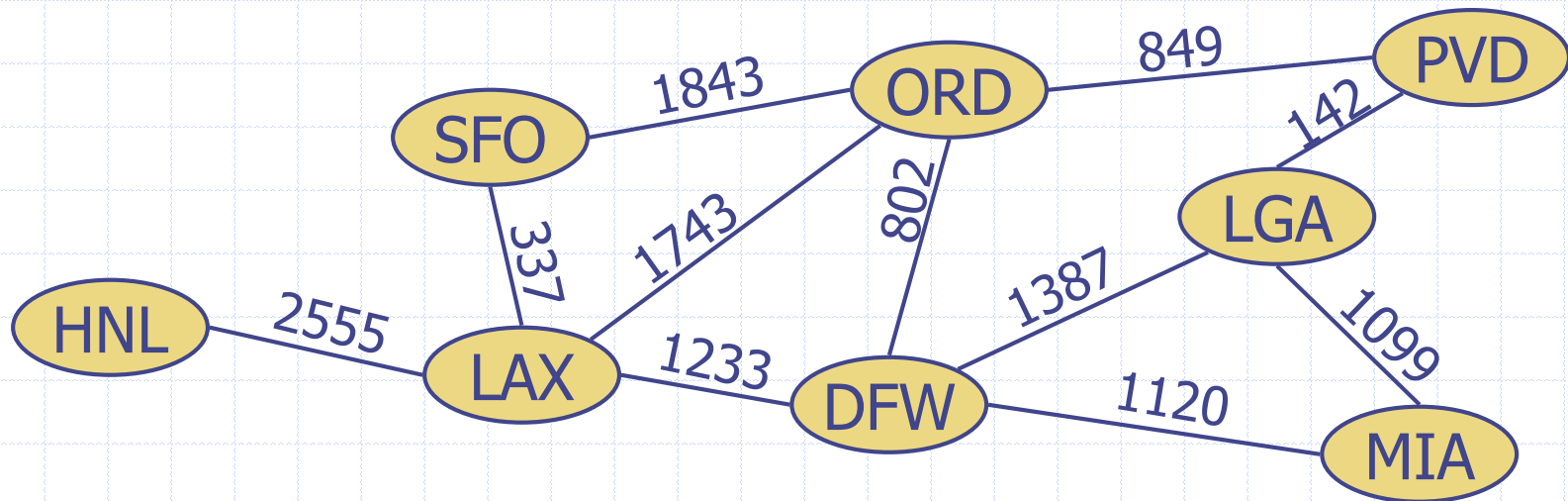
Facebook – 10 Million People



Visualizing Friendships – Paul Butler

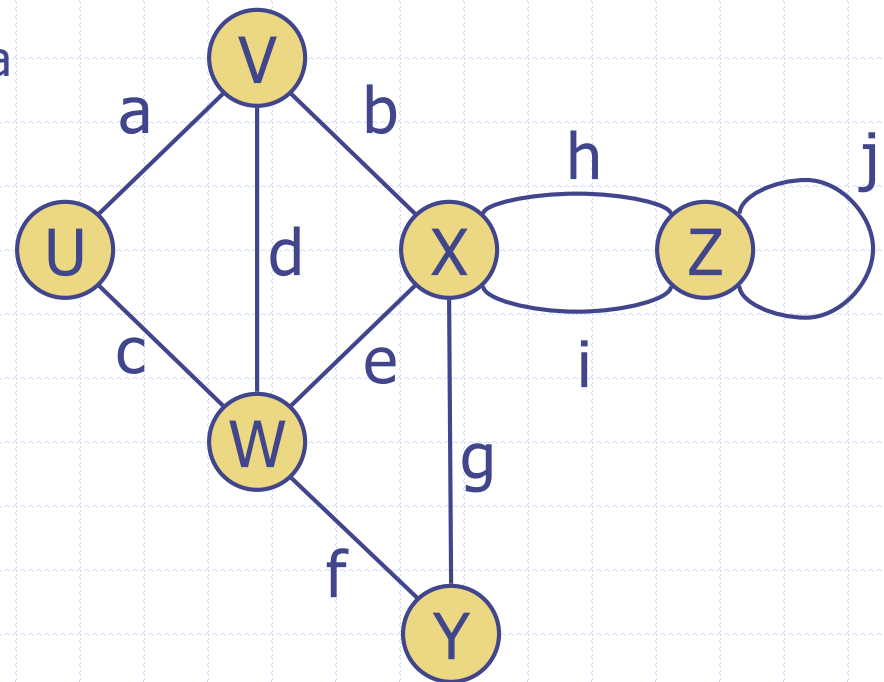
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

Working Example – Flight Routes



Terminology

- ◆ End vertices (or endpoints) of an edge
 - U and V are the endpoints of a
- ◆ Edges incident on a vertex
 - a, d, and b are incident on V
- ◆ Adjacent vertices
 - U and V are adjacent
- ◆ Degree of a vertex
 - X has degree 5
- ◆ Parallel edges
 - h and i are parallel edges
- ◆ Self-loop
 - j is a self-loop



Terminology

◆ Path

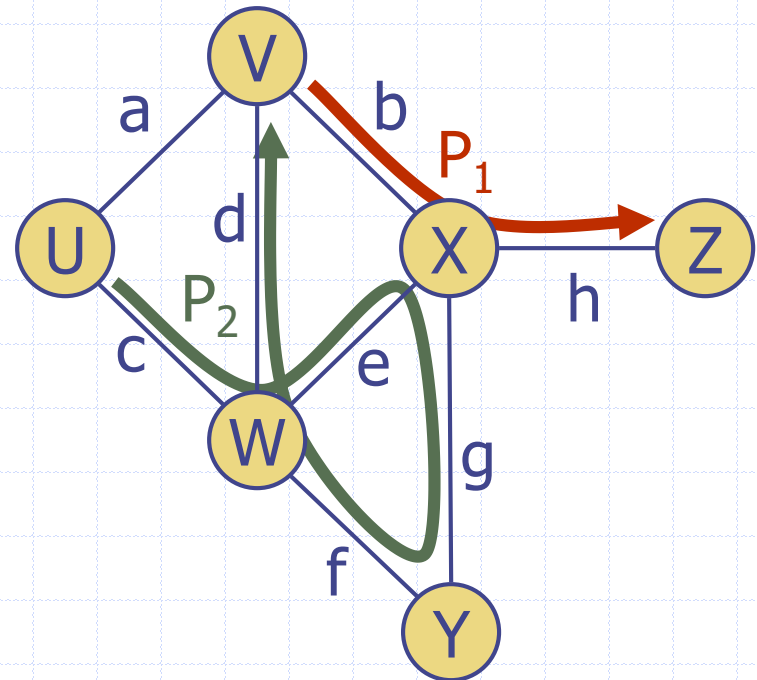
- sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge is preceded and followed by its endpoints

◆ Simple path

- path such that all its vertices and edges are distinct

◆ Examples

- $P_1 = (V, b, X, h, Z)$ is a simple path
- $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ is a path that is not simple



Terminology

◆ Cycle

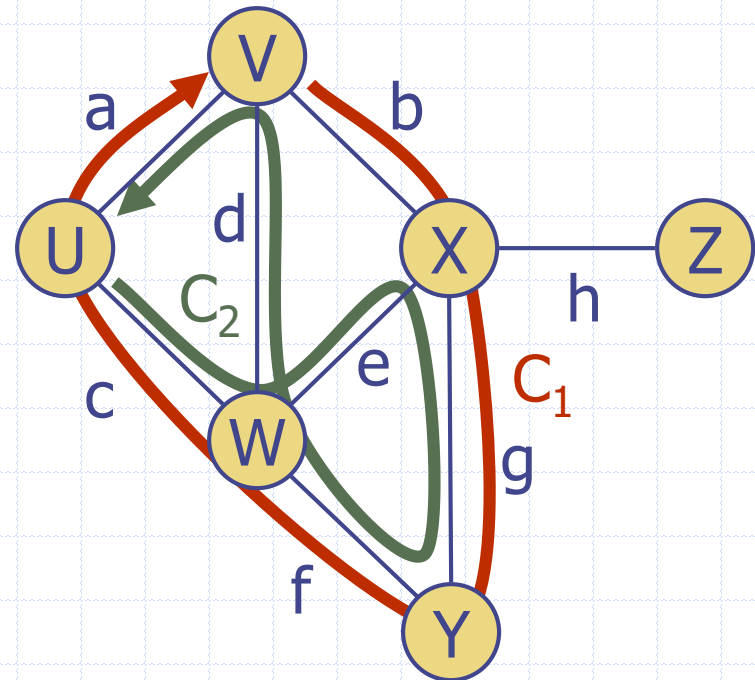
- circular sequence of alternating vertices and edges
- each edge is preceded and followed by its endpoints

◆ Simple cycle

- cycle such that all its vertices and edges are distinct

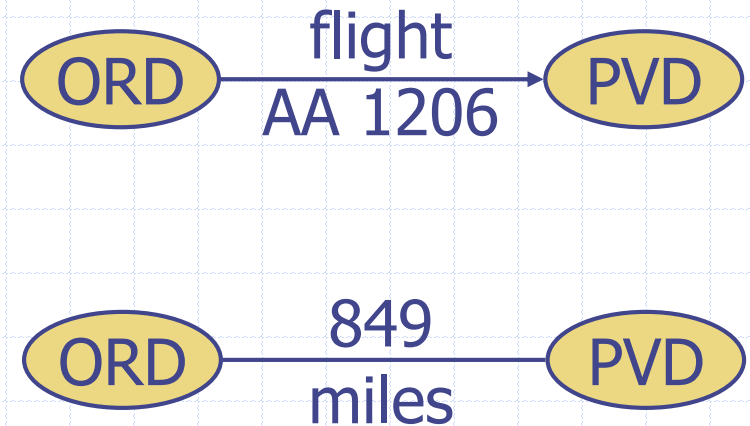
◆ Examples

- $C_1 = (V, b, X, g, Y, f, W, c, U, a, \downarrow)$ is a simple cycle
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \downarrow)$ is a cycle that is not simple



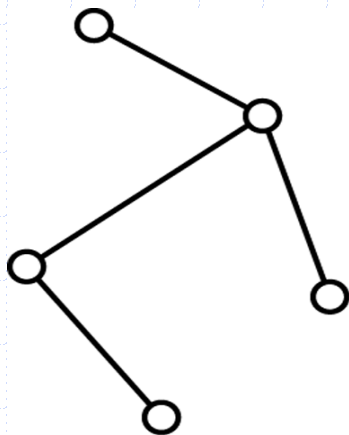
Edge Types

- ◆ Directed edge
 - ordered pair of vertices (u,v)
 - first vertex u is the origin
 - second vertex v is the destination
 - e.g., a flight
- ◆ Undirected edge
 - unordered pair of vertices (u,v)
 - e.g., a flight route
- ◆ Directed graph
 - all the edges are directed
 - e.g., route network
- ◆ Undirected graph
 - all the edges are undirected
 - e.g., flight network

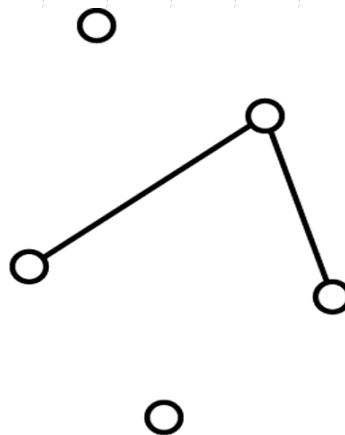


Graph Connectedness

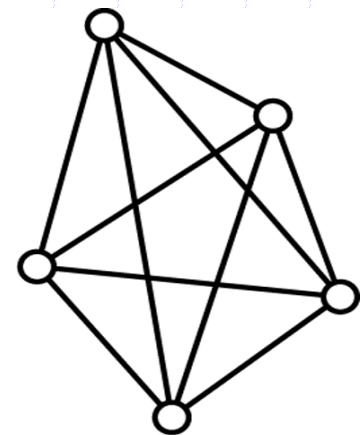
- ◆ A **connected graph** has a path between each pair of distinct vertices.
- ◆ A **complete graph** has an edge between each pair of distinct vertices.
 - A complete graph is also a connected graph. But a connected graph may not be a complete graph.



(a) **connected**



(b) **disconnected**



(c) **complete**

Vertices and Edges

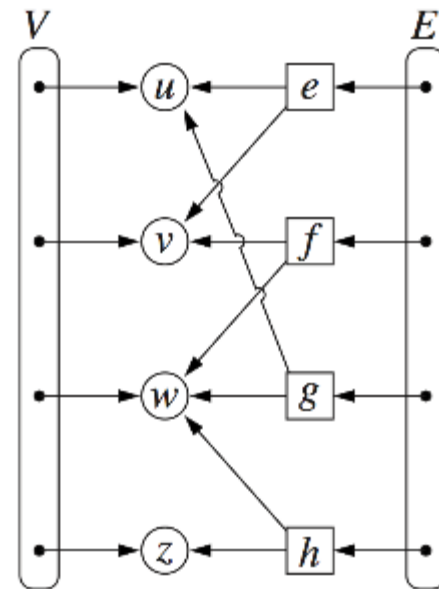
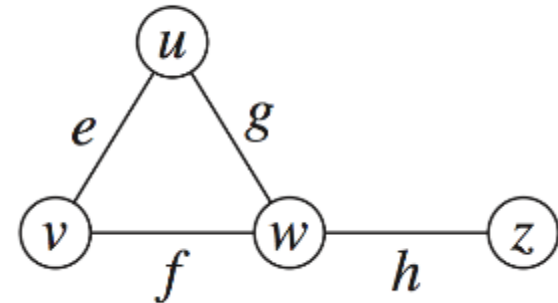
- ◆ A **graph** is a collection of **vertices** and **edges**.
- ◆ We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- ◆ A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
 - We assume it supports a method, `element()`, to retrieve the stored element.
- ◆ An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the `element()` method.

How to Implement Graphs?

- ◆ Edge List
- ◆ Adjacency List
- ◆ Adjacency Matrix

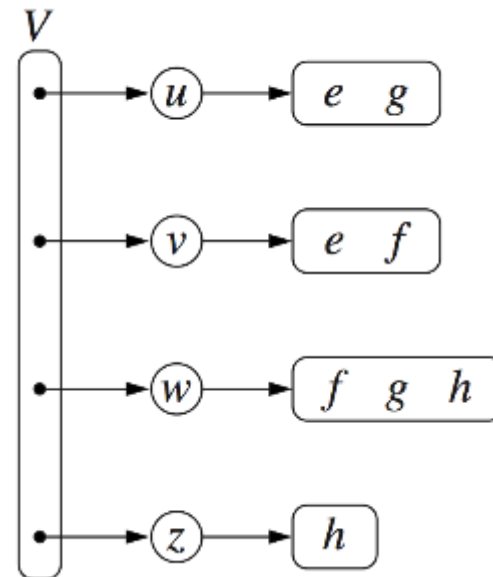
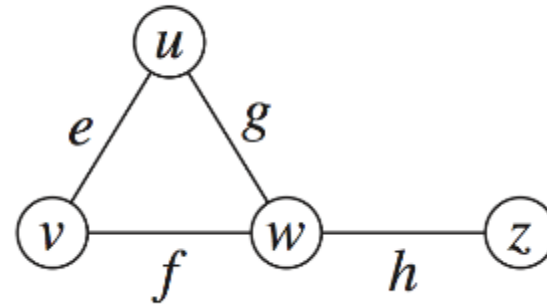
Edge List Structure

- ◆ Vertex object
 - element
 - reference to position in vertex sequence
- ◆ Edge object
 - element
 - origin vertex object
 - destination vertex object
 - reference to position in edge sequence
- ◆ Vertex sequence
 - sequence of vertex objects
- ◆ Edge sequence
 - sequence of edge objects



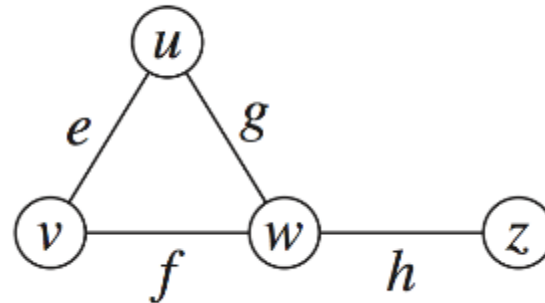
Adjacency List Structure

- ◆ Incidence sequence for each vertex
 - sequence of references to edge objects of incident edges
- ◆ Augmented edge objects
 - references to associated positions in incidence sequences of end vertices



Adjacency Matrix Structure

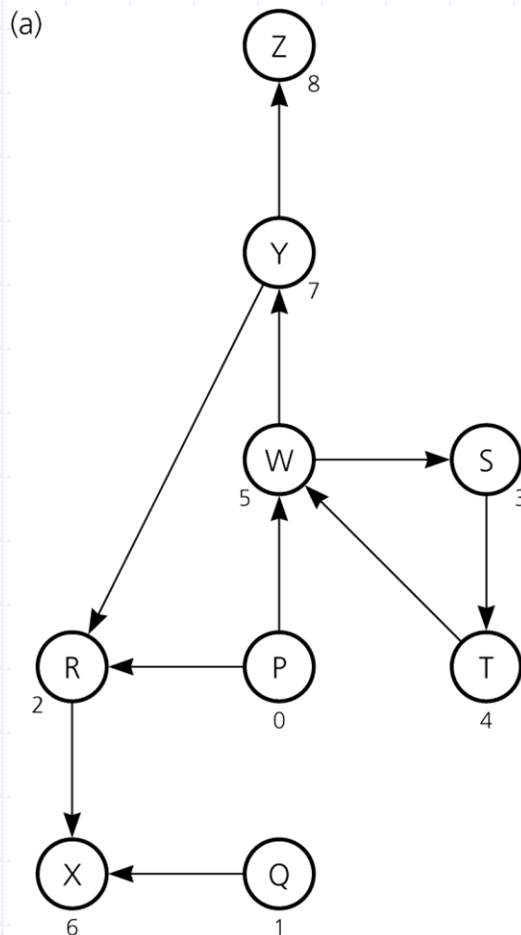
- ◆ Edge list structure
- ◆ Augmented vertex objects
 - Integer key (index) associated with vertex
- ◆ 2D-array adjacency array
 - Reference to edge object for adjacent vertices
 - Null for non adjacent vertices
- ◆ The “old fashioned” version just has 0 for no edge and 1 for edge



		0	1	2	3	
u	\longrightarrow	0		e	g	
v	\longrightarrow	1	e		f	
w	\longrightarrow	2	g	f		h
z	\longrightarrow	3			h	

Adjacency Matrix – Example1

A directed graph



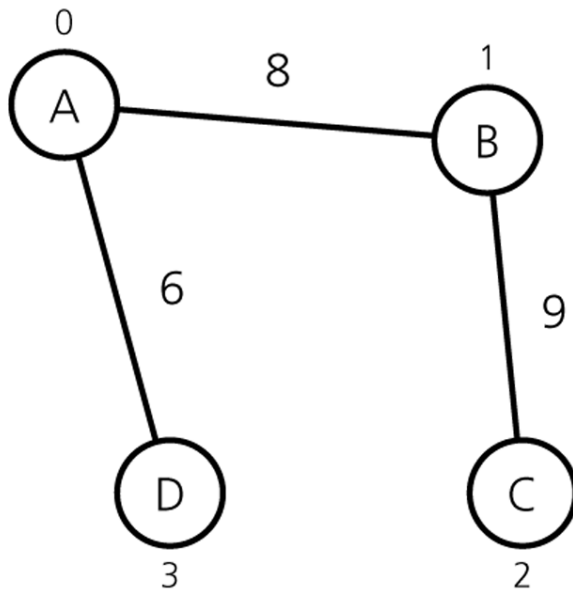
Its adjacency matrix

(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Adjacency Matrix – Example2

An Undirected Weighted Graph

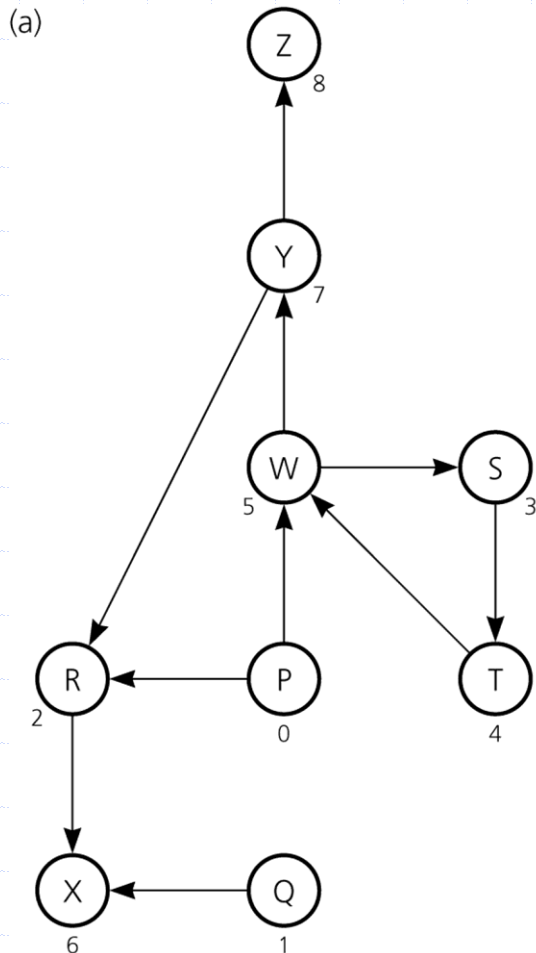


Its Adjacency Matrix

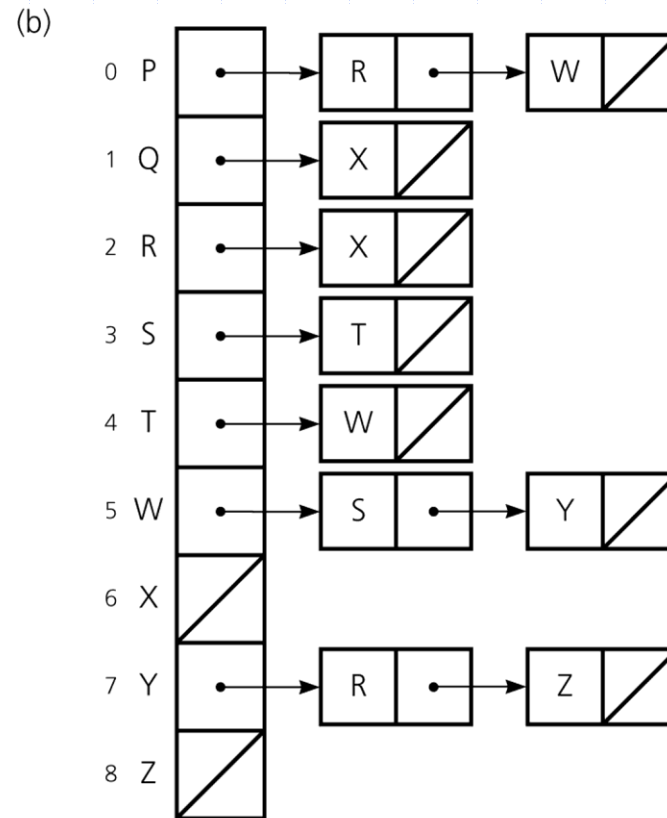
		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Adjacency List – Example1

A directed graph

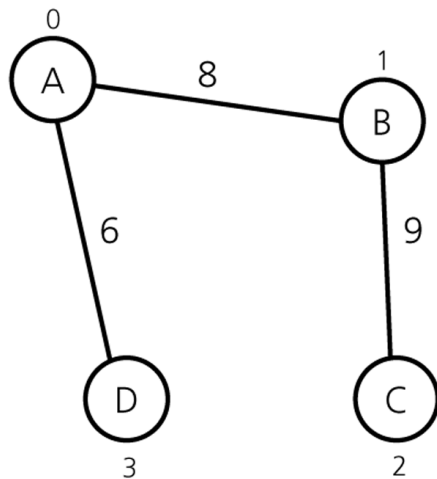


Its Adjacency List

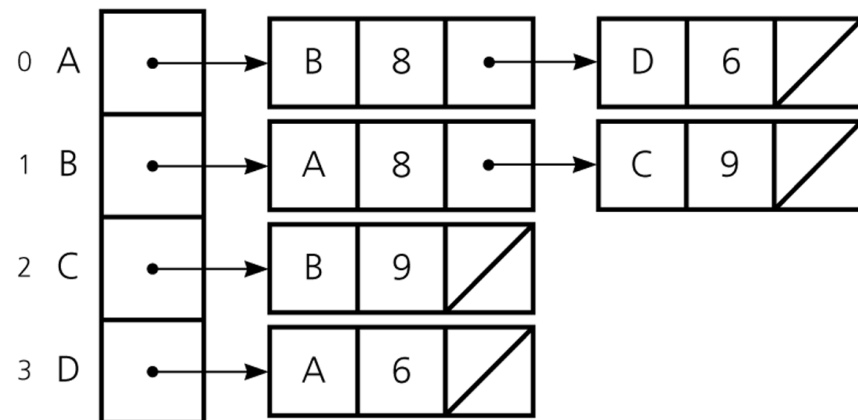


Adjacency List – Example2

An Undirected Weighted Graph



Its Adjacency List



Adjacency Matrix vs Adjacency List

- ◆ Two common graph operations:
 1. Determine whether there is an edge from vertex i to vertex j .
 2. Find all vertices adjacent to a given vertex i .
- ◆ An adjacency matrix supports operation 1 more efficiently.
- ◆ An adjacency list supports operation 2 more efficiently.
- ◆ An adjacency list often requires less space than an adjacency matrix.

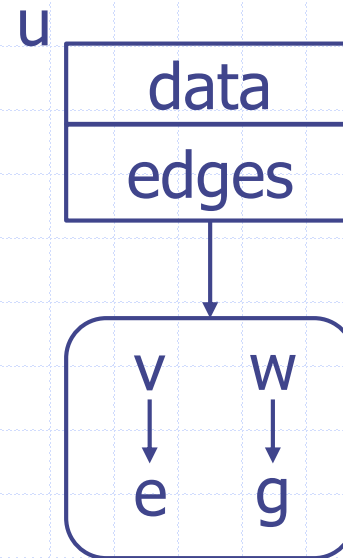
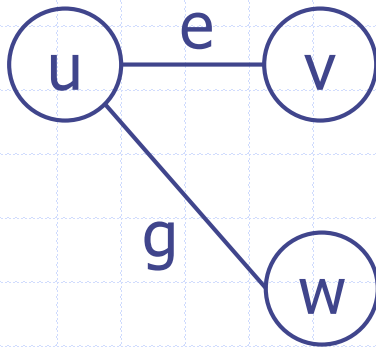
Let's Define a Graph ADT

- `boolean isEmpty()`
- `int numVertices()`
- `int numEdges()`
- `Iterable<Vertex> vertices()`
- `Iterable<Edge> edges()`
- `Edge getEdge(Vertex u, Vertex v)`
- `Iterable<Vertex> getVertices(Edge e)`
- `int degree(Vertex v)` (inDegree and outDegree if directed)
- `Iterable<Edge> incidentEdges(Vertex v)` (incomingEdges and outgoingEdges if directed)
- `boolean areAdjacent(Vertex u, Vertex v)` (directed versions possible)
- `Data getData(Vertex v)` data: data stored in the vertex
- `double getWeight(Edge e)` if weighted
- `Iterable<Vertex> getNeighbors(Vertex v)`
- `Vertex insertVertex(Data x)`
- `Edge insertEdge(Vertex u, Vertex v, double w)` w: edge weight
- `Vertex removeVertex(Data x)`
- `Edge insertEdge(Vertex u, Vertex v, double w)` w: edge weight
- `Edge removeEdge(Data x)`

Iterables can be
any container

Adjacency Map

- ◆ Adjacency list implementation with HashMaps as the edge container
- ◆ Within a given vertex's edge container, the edges are "indexed" by the other vertex



Brainstorm: Implementing the Graph ADT with Adjacency Maps

Vertex

```
Data data  
HashMap<Vertex, Edge> edgeMap
```

Graph

```
ArrayList<Vertex> vertices  
ArrayList<Edge> edges  
// Skipping some of the methods...  
ArrayList<Edge> incidentEdges(Vertex v)  
boolean areAdjacent(Vertex u, Vertex v)  
ArrayList<Vertex> getNeighbors(Vertex v)  
Vertex insertVertex(Data x)  
Edge insertEdge(Vertex u, Vertex v, double w)  
Vertex removeVertex(Data x)  
Edge insertEdge(Vertex u, Vertex v, double w)  
Edge removeEdge(Data x)
```

Discussion on the board

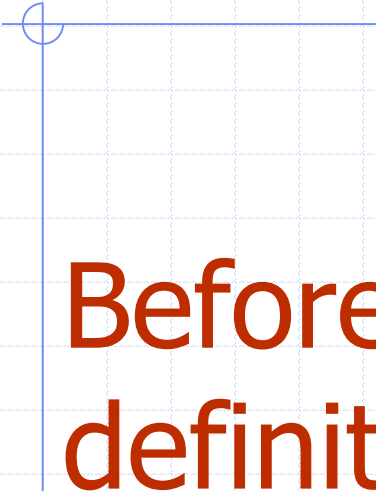
Performance

<ul style="list-style-type: none"> ▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops 	Edge List	Adjacency List/Map	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w)) / 1$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

Warning: These performances can be different based on the underlying containers and edge directionality!

Using Graphs in Practise

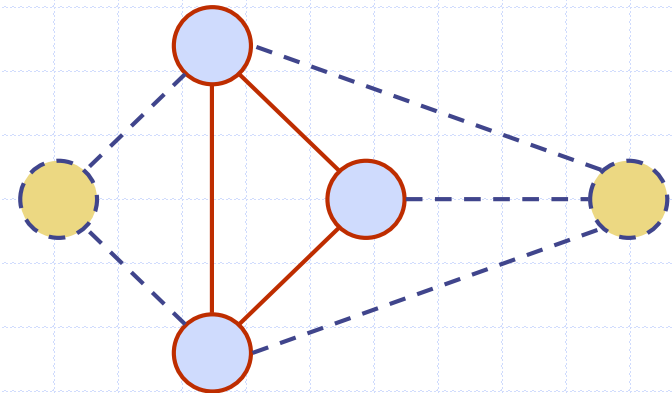
- ◆ Not too sure? Pick a 3rd party library! I suggest <http://jgrapht.org>
- ◆ Cannot use 3rd party but need to be generic? Implement adjacency map (adjacency lists with HashMaps as the edge containers)
- ◆ High Performance or niche application? Define your requirements
 - Do you need undirected or directed graphs?
 - Do you need weighted edges or not?
 - Do you need to insert/remove edges and vertices frequently?
 - Can you say anything about the number of edges with respect to number of vertices? Which implementation do you want to chose?
 - ...
- ◆ You do not always have to use an explicit graph data structure!
 - E.g. game playing, you do not store your neighbors explicitly but generate them on the fly using available moves



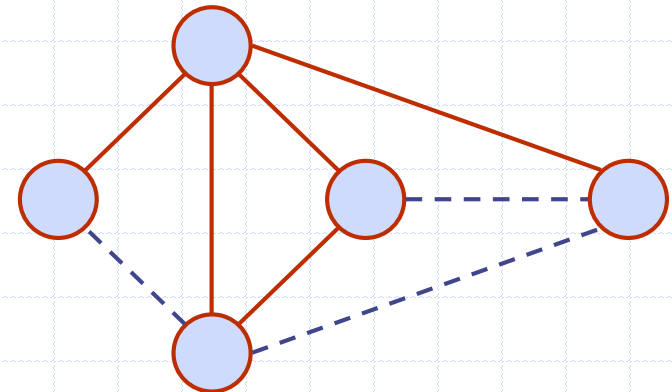
Before we move on, some more definitions...

Subgraphs

- ◆ A subgraph S of a graph G is a graph such that
 - The vertices of S are a subset of the vertices of G
 - The edges of S are a subset of the edges of G
- ◆ A spanning subgraph of G is a subgraph that contains all the vertices of G



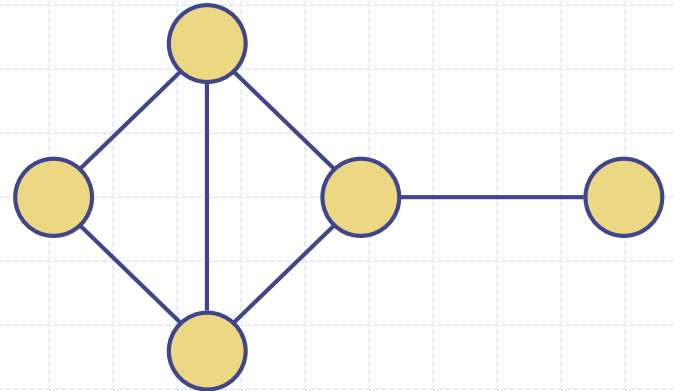
Subgraph



Spanning subgraph

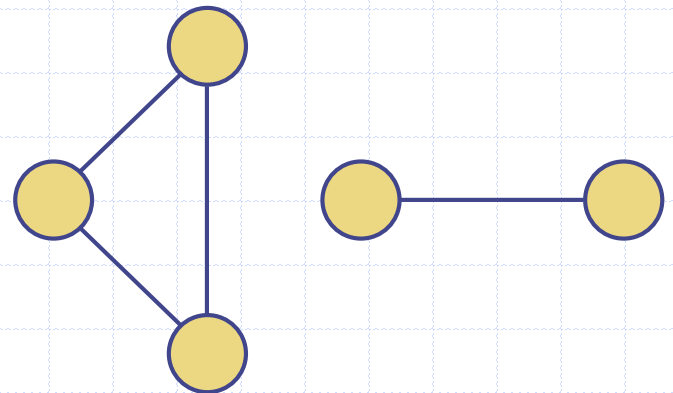
Connectivity

◆ A graph is connected if there is a path between every pair of vertices



Connected graph

◆ A connected component of a graph G is a maximal connected subgraph of G



Non connected graph with two connected components

Trees and Forests

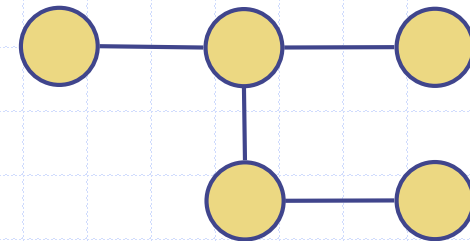
◆ A (free) tree is an undirected graph T such that

- T is connected
- T has no cycles

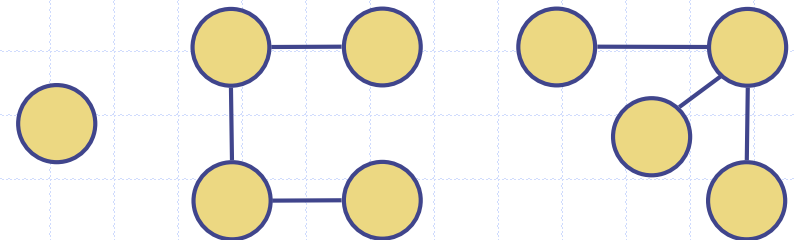
This definition of tree is different from the one of a rooted tree

◆ A forest is an undirected graph without cycles

◆ The connected components of a forest are trees



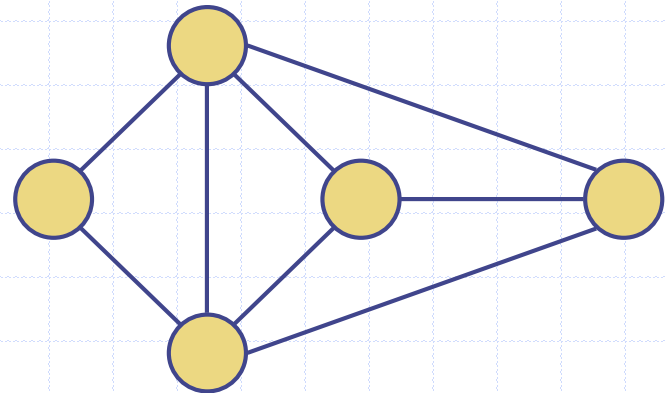
Tree



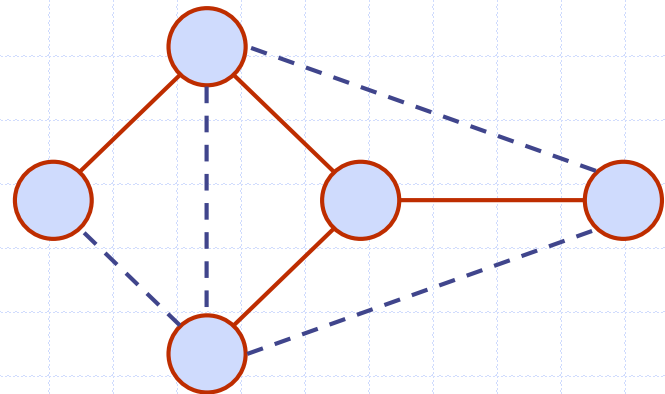
Forest

Spanning Trees and Forests

- ◆ A spanning tree of a connected graph is a spanning subgraph that is a tree
- ◆ A spanning tree is not unique unless the graph is a tree
- ◆ Spanning trees have applications to the design of communication networks
- ◆ A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree