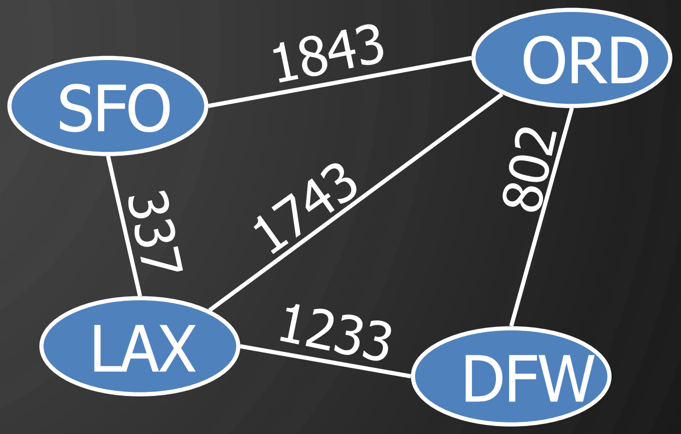
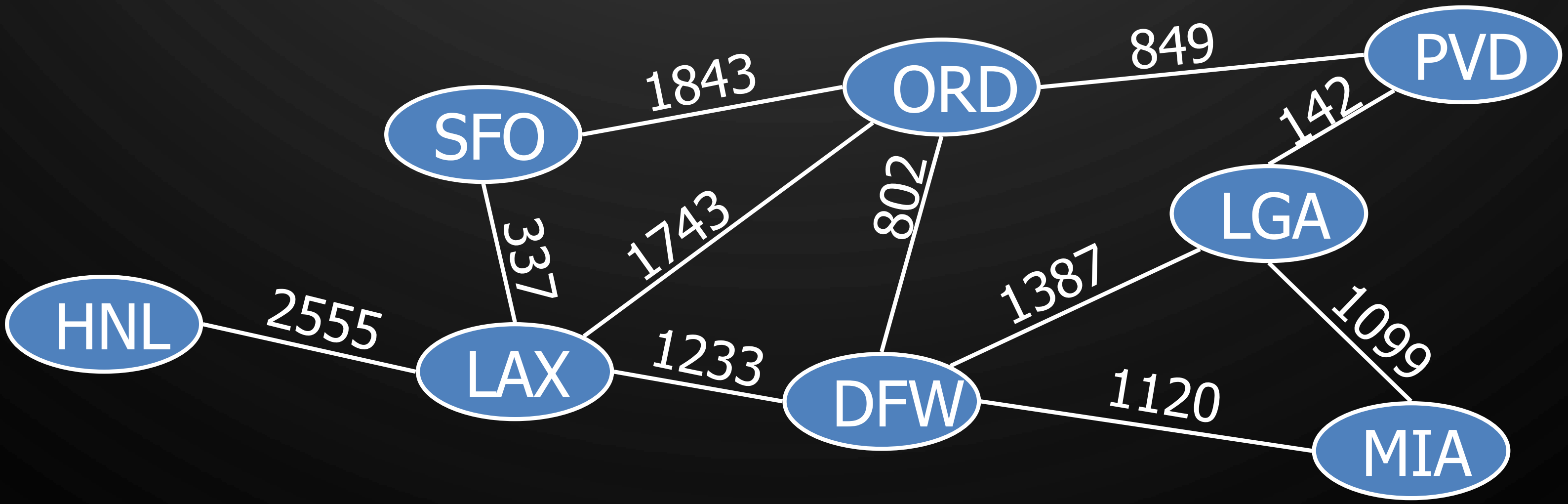
CHAPTER 14
GRAPH ALGORITHMS



ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

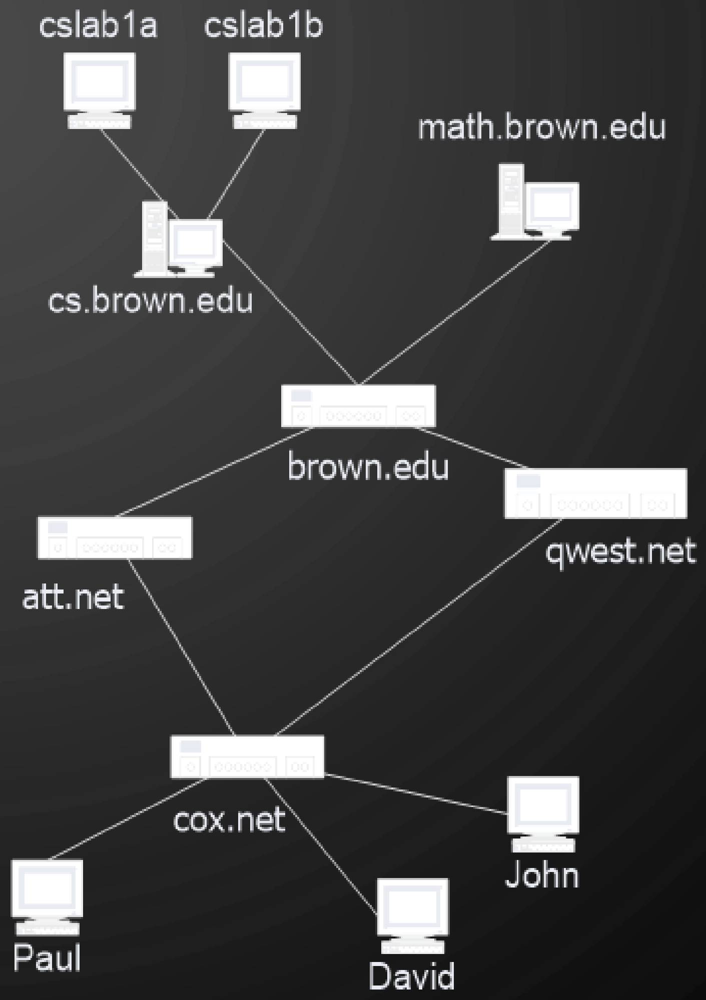
GRAPH

- A **graph** is a pair $G = (V, E)$, where
  - V is a set of nodes, called **vertices**
  - E is a collection of pairs of vertices, called **edges**
  - Vertices and edges can store arbitrary elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



APPLICATIONS

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



TERMINOLOGY
EDGE AND GRAPH TYPES

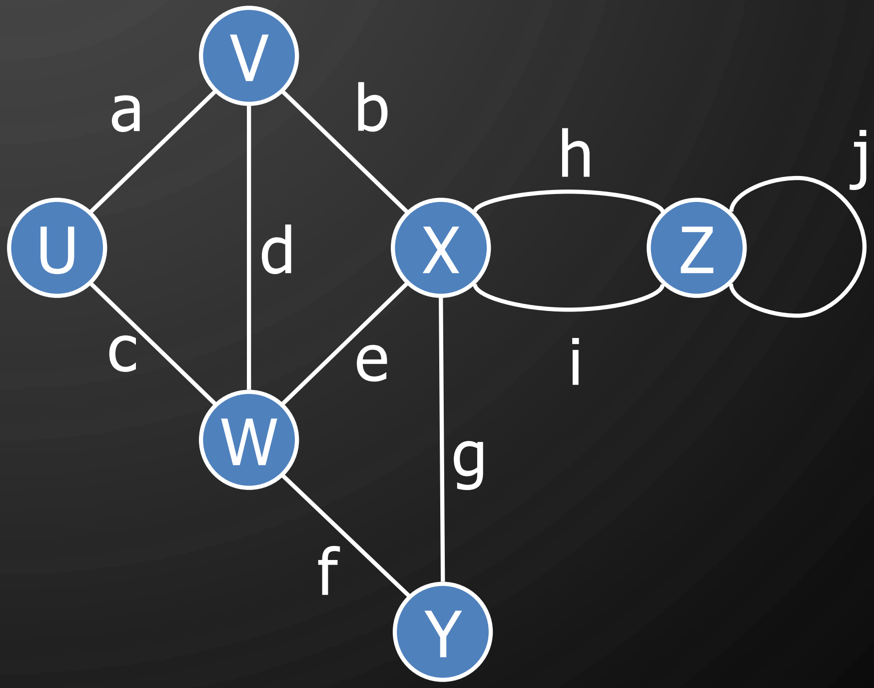
- Edge Types
  - **Directed** edge
    - ordered pair of vertices (u, v)
    - first vertex u is the origin/source
    - second vertex v is the destination/target
    - e.g., a flight
  - **Undirected** edge
    - unordered pair of vertices (u, v)
    - e.g., a flight route
  - **Weighted** edge
    - Numeric label associated with edge

(u, v)

- Graph Types
  - **Directed** graph (Digraph)
    - all the edges are directed
    - e.g., route network
  - **Undirected** graph
    - all the edges are undirected
    - e.g., flight network
  - **Weighted** graph
    - all the edges are weighted

TERMINOLOGY
VERTICES AND EDGES

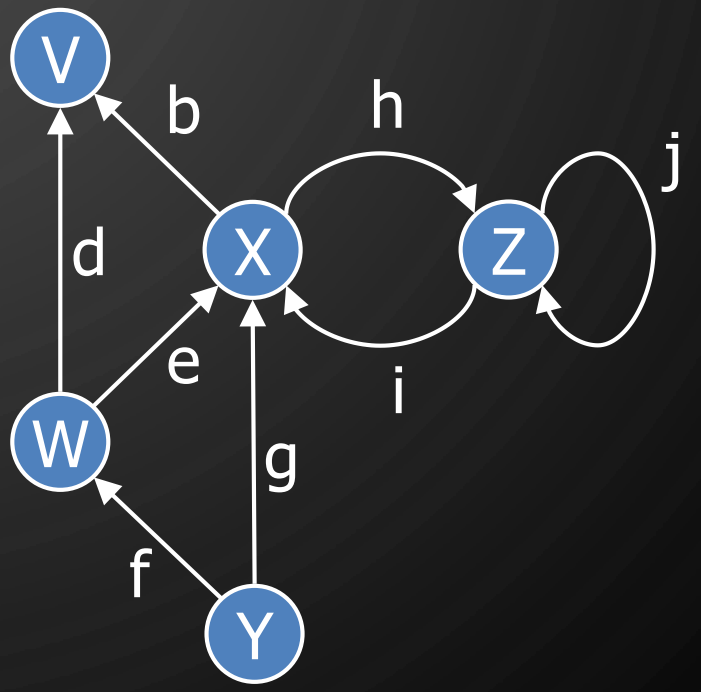
- **End points** (or end vertices) of an edge
  - U and V are the endpoints of a
- Edges **incident** on a vertex
  - a, d, and b are incident on V
- **Adjacent** vertices
  - U and V are adjacent
- **Degree** of a vertex
  - X has degree 5
- **Parallel (multiple)** edges
  - h and i are parallel edges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



Note: A graph with no parallel edges or self loops are said to be **simple**. Unless otherwise stated, you should assume all graphs discussed are simple

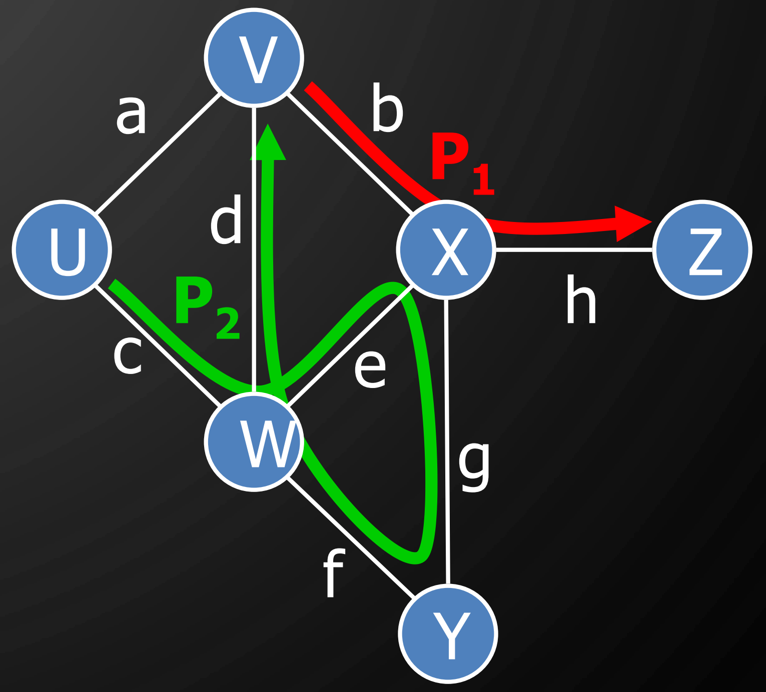
TERMINOLOGY
VERTICES AND EDGES

- **Outgoing** edges of a vertex
  - h and b are the outgoing edges of X
- **Incoming** edges of a vertex
  - e, g, and i are incoming edges of X
- **In-degree** of a vertex
  - X has in-degree 3
- **Out-degree** of a vertex
  - X has out-degree 2



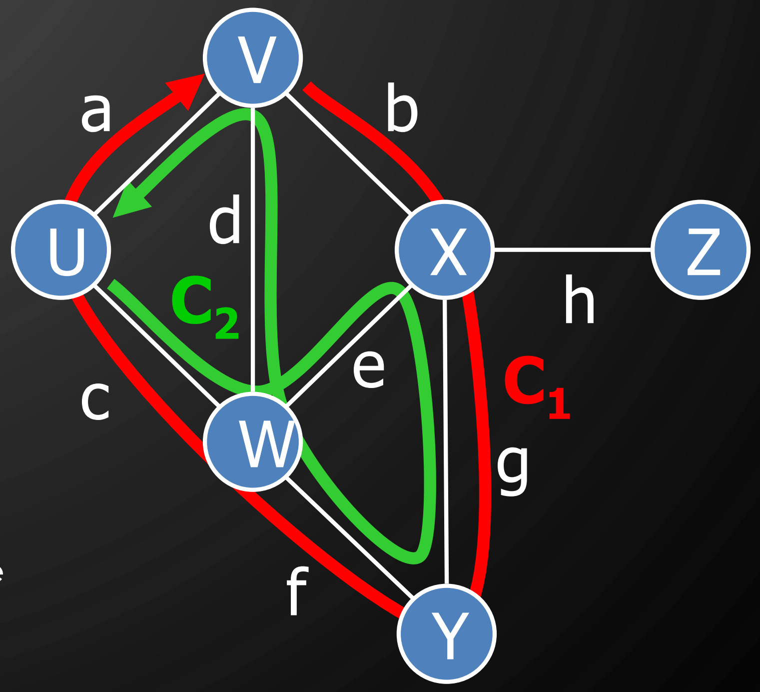
TERMINOLOGY
PATHS

- **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



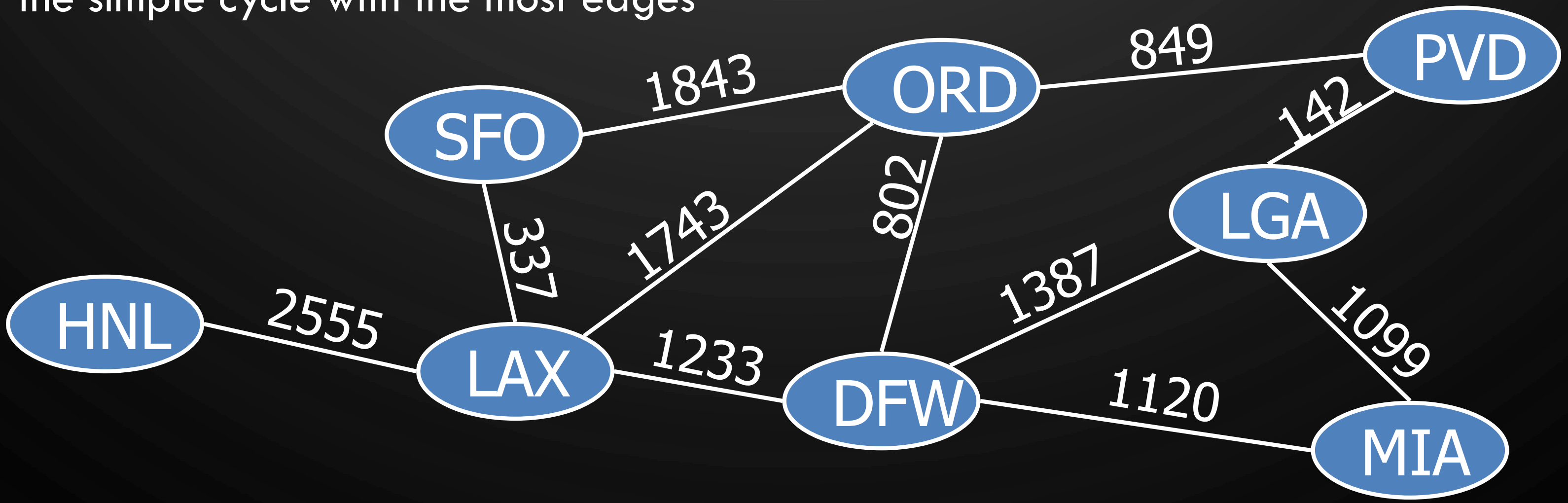
TERMINOLOGY
CYCLES

- **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 except for the beginning and ending vertex
- Examples
  - $C_1 = \{V, b, X, g, Y, f, W, c, U, a, V\}$ is a simple cycle
  - $C_2 = \{U, c, W, e, X, g, Y, f, W, d, V, a, U\}$ is a cycle that is not simple
- A digraph is called **acyclic** if it does not contain any cycles



EXERCISE ON TERMINOLOGY

1. Number of vertices?
2. Number of edges?
3. What type of the graph is it?
4. Show the end vertices of the edge with largest weight
5. Show the vertices of smallest degree and largest degree
6. Show the edges incident to the vertices in the above question
7. Identify the shortest simple path from HNL to PVD
8. Identify the simple cycle with the most edges



EXERCISE
PROPERTIES OF UNDIRECTED GRAPHS

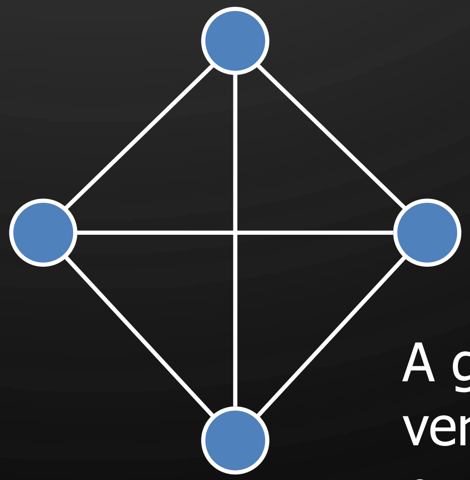
- Property 1 – Total degree

$$\Sigma_v deg(v) = ?$$

- Property 2 – Total number of edges
  - In an undirected graph with no self-loops and no multiple edges

$$m \leq Upper\ Bound?$$
$$Lower\ Bound? \leq m$$

- Notation
  - n number of vertices
  - m number of edges
  - $\deg(v)$ degree of vertex v



Example

- $n = ?$
- $m = ?$
- $\deg(v) = ?$

A graph with given number of vertices (4) and maximum number of edges

EXERCISE
PROPERTIES OF UNDIRECTED GRAPHS

- Property 1 – Total degree

$$\Sigma_v deg(v) = 2m$$

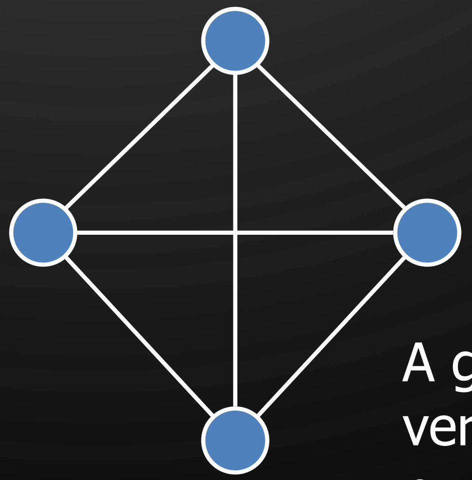
- Property 2 – Total number of edges
  - In an undirected graph with no self-loops and no multiple edges

$$m \leq \frac{n(n-1)}{2}$$

$$0 \leq m$$

Proof: Each vertex can have degree at most $(n-1)$

- Notation
  - n number of vertices
  - m number of edges
  - $\deg(v)$ degree of vertex v



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

A graph with given number of vertices (4) and maximum number of edges

EXERCISE
PROPERTIES OF DIRECTED GRAPHS

- Property 1 – Total in-degree and out-degree

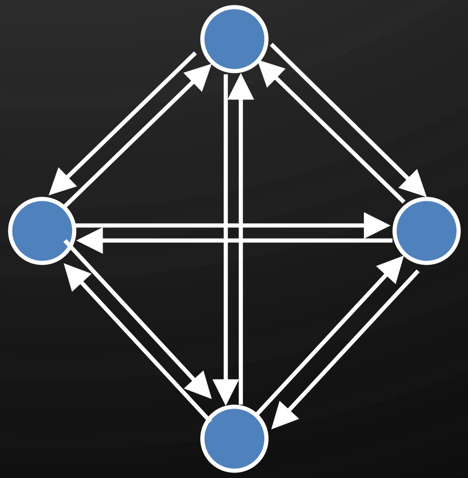
  $$\Sigma_v in - \deg(v) = ?$$
  $$\Sigma_v out - \deg(v) = ?$$

- Property 2 – Total number of edges
  - In an directed graph with no self-loops and no multiple edges

    $m \leq UpperBound?$

    $LowerBound? \leq m$

- Notation
  - n number of vertices
  - m number of edges
  - $\deg(v)$ degree of vertex v



Example

- $n = ?$
- $m = ?$
- $\deg(v) = ?$

A graph with given number of vertices (4) and maximum number of edges

EXERCISE
PROPERTIES OF DIRECTED GRAPHS

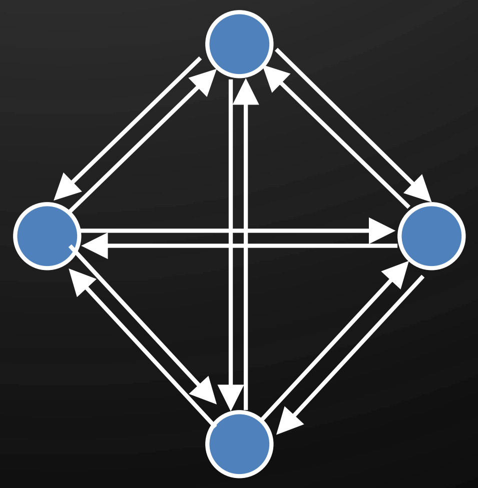
- Property 1 – Total in-degree and out-degree

$$\Sigma_v in - \deg(v) = m$$
$$\Sigma_v out - \deg(v) = m$$

- Property 2 – Total number of edges
  - In an directed graph with no self-loops and no multiple edges

$$m \leq n(n-1)$$
$$0 \leq m$$

- Notation
  - n number of vertices
  - m number of edges
  - $\deg(v)$ degree of vertex v



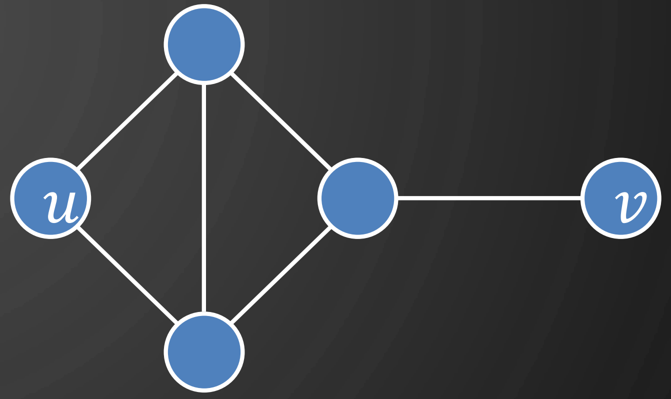
Example

- $n = 4$
- $m = 12$
- $\deg(v) = 6$

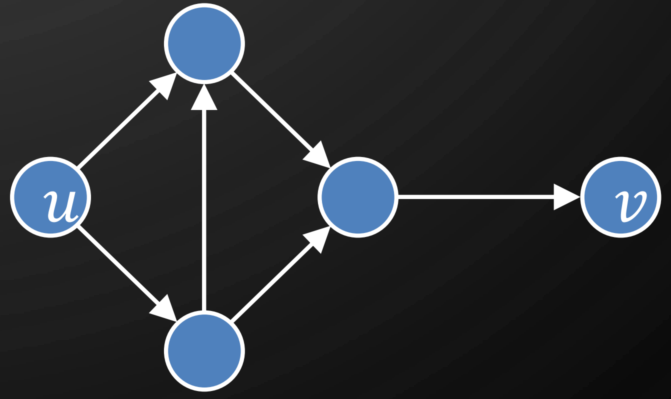
A graph with given number of vertices (4) and maximum number of edges

TERMINOLOGY
CONNECTIVITY

- Given two vertices u and v, we say u **reaches** v, and that v is **reachable** from u, if there exists a path from u to v. In an undirected graph **reachability** is symmetric
- A graph is **connected** if there is a path between every pair of vertices
- A digraph is **strongly connected** if there every pair of vertices is reachable



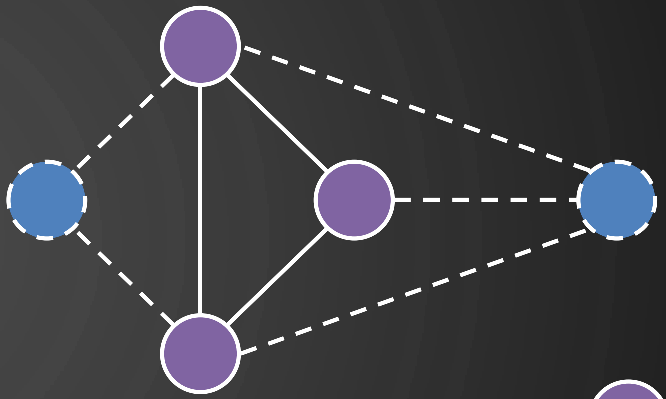
Connected graph
u and v are reachable



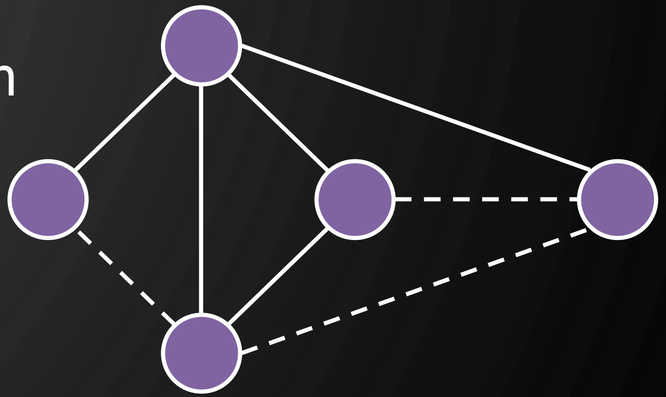
Connected digraph
u and v are not mutually reachable

TERMINOLOGY
SUBGRAPHS

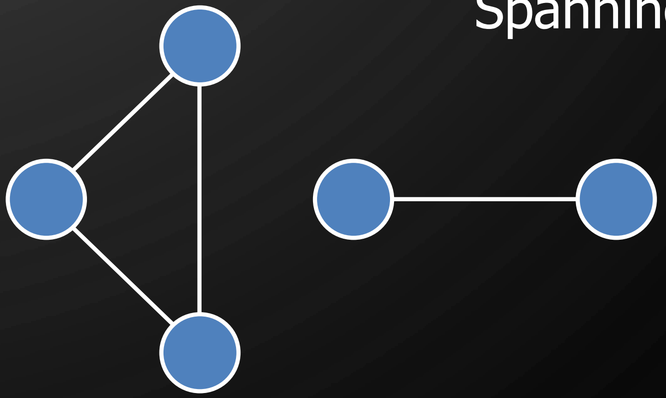
- A **subgraph** H of a graph G is a graph whose vertices and edges are subsets of G, respectively
- A **spanning subgraph** of G is a subgraph that contains all the vertices of G
- A **connected component** of a graph G is a maximal connected subgraph of G



Subgraph



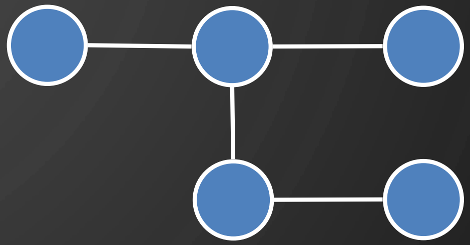
Spanning subgraph



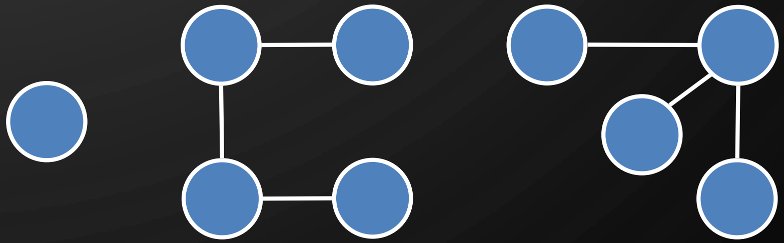
Non connected graph with two connected components

TERMINOLOGY
TREES AND FORESTS

- A **forest** is a graph without cycles
- A **(free) tree** is connected forest
  - This definition of tree is different from the one of a rooted tree
- The connected components of a forest are trees



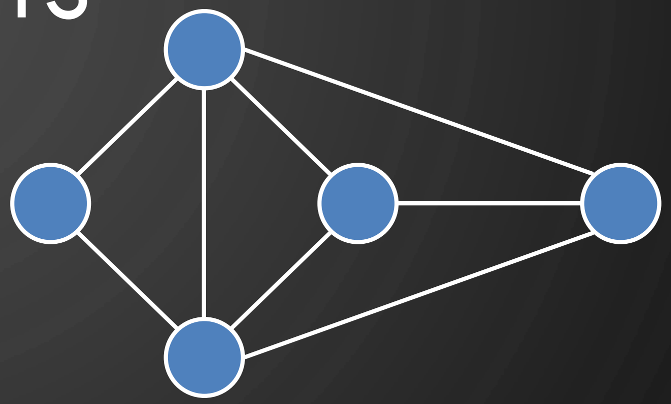
Tree



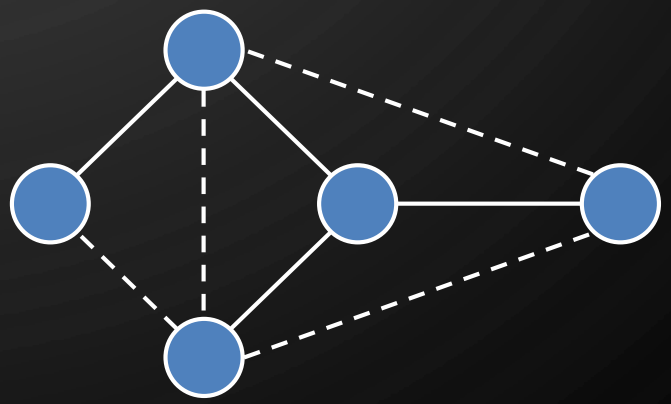
Forest

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



Graph



Spanning tree

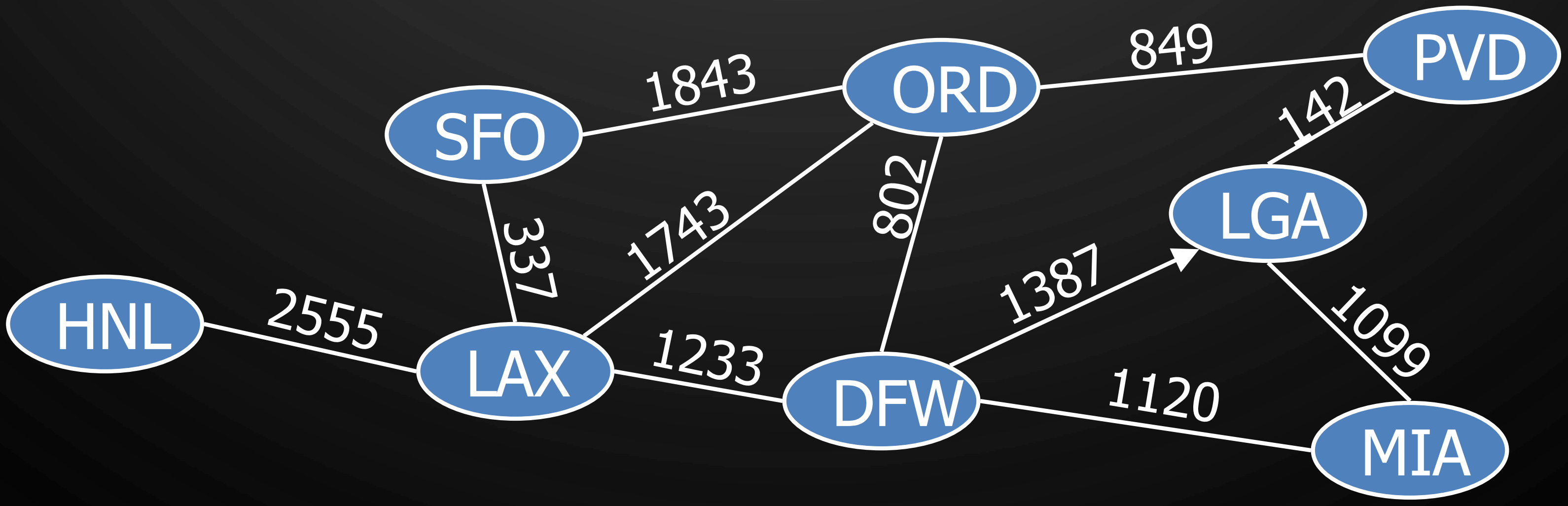
GRAPH ADT

- Vertices and edges are lightweight objects and store elements
- Although the ADT is specified from the graph object, we often have similar functions in the Vertex and Edge objects

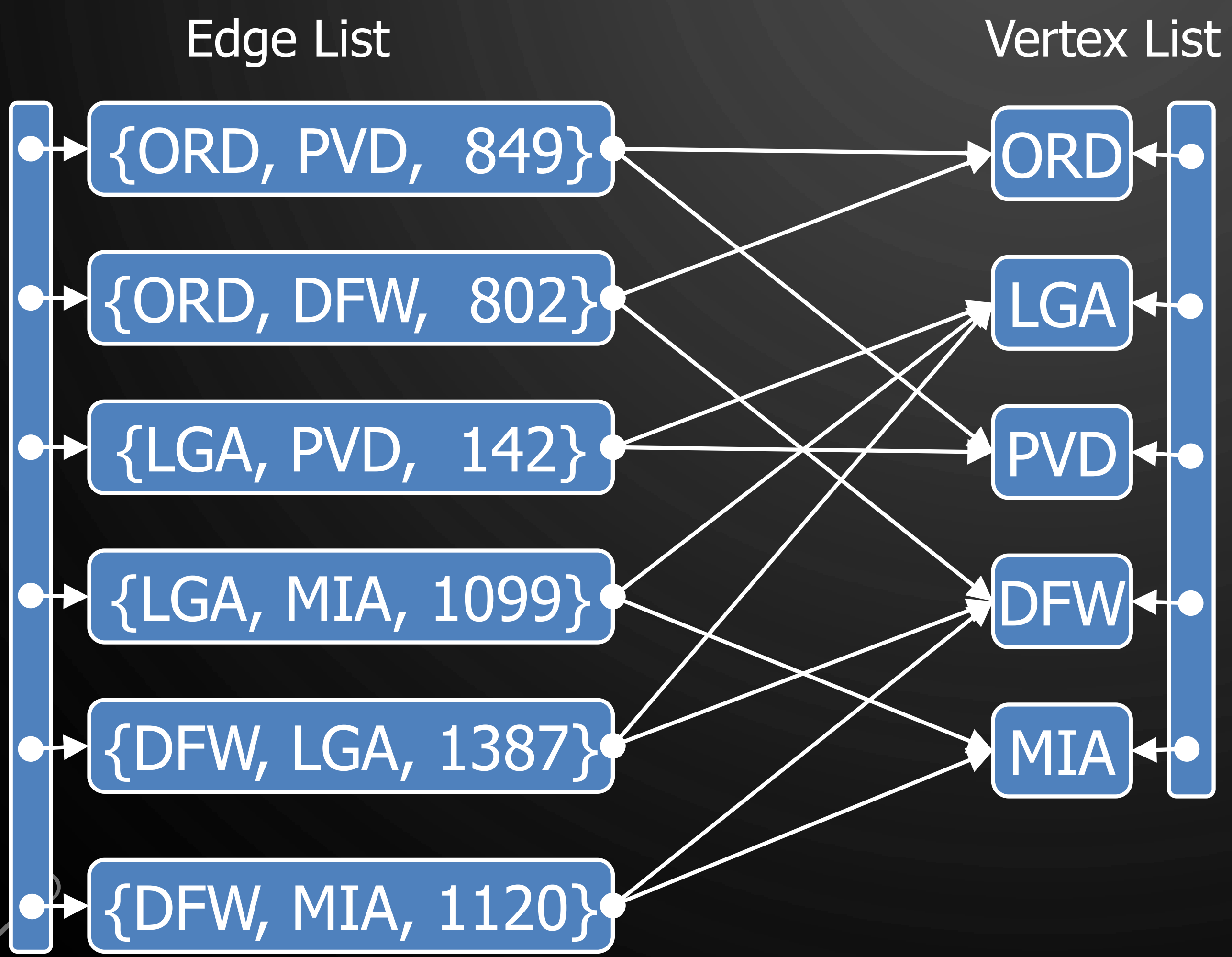
numVertices():	Returns the number of vertices of the graph.
vertices():	Returns an iteration of all the vertices of the graph.
numEdges():	Returns the number of edges of the graph.
edges():	Returns an iteration of all the edges of the graph.
getEdge(u, v):	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e):	Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
opposite(v, e):	For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
outDegree(v):	Returns the number of outgoing edges from vertex v.
inDegree(v):	Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v):	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v):	Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x):	Creates and returns a new Vertex storing element x.
insertEdge(u, v, x):	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v):	Removes vertex v and all its incident edges from the graph.
removeEdge(e):	Removes edge e from the graph.

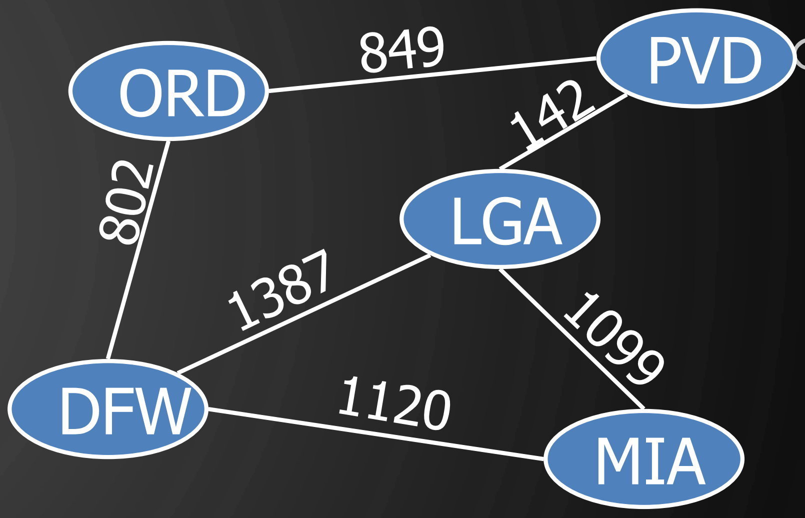
EXERCISE ON ADT

1. `outgoingEdges`(ord)
2. `incomingEdges`(ord)
3. `outDegree`(ord)
4. `endVertices`$(\{lga, mia\})$
5. `opposite`$(dfw, \{dfw, lga\})$
6. `insertVertex`(iah)
7. `insertEdge`$(mia,\ pvd,\ 1200)$
8. `removeVertex`(ord)
9. `removeEdge`$(\{dfw, ord\})$



EDGE LIST STRUCTURE

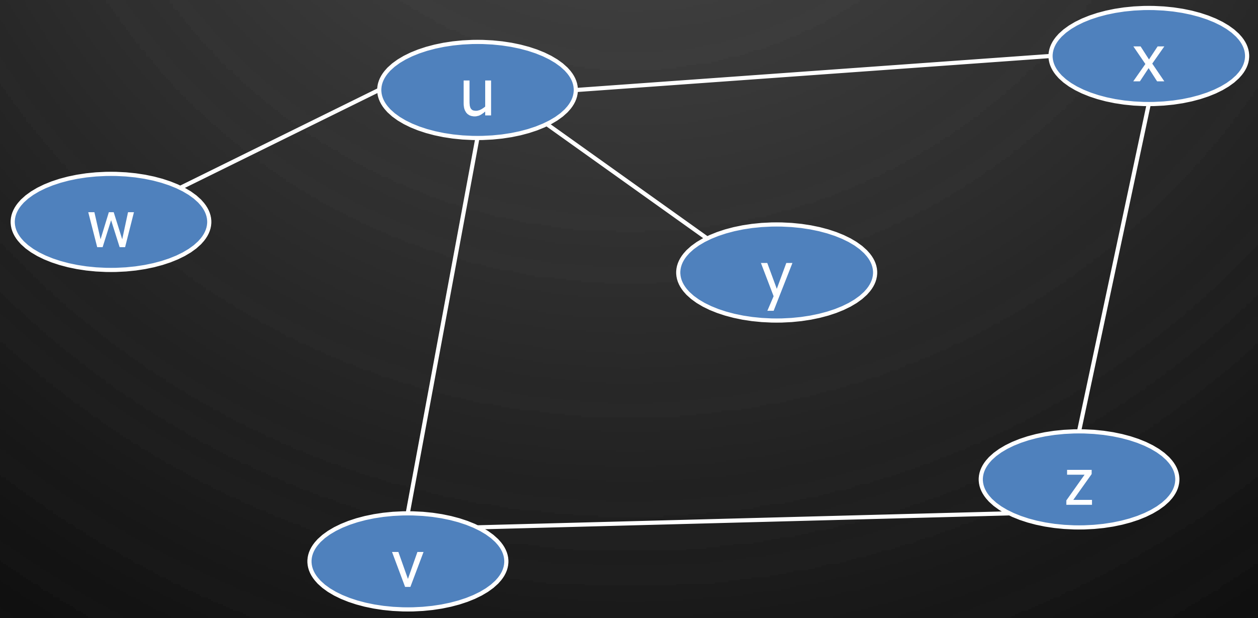




- An **edge list** can be stored in a list or a map/dictionary (e.g. hash table)
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence

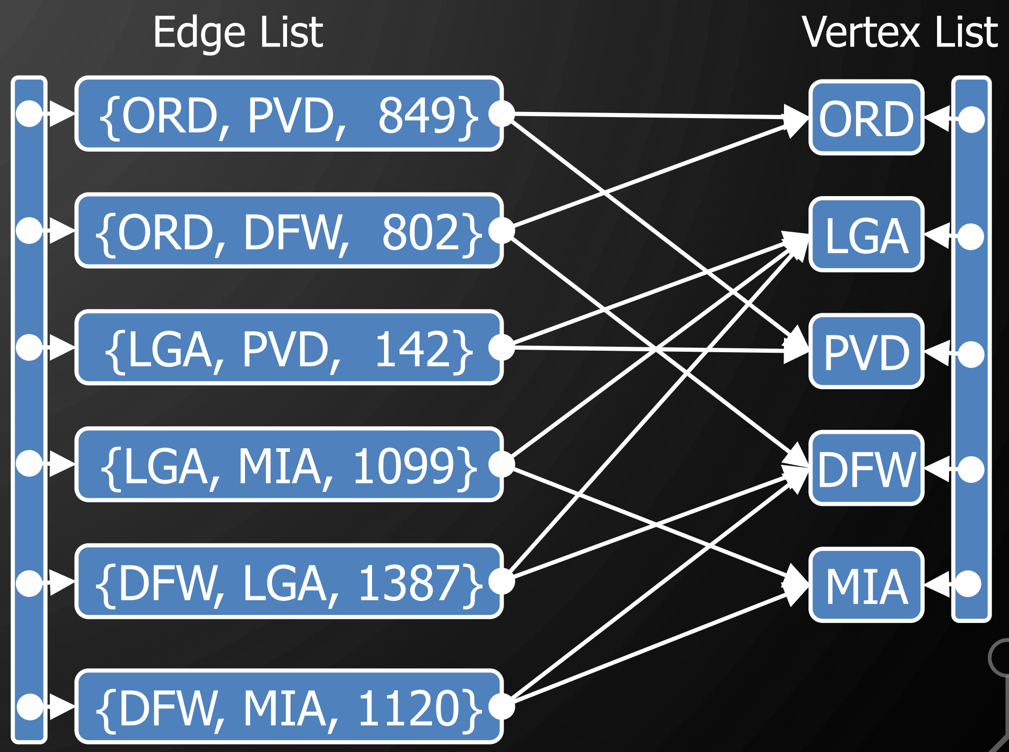
EXERCISE
EDGE LIST STRUCTURE

- Construct the edge list for the following graph



ASYMPTOTIC PERFORMANCE
EDGE LIST STRUCTURE

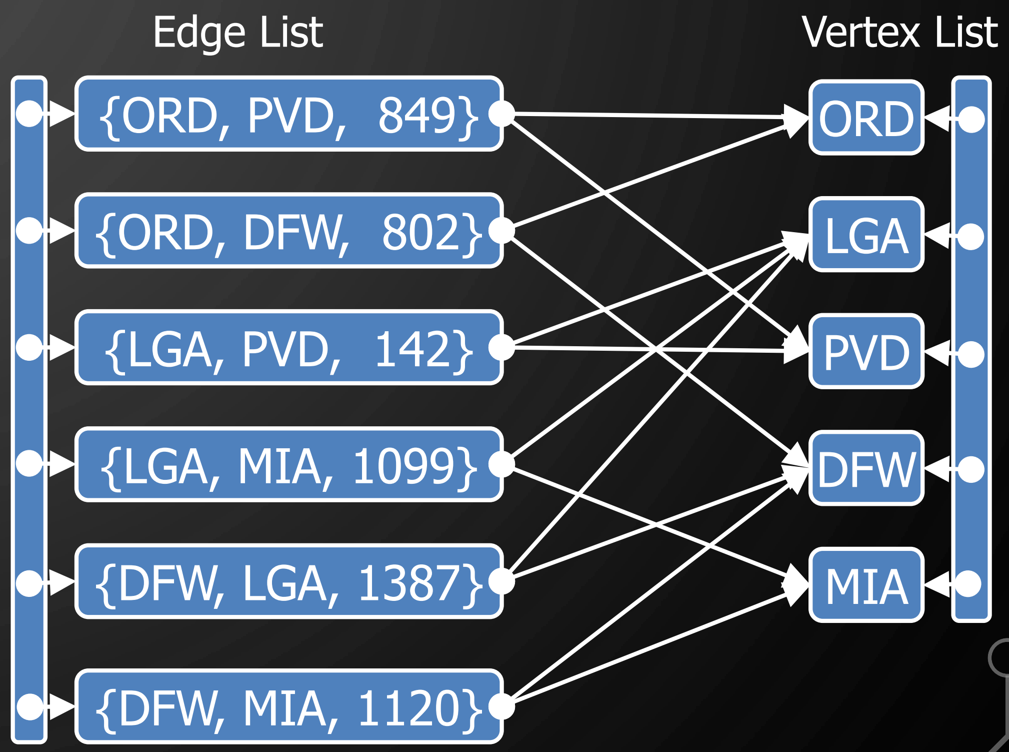
• n vertices, m edges • No parallel edges • No self-loops	Edge List
Space	?
`getEdge`(u,v), `outDegree`(v), `outgoingEdges`(v), `insertEdge`(u,v,w), `removeVertex`(v)	?
`insertVertex`(x), `removeEdge`(e)	?



ASYMPTOTIC PERFORMANCE

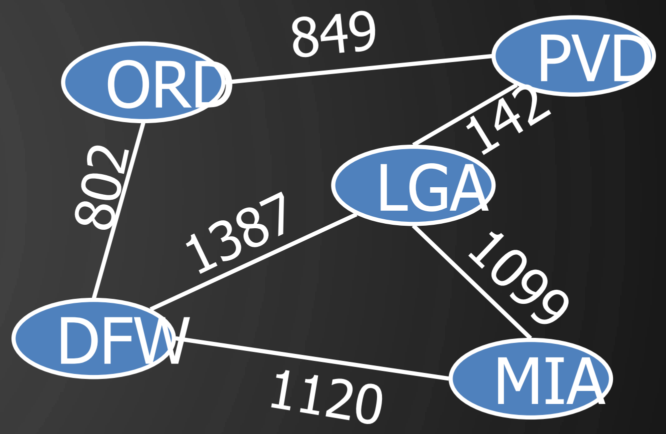
EDGE LIST STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Edge List
Space	$O(n+m)$
`getEdge`(u,v), `outDegree`(v), `outgoingEdges`(v), `insertEdge`(u,v,w), `removeVertex`(v)	$O(m)$
`insertVertex`(x), `removeEdge`(e)	$O(1)$



ADJACENCY LIST STRUCTURE

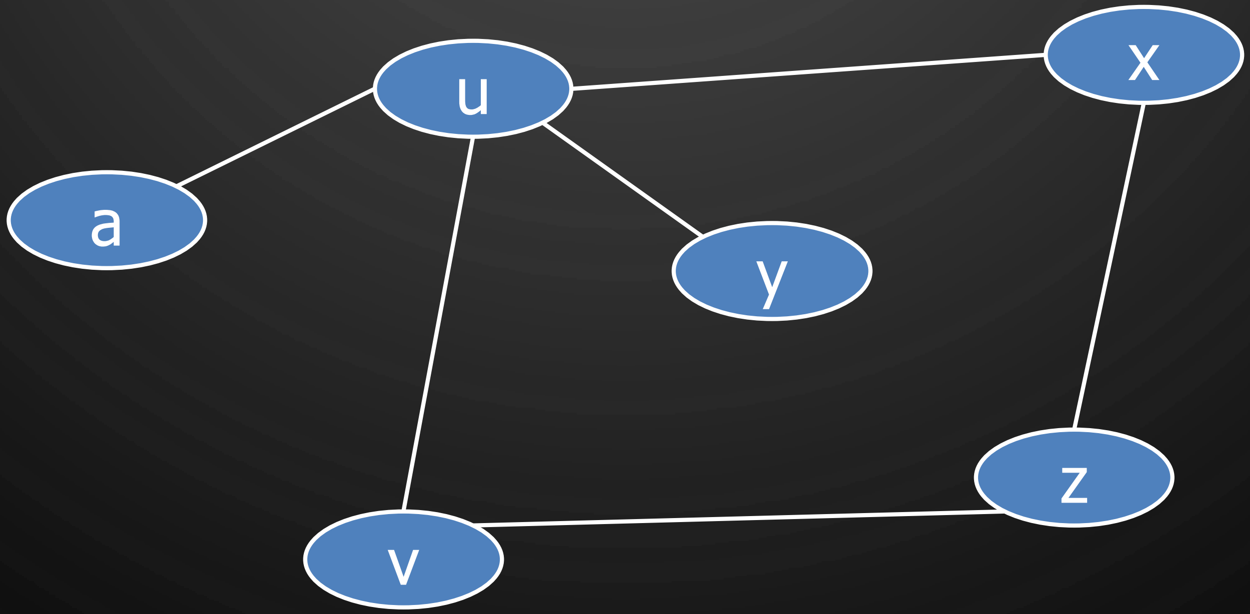
Adjacency List



- **Adjacency Lists** associate vertices with their edges (in addition to edge list!)
- Each vertex stores a list of incident edges
  - List of references to incident edge objects
- Augmented edge object
  - Stores references to associated positions in incident adjacency lists

EXERCISE
ADJACENCY LIST STRUCTURE

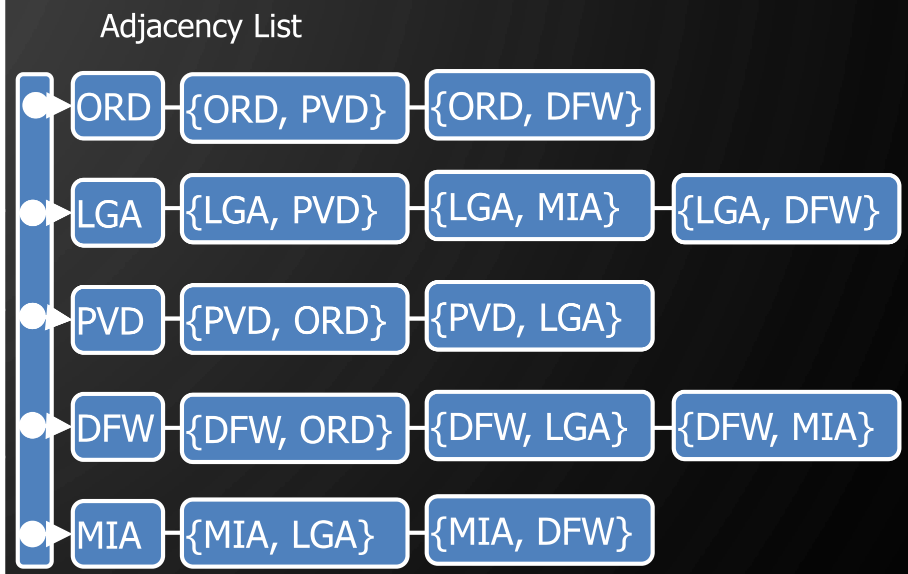
- Construct the adjacency list for the following graph



ASYMPTOTIC PERFORMANCE

ADJACENCY LIST STRUCTURE

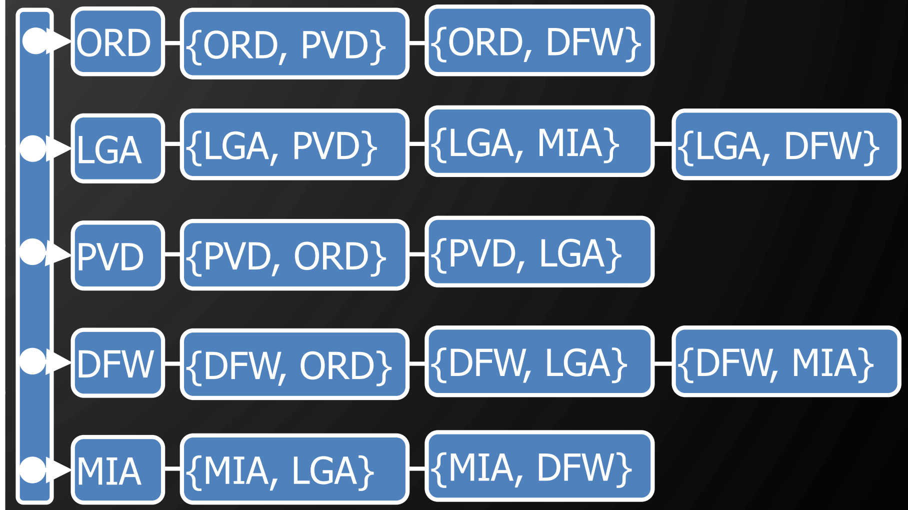
• n vertices, m edges • No parallel edges • No self-loops	Adjacency List
Space	?
`getEdge`(u,v), `insertEdge`(u,v,w)	?
`outDegree`(v), `insertVertex`(x), `removeEdge`(e)	?
`outgoingEdges`(v), `removeVertex`(v)	?



ASYMPTOTIC PERFORMANCE
ADJACENCY LIST STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Adjacency List
Space	$O(n + m)$
`getEdge`(u, v), `insertEdge`(u, v, w)	$O(\min(\deg(v), \deg(u)))$
`outDegree`(v), `insertVertex`(x), `removeEdge`(e)	$O(1)$
`outgoingEdges`(v), `removeVertex`(v)	$O(\deg(v))$

Adjacency List

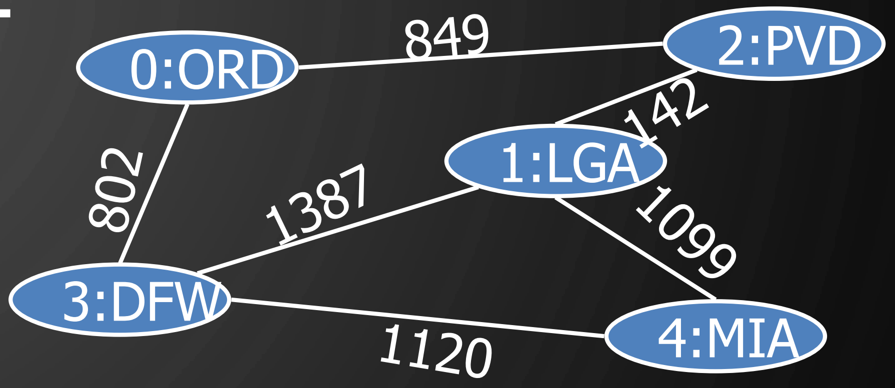


ADJACENCY MAP STRUCTURE

- We can store augmenting incidence structures in maps, instead of lists. This is called an **adjacency map**.
  - In general an "adjacency list" means storing adjacency with vertices, so the terms are interchangeable
- What would this do to the complexities?
  - If it is implemented as a hash table?
  - If it is implemented as a red-black tree?

ADJACENCY MATRIX STRUCTURE

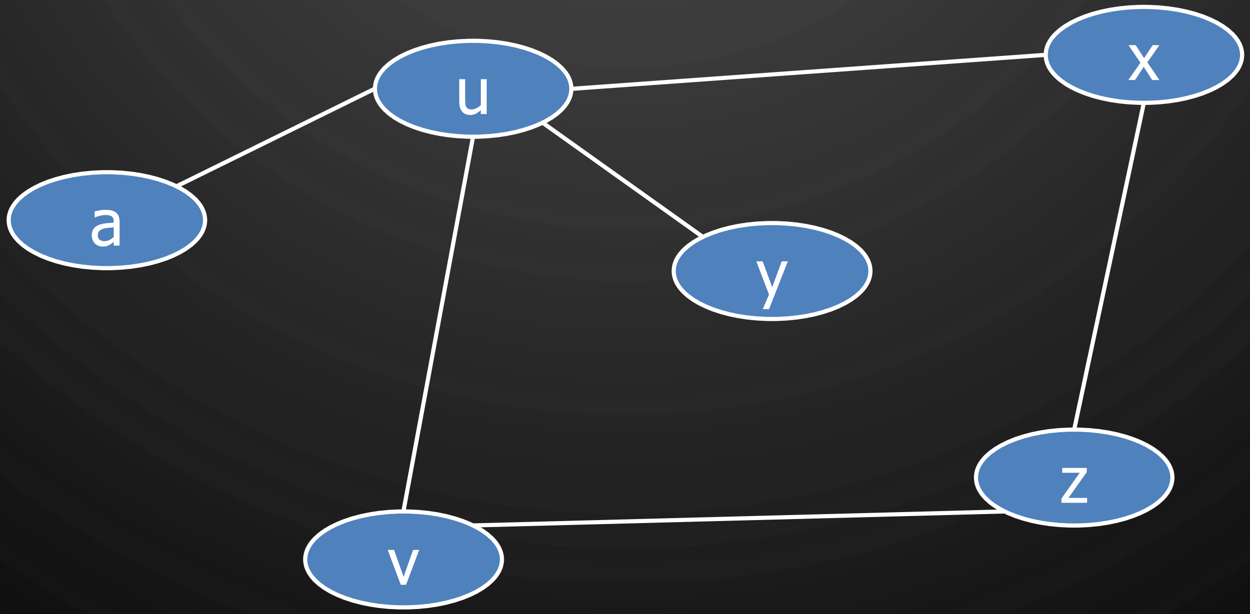
	0	1	2	3	4
0	∅	∅	{0, 2}	{0, 3}	∅
1	∅	∅	{1, 2}	{1, 3}	{1, 4}
2	{0, 2}	{1, 2}	∅	∅	∅
3	{0, 3}	{1, 3}	∅	∅	{3, 4}
4	∅	{1, 4}	∅	{3, 4}	∅



- Adjacency matrices store references to edges in a table (in addition to the edge list)
- Augment vertices with integer keys (often done in all graph implementations!)

EXERCISE
ADJACENCY MATRIX STRUCTURE

- Construct the adjacency matrix for the following graph



ASYMPTOTIC PERFORMANCE
ADJACENCY MATRIX STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Adjacency Matrix
Space	?
`outDegree`(v), `outgoingEdges`(v)	?
`getEdge`(u, v), `insertEdge`(u, v, w), `removeEdge`(e)	?
`insertVertex`(x), `removeVertex`(v)	?

	0	1	2	3	4
0	∅	∅	{0, 2}	{0, 3}	∅
1	∅	∅	{1, 2}	{1, 3}	{1, 4}
2	{0, 2}	{1, 2}	∅	∅	∅
3	{0, 3}	{1, 3}	∅	∅	{3, 4}
4	∅	{1, 4}	∅	{3, 4}	∅

Asymptotic Performance
Adjacency Matrix Structure

• n vertices, m edges • No parallel edges • No self-loops	Adjacency Matrix
Space	$O(n^2)$
`outDegree`(v), `outgoingEdges`(v)	$O(n)$
`getEdge`(u, v), `insertEdge`(u, v, w), `removeEdge`(e)	$O(1)$
`insertVertex`(x), `removeVertex`(v)	$O(n^2)$

	0	1	2	3	4
0	∅	∅	{0, 2}	{0, 3}	∅
1	∅	∅	{1, 2}	{1, 3}	{1, 4}
2	{0, 2}	{1, 2}	∅	∅	∅
3	{0, 3}	{1, 3}	∅	∅	{3, 4}
4	∅	{1, 4}	∅	{3, 4}	∅

ASYMPTOTIC PERFORMANCE

• n vertices, m edges • No parallel edges • No self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n+m)$	$O(n+m)$	$O(n^2)$
`outgoingEdges`(v)	$O(m)$	$O(\deg(v))$	$O(n)$
`getEdge`(u,v)	$O(m)$	$O(\min(\deg(v), \deg(w)))$	$O(1)$
`insertEdge`(u,v,w)	$O(m)$	$O(\min(\deg(v), \deg(w)))$	$O(1)$
`eraseEdge`(e)	$O(1)$	$O(1)$	$O(1)$
`insertVertex`(x)	$O(1)$	$O(1)$	$O(n^2)$
`removeVertex`(v)	$O(m)$	$O(\deg(v))$	$O(n^2)$