SE274 Data Structure

Lecture 9: Graphs

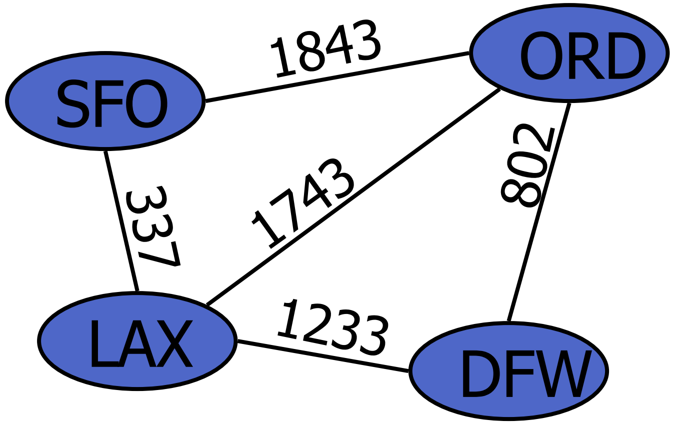
(textbook: Chapter 14)

May *11*, 2020

Instructor: Sunjun Kim

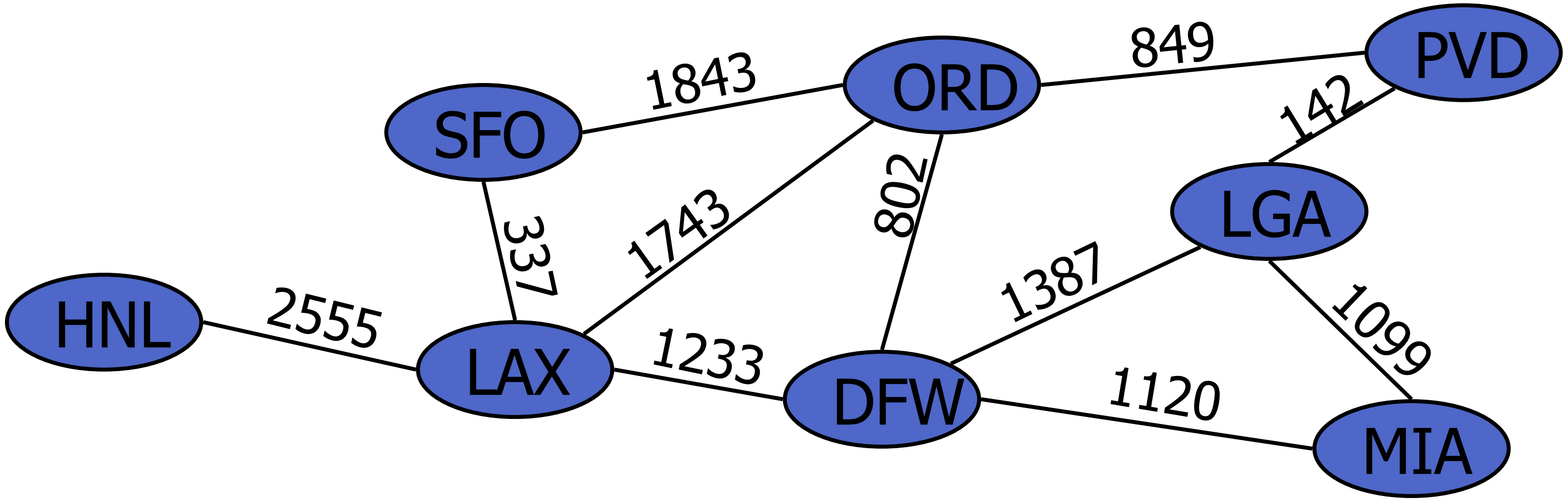
Information&Communication Engineering, DGIST

Graphs



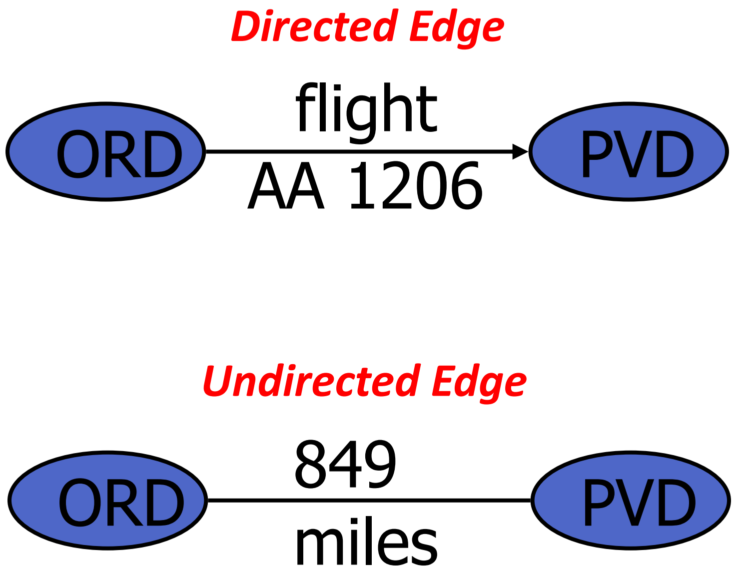
Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store 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



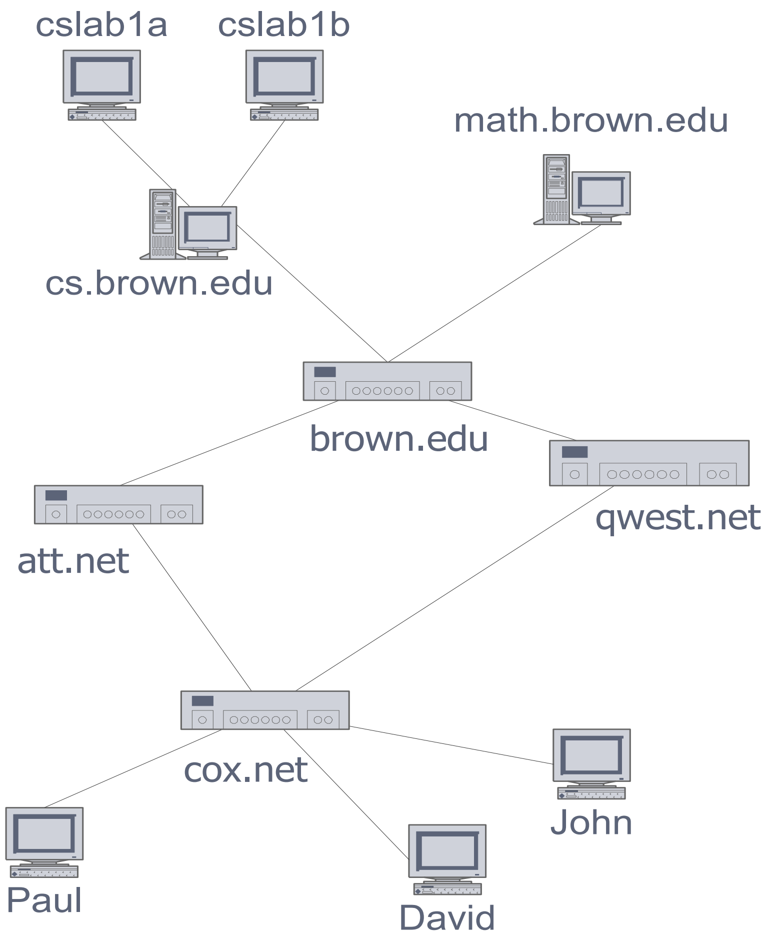
Edge Types

- **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 (=digraph)
  - all the edges are directed
  - e.g., route network, one-way streets, flights, task scheduling
- Undirected graph
  - all the edges are undirected
  - e.g., flight network,
- Mixed graph
  - Some edges are directed, and some are undirected.



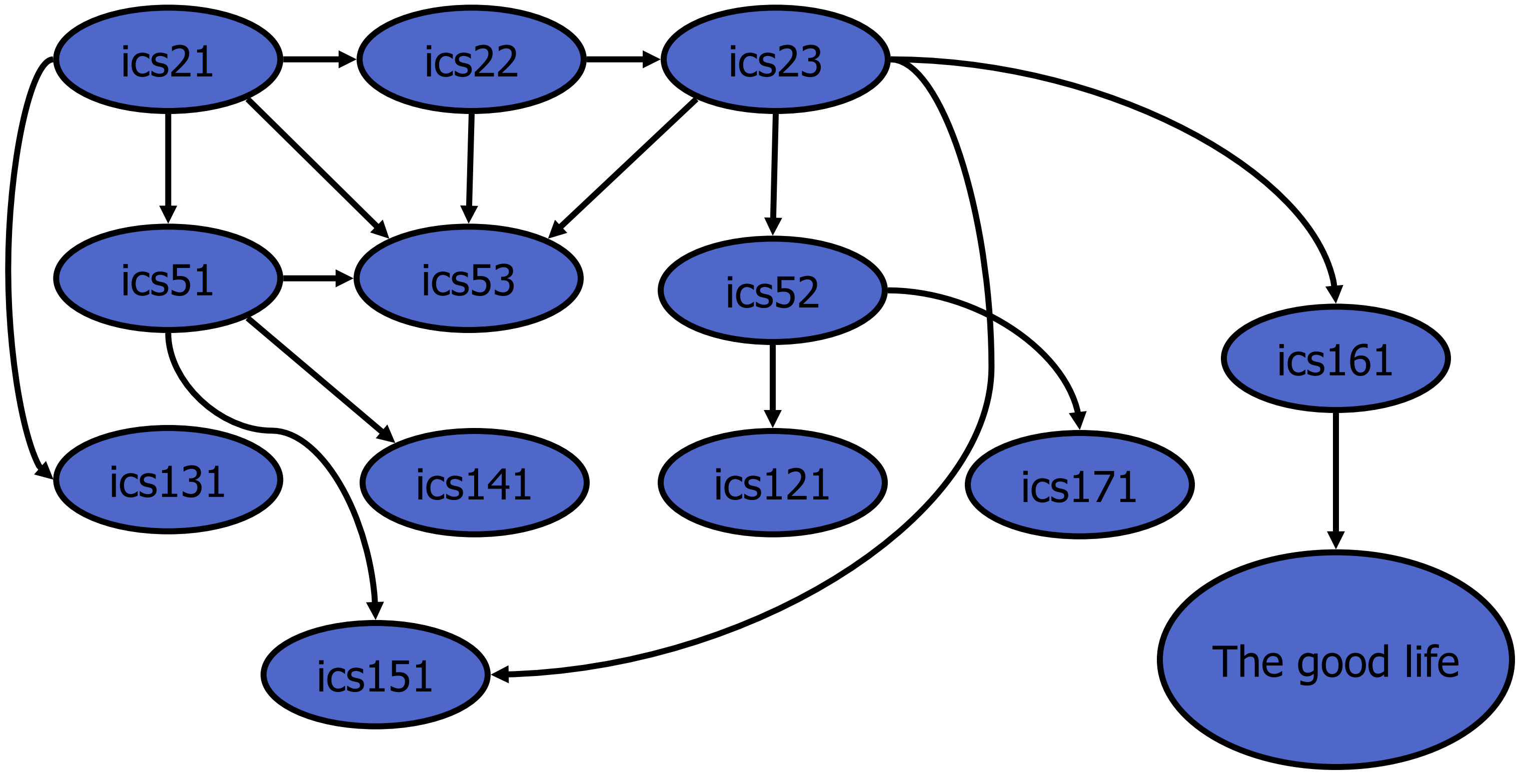
Applications

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



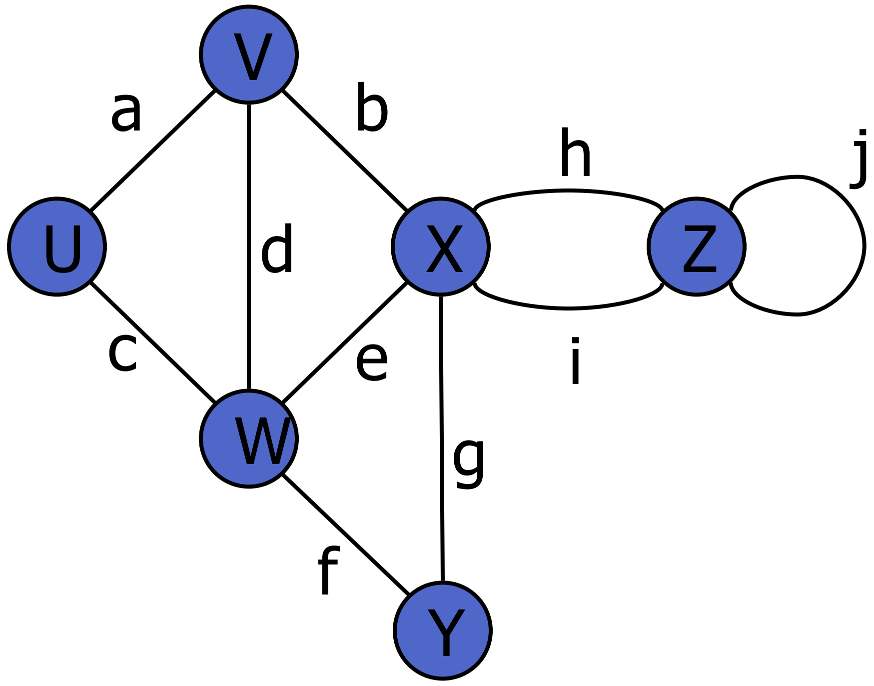
Digraph Application

- Scheduling: edge (a,b) means task a must be completed before b can be started



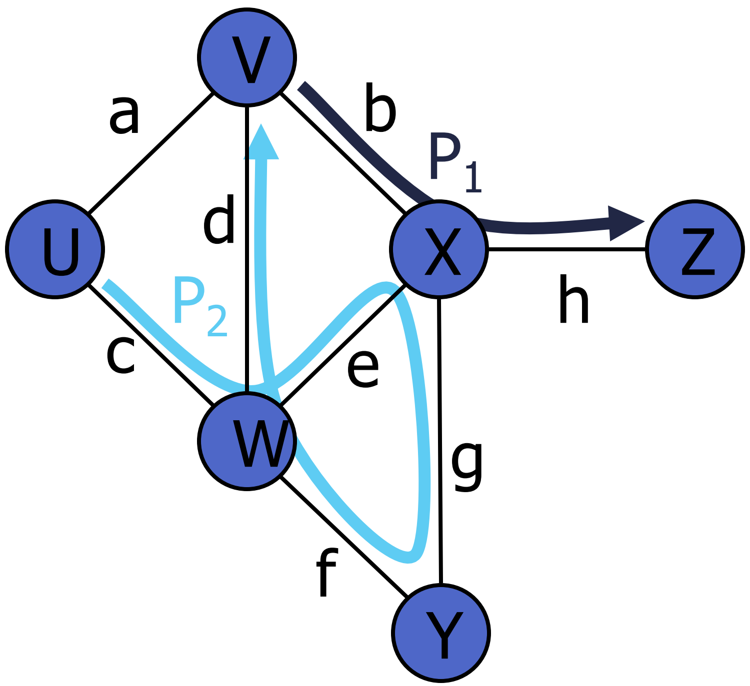
Terminology

- **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



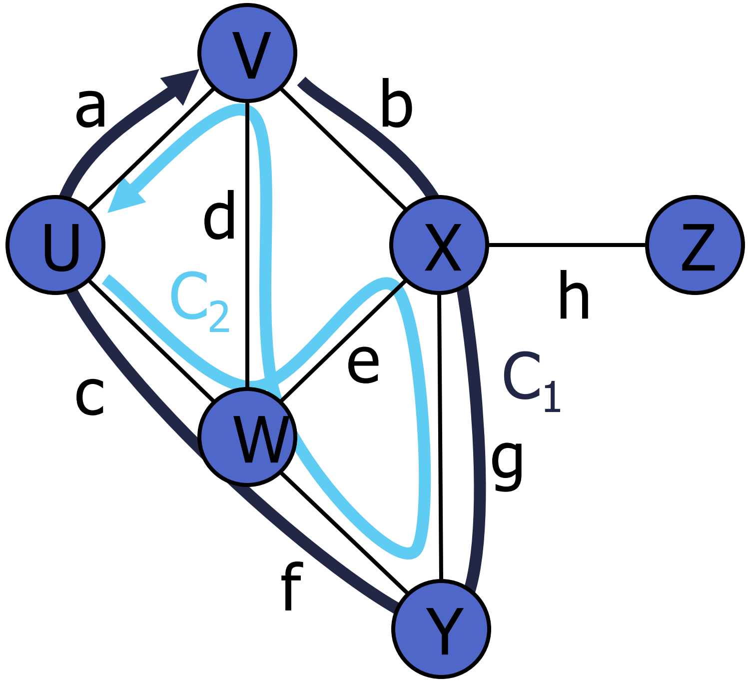
Terminology (cont.)

- **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 (cont.)

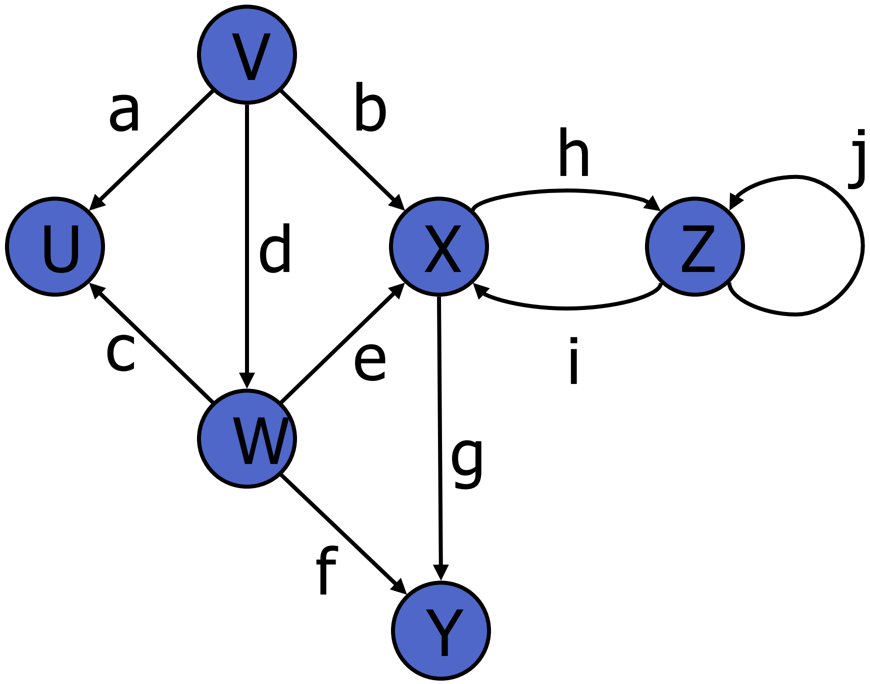
- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Terminology (cont.)

- **Origin** / **destination** endpoints
  - V is the ***origin*** of **b**
  - X is the ***destination*** of **b**

- **Incoming** / **outgoing** edges
  - **g, h** are ***outgoing*** edges of **X**
    - **Out-degree** of X = 2
  - **b**, **e**, **i** are ***incoming*** edges of **X**
    - **In-degree** of X = 3



Properties

Property 1

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

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

$m \leq n\,(n-1)/2$

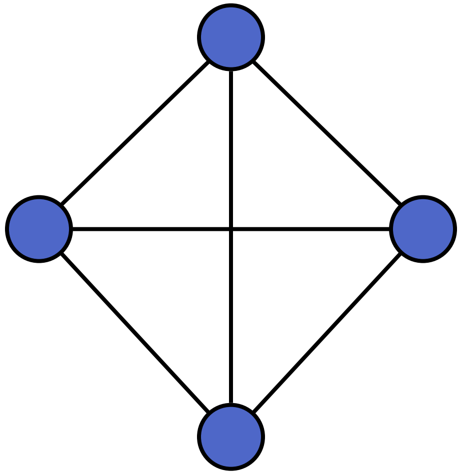
Proof: each vertex has degree at most $(n-1)$

Q: What is the bounds for a directed graph?

(no self-loops and no multiple edges)

Notation

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

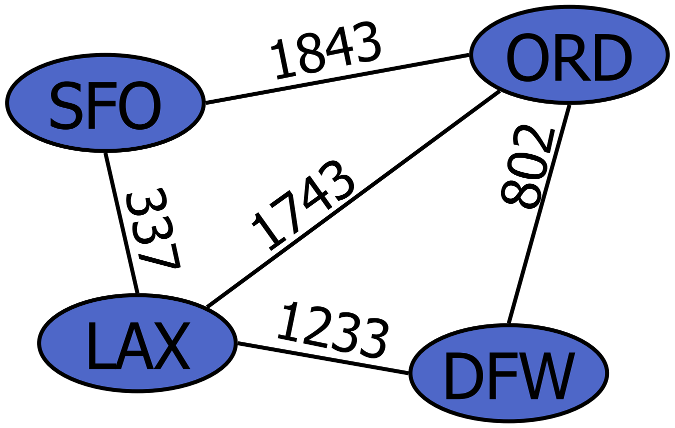


Example

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

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.
- In addition, we assume that an Edge supports the following methods:



endpoints( ): Return a tuple (u, v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.

opposite(v): Assuming vertex v is one endpoint of the edge (either origin or destination), return the other endpoint.

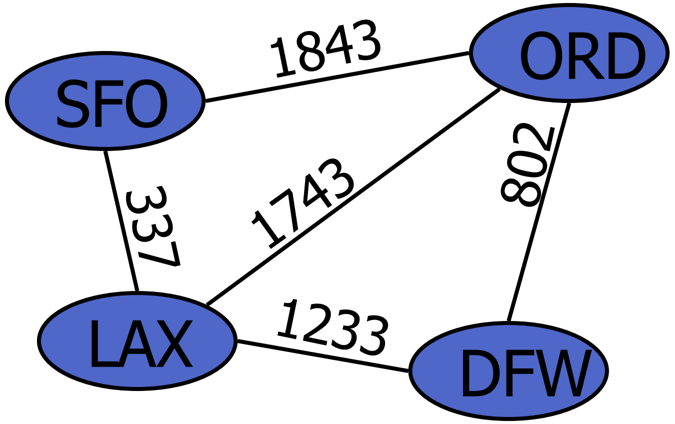
Vertex Class

```
#------------------------ nested Vertex class ------------------------
class Vertex:
  """Lightweight vertex structure for a graph."""
  __slots__ = '_element'

  def __init__(self, x):
    """Do not call constructor directly. Use Graph's insert_vertex(x)."""
    self._element = x

  def element(self):
    """Return element associated with this vertex."""
    return self._element

  def __hash__(self):          # will allow vertex to be a map/set key
    return hash(id(self))
```

Edge Class

```python
#------------------------- nested Edge class -------------------------
class Edge:
  """Lightweight edge structure for a graph."""
  __slots__ = '_origin', '_destination', '_element'

  def __init__(self, u, v, x):
    """Do not call constructor directly. Use Graph's insert_edge(u,v,x)."""
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    """Return (u,v) tuple for vertices u and v."""
    return (self._origin, self._destination)

  def opposite(self, v):
    """Return the vertex that is opposite v on this edge."""
    return self._destination if v is self._origin else self._origin

  def element(self):
    """Return element associated with this edge."""
    return self._element

  def __hash__(self):              # will allow edge to be a map/set key
    return hash( (self._origin, self._destination) )
```

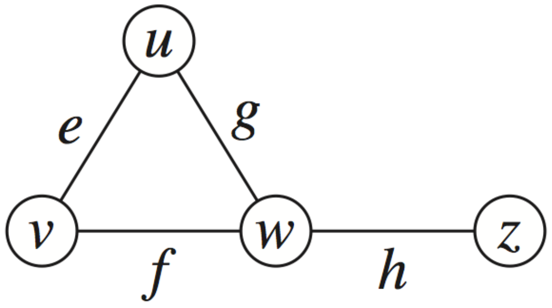
Graph ADT

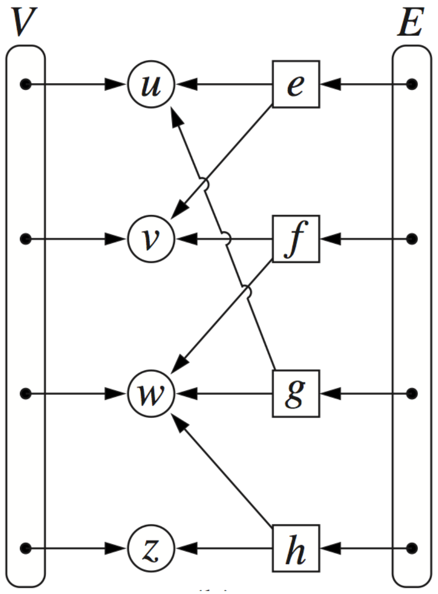


vertex_count():	Return the number of vertices of the graph.
vertices():	Return an iteration of all the vertices of the graph.
edge_count():	Return the number of edges of the graph.
edges():	Return an iteration of all the edges of the graph.
get_edge(u,v):	Return the edge from vertex u to vertex v, if one exists; otherwise return None. For an undirected graph, there is no difference between get_edge(u,v) and get_edge(v,u).
degree(v, out=True):	For an undirected graph, return the number of edges incident to vertex v. For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex v, as designated by the optional parameter.
incident_edges(v, out=True):	Return an iteration of all edges incident to vertex v. In the case of a directed graph, report outgoing edges by default; report incoming edges if the optional parameter is set to False.
insert_vertex(x=None):	Create and return a new Vertex storing element x.
insert_edge(u, v, x=None):	Create and return a new Edge from vertex u to vertex v, storing element x (None by default).
remove_vertex(v):	Remove vertex v and all its incident edges from the graph.
remove_edge(e):	Remove edge e from the graph.

Method 1) 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

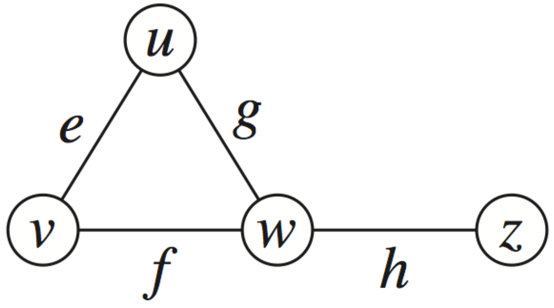


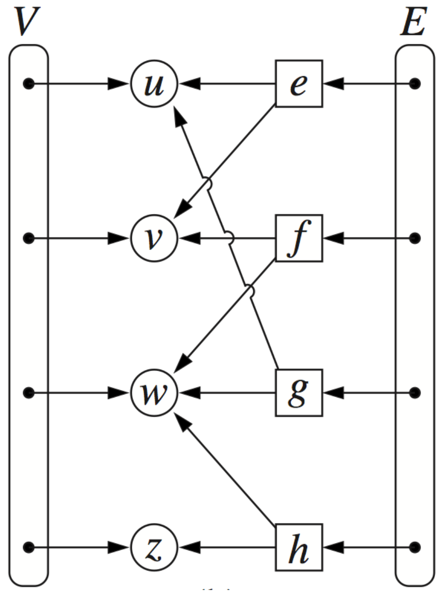



```
u = Vertex(elem1)
v = Vertex(elem2)
w = Vertex(elem3)
z = Vertex(elem4)

e = Edge(u,v, e_elem1)
f = Edge(v,w, e_elem2)
g = Edge(u,w, e_elem3)
h = Edge(w,z, e_elem4)

V = [u, v, w, z]
E = [e, f, g, h]
```

Performance

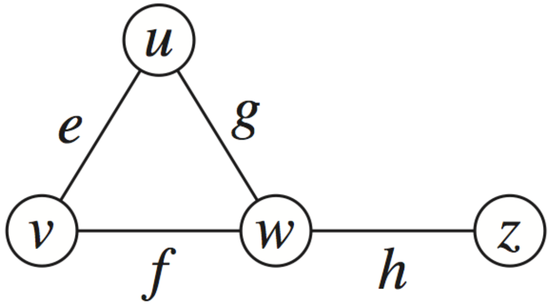
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space			
incidentEdges(v)			
areAdjacent (v, w)			
insertVertex(o)			
insertEdge(v, w, o)			
removeVertex(v)			
removeEdge(e)			

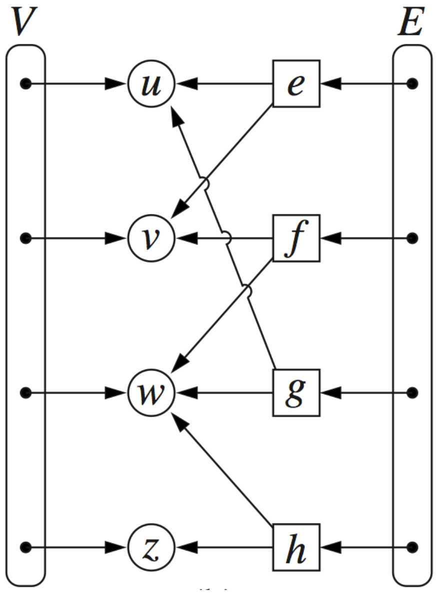




Performance

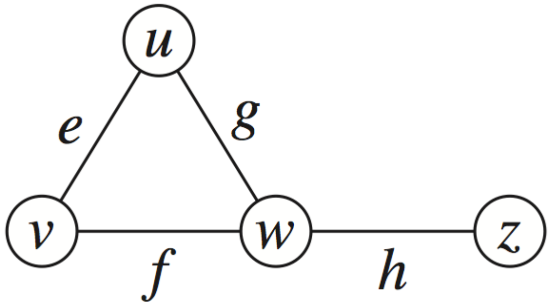
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$		
incidentEdges(v)	m		
areAdjacent (v, w)	m		
insertVertex(o)	1		
insertEdge(v, w, o)	1		
removeVertex(v)	m		
removeEdge(e)	1		

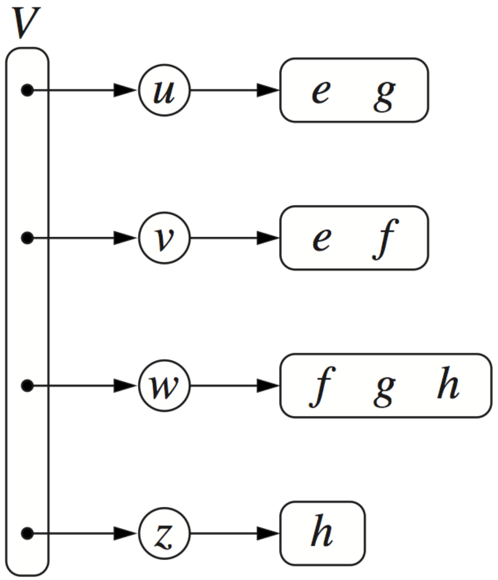




Method 2) Adjacency List Structure

- 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





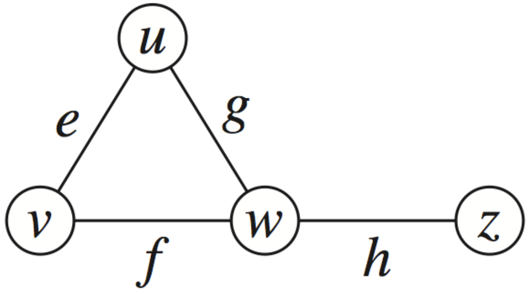

```
u = Vertex(elem1)
v = Vertex(elem2)
w = Vertex(elem3)
z = Vertex(elem4)

e = Edge(u,v, e_elem1)
f = Edge(v,w, e_elem2)
g = Edge(u,w, e_elem3)
h = Edge(w,z, e_elem4)

#idx:0  1  2  3
V = [u, v, w, z]

adj = [[e, g],     #0
       [e, f],     #1
       [f, g, h],  #2
       [h]]        #3
```

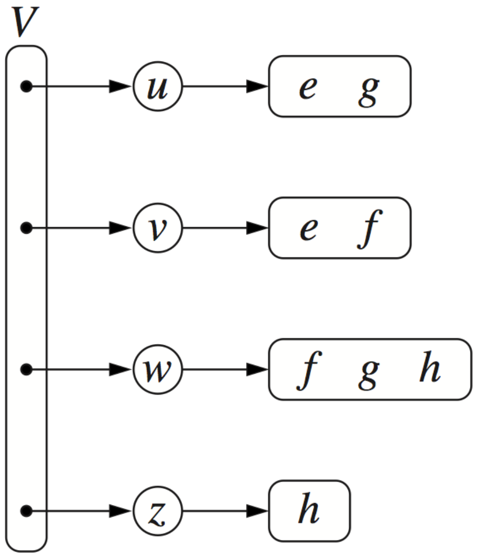
or

```
adj[u] = [e, g]
adj[v] = [e, f]
adj[w] = [f, g, h]
adj[z] = [h]
```

Performance

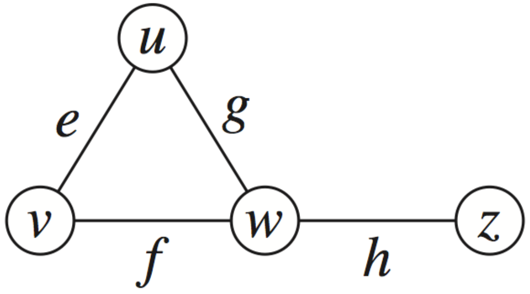
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$		
incidentEdges(v)	m		
areAdjacent (v, w)	m		
insertVertex(o)	1		
insertEdge(v, w, o)	1		
removeVertex(v)	m		
removeEdge(e)	1		

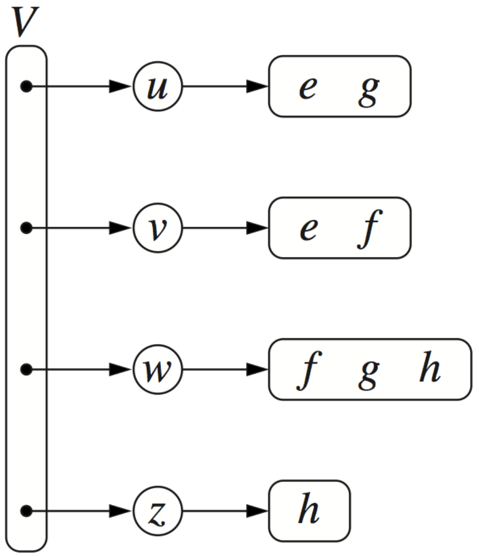




Performance

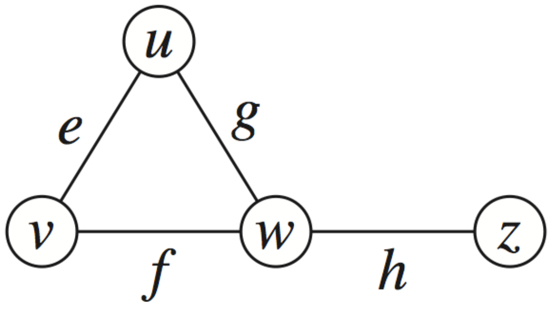
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	
incidentEdges(v)	m	$\deg(v)$	
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	
insertVertex(o)	1	1	
insertEdge(v, w, o)	1	1	
removeVertex(v)	m	$\deg(v)$	
removeEdge(e)	1	1	





Method 3) 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 nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge

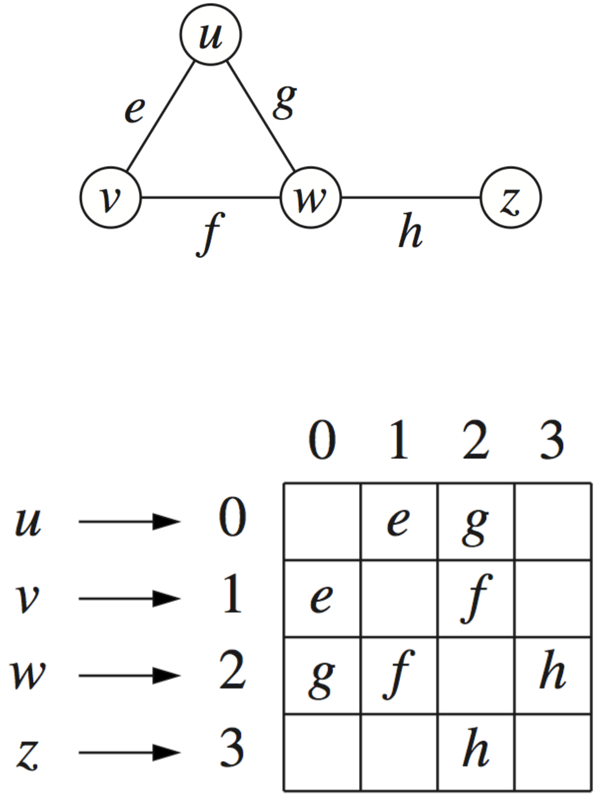


		0	1	2	3
$u \rightarrow$	0		e	g	
$v \rightarrow$	1	e		f	
$w \rightarrow$	2	g	f		h
$z \rightarrow$	3			h	

```
u = Vertex(elem1)
v = Vertex(elem2)
w = Vertex(elem3)
z = Vertex(elem4)

e = Edge(u,v, e_elem1)
f = Edge(v,w, e_elem2)
g = Edge(u,w, e_elem3)
h = Edge(w,z, e_elem4)

#idx:0  1  2  3
V = [u, v, w, z]
#          0       1       2      3
MAT = [[None,      e,      g, None], # 0
       [   e,   None,      f, None], # 1
       [   g,      f,   None,    f], # 2
       [None,   None,      h, None]] # 3
```

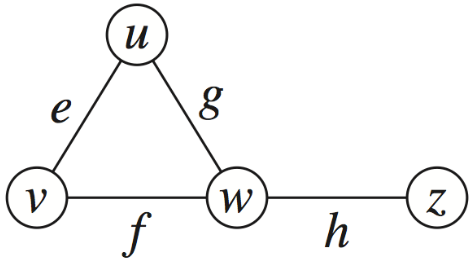
Performance

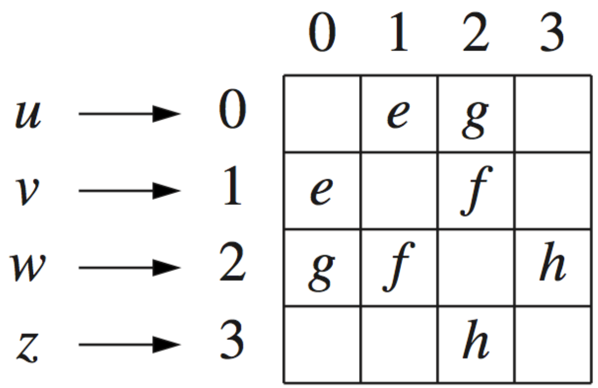
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	
incidentEdges(v)	m	$\deg(v)$	
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	
insertVertex(o)	1	1	
insertEdge(v, w, o)	1	1	
removeVertex(v)	m	$\deg(v)$	
removeEdge(e)	1	1	



Performance

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	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
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





Python Graph Implementation

- We use a variant of the ***adjacency map*** representation.
- For each vertex v, we use a Python dictionary to represent the secondary incidence map $I(v)$.
- The list V is replaced by a top-level dictionary D that maps each vertex v to its incidence map $I(v)$.
  - Note that we can iterate through all vertices by generating the set of keys for dictionary D.
- A vertex does not need to explicitly maintain a reference to its position in D, because it can be determined in $O(1)$ expected time.
- Running time bounds for the adjacency-list graph ADT operations, given above, become ***expected*** bounds.

Graph, Part 1

```
class Graph:
  """Representation of a simple graph using an adjacency map."""

  def __init__(self, directed=False):
    """Create an empty graph (undirected, by default).

    Graph is directed if optional paramter is set to True.
    """
    self._outgoing = { }
    # only create second map for directed graph; use alias for undirected
    self._incoming = { } if directed else self._outgoing

  def is_directed(self):
    """Return True if this is a directed graph; False if undirected.

    Property is based on the original declaration of the graph, not its contents.
    """
    return self._incoming is not self._outgoing # directed if maps are distinct

  def vertex_count(self):
    """Return the number of vertices in the graph."""
    return len(self._outgoing)

  def vertices(self):
    """Return an iteration of all vertices of the graph."""
    return self._outgoing.keys( )

  def edge_count(self):
    """Return the number of edges in the graph."""
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    # for undirected graphs, make sure not to double-count edges
    return total if self.is_directed( ) else total // 2

  def edges(self):
    """Return a set of all edges of the graph."""
    result = set( )          # avoid double-reporting edges of undirected graph
    for secondary_map in self._outgoing.values( ):
      result.update(secondary_map.values( ))    # add edges to resulting set
    return result
```

Graph, end

```
def get_edge(self, u, v):
  """Return the edge from u to v, or None if not adjacent."""
  return self._outgoing[u].get(v)          # returns None if v not adjacent

def degree(self, v, outgoing=True):
  """Return number of (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to count incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  return len(adj[v])

def incident_edges(self, v, outgoing=True):
  """Return all (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to request incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  for edge in adj[v].values():
    yield edge

def insert_vertex(self, x=None):
  """Insert and return a new Vertex with element x."""
  v = self.Vertex(x)
  self._outgoing[v] = { }
  if self.is_directed():
    self._incoming[v] = { }        # need distinct map for incoming edges
  return v

def insert_edge(self, u, v, x=None):
  """Insert and return a new Edge from u to v with auxiliary element x."""
  e = self.Edge(u, v, x)
  self._outgoing[u][v] = e
  self._incoming[v][u] = e
```