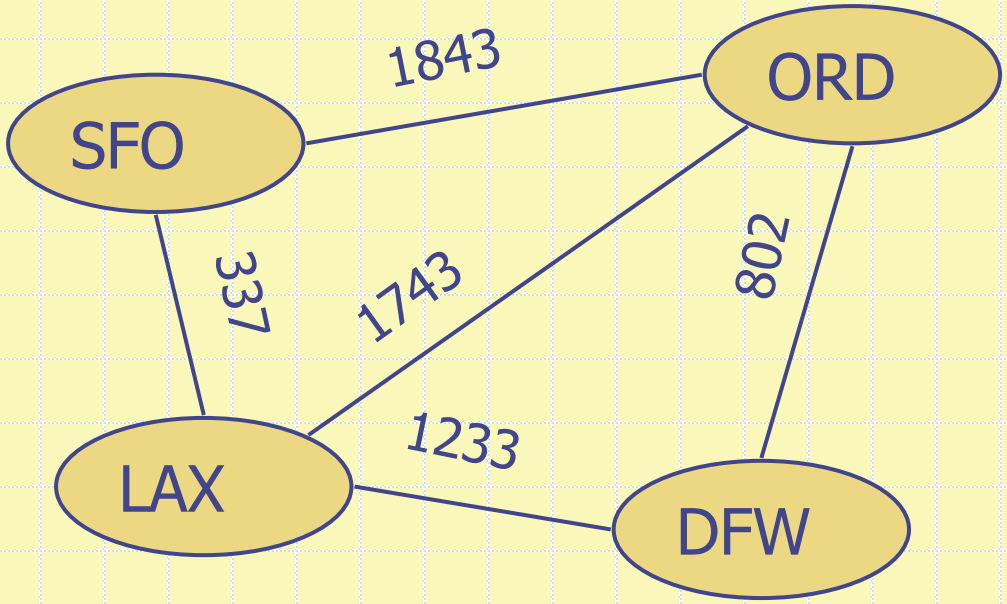
Graphs

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

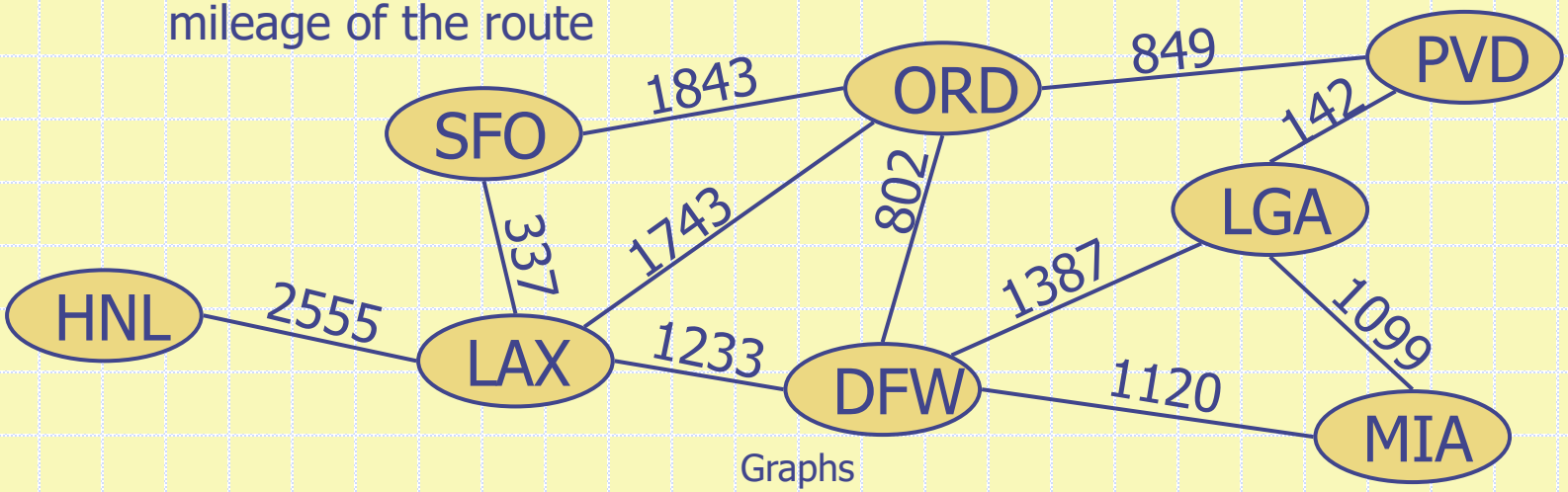
Coverage

- Graphs
- Data Structures for Graphs



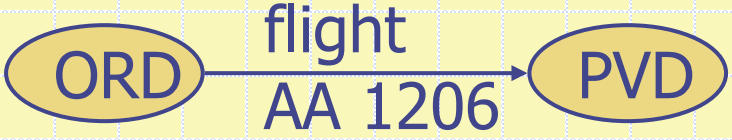
Graphs

- A graph is a collection of objects (called vertices) and connections (edges) between these vertices.
- A graph can hence be viewed as 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/values
- 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

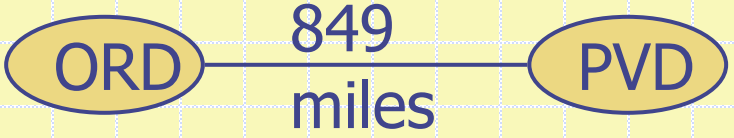


Edge Types

- Directed edge
  - **ordered** pair of vertices (u,v)
  - the first vertex, u, is the origin
  - the second vertex, v, is the destination
  - e.g., a flight



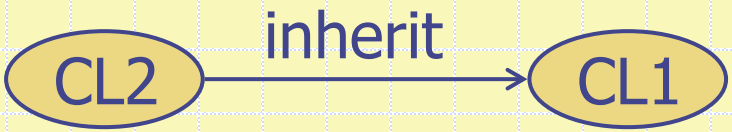
- Undirected edge
  - **unordered** pair of vertices (u,v)
  - e.g., a flight route, distance between two cities



Edge Types

- Examples:
  - A graph representing relations between authors. An edge exists if two authors have coauthored some publication. The edges are undirected since co-authorship is a ***symmetric*** relation; that is if A has co-authored a book with B then B must has co-authored something with A.

  - A graph representing relationship between different inherited classes of an object-oriented language. Such edges (between a parent and child class) are directed edges since inheritance relationship is ***asymmetric***; since inheritance goes only one way.

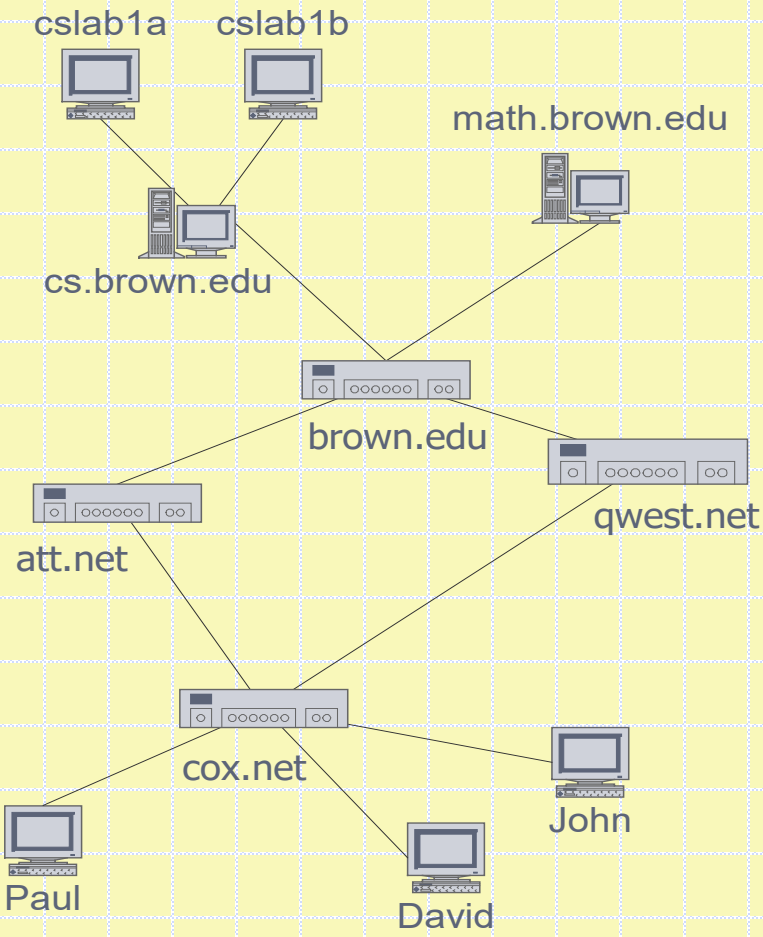


Directed, Undirected & Mixed Graphs

- A **directed graph** (also referred to as *digraph*) is a graph where:
  - **all** the edges are directed
    - e.g., route network
- An **undirected graph** is a graph where:
  - **all** the edges are undirected
    - e.g., flight network
- An **mixed graph** is a graph where:
  - Some of the edges are directed and some are undirected
- It is possible to convert a mixed or undirected graph to a directed graph by replacing every undirected edge (u, v) by two directed edges (u, v) and (v, u)
  - However, it is often useful to keep such graphs as they are

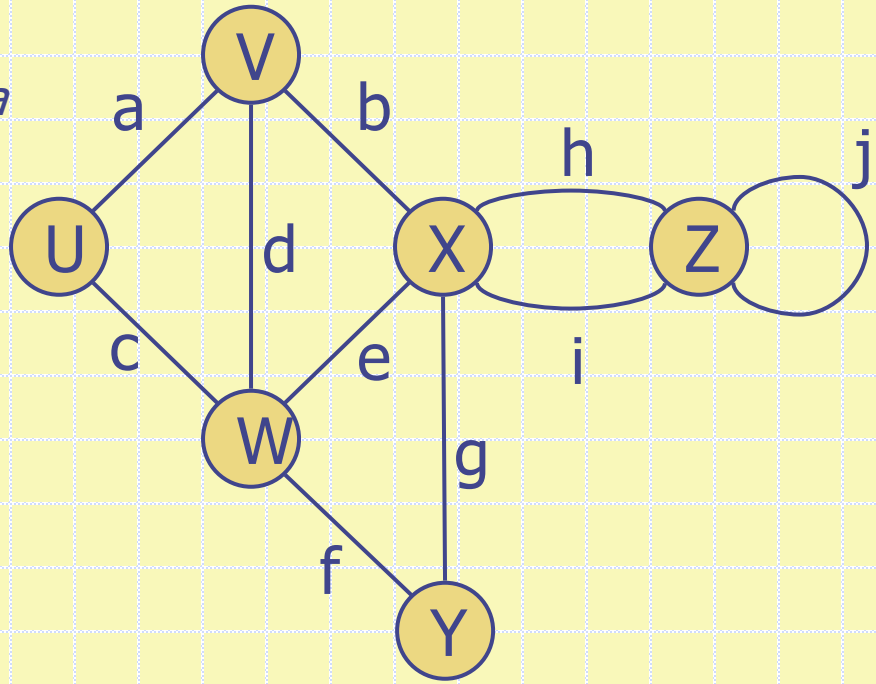
Applications

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



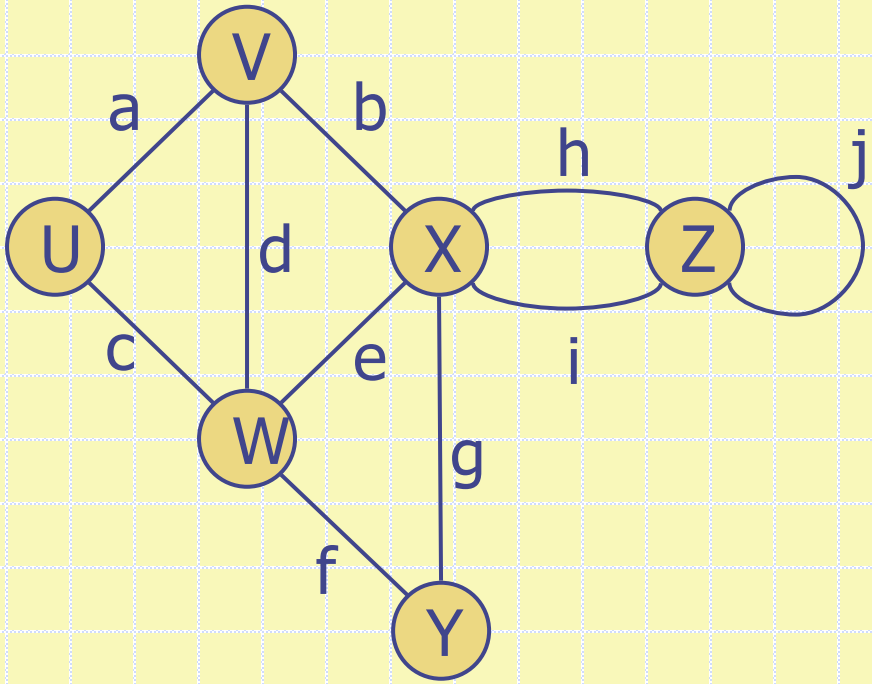
Terminology

- The two vertices joined by an edge are referred to as ***end vertices*** or ***endpoints***.
  - U and V are the endpoints of *a*
- If an edge is directed, then its first endpoint is referred to as the ***origin***, while its second endpoint is referred to as ***destination***
- Two vertices are called ***adjacent*** if there is an edge with these two vertices as endpoints .
  - W, and Y are adjacent. V and Z are not



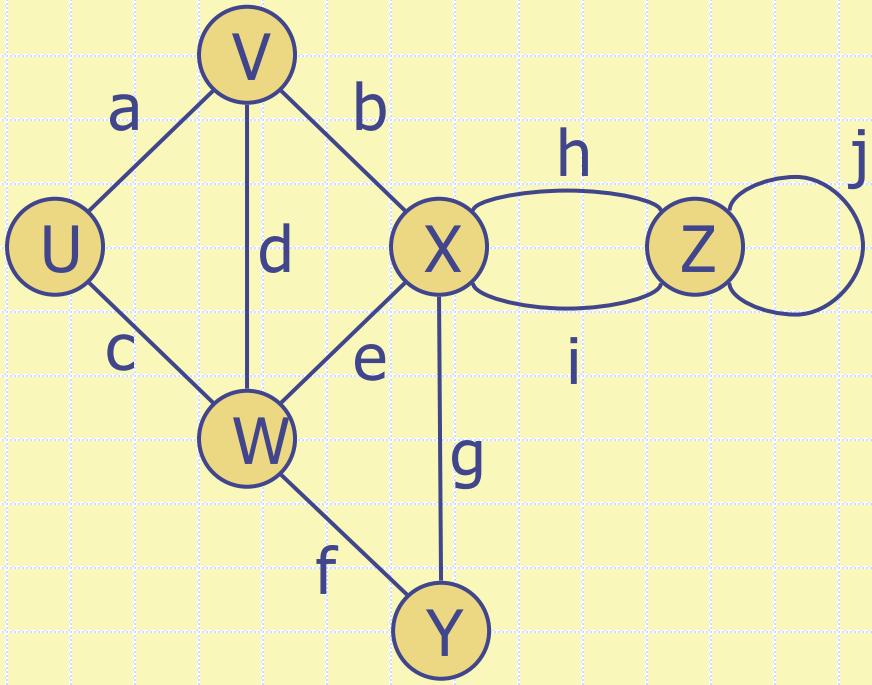
Terminology

- An edge is said to be *incident on a vertex* if the vertex is one of the edge's endpoints
  - e is incident on X; e is also incident on W
- The *outgoing edges* of a vertex are those directed edges whose origin is that vertex
- The *incoming edges* of a vertex are those directed edges whose destination is that vertex



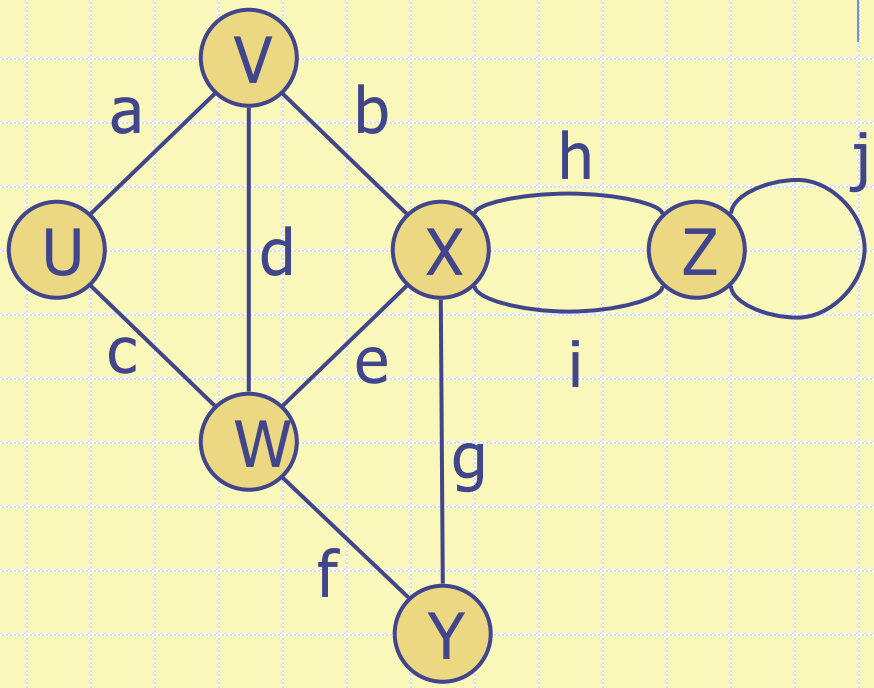
Terminology

- The ***degree*** of a vertex v, ***deg(v)***, is the number of incident edges of that vertex v
  - X has degree 5; that is $deg(x) = 5$
- The ***in-degree*** of a vertex v, ***indeg(v)***, is the number of directed incoming edges of that vertex v
- The ***out-degree*** of a vertex v, ***outdeg(v)***, is the number of directed outgoing edges of that vertex v



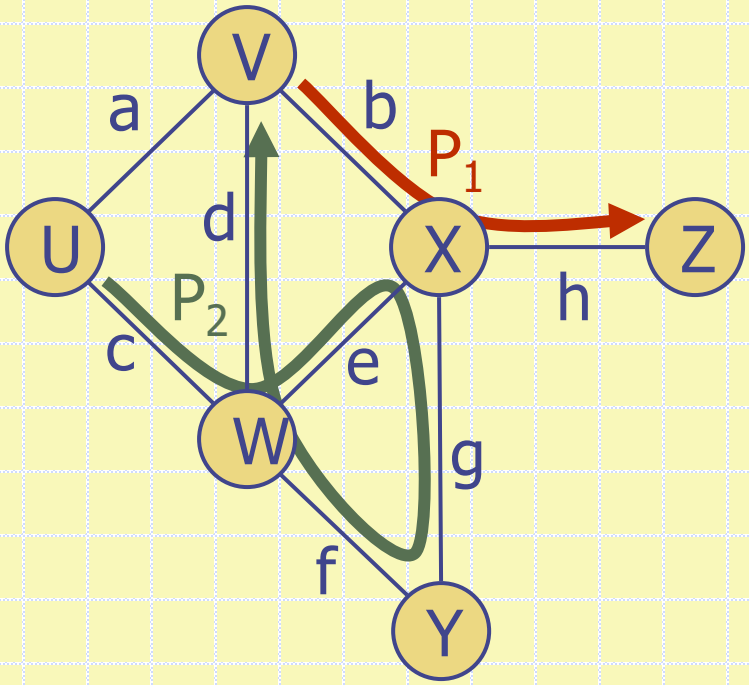
Terminology

- A Graph refers to the group of edges as a *collection* and **NOT** as a *set*
- This classification allows two undirected edges to have the same endpoints and allows two directed edges to have the same origin and the same destination (i.e. two different flights from one city to another)
- Such edges are referred to as *Parallel edges* or *multiple edges*
  - h and i are parallel edges
- *Self-loop* referred an edge that connects one vertex to itself
  - j is a self-loop



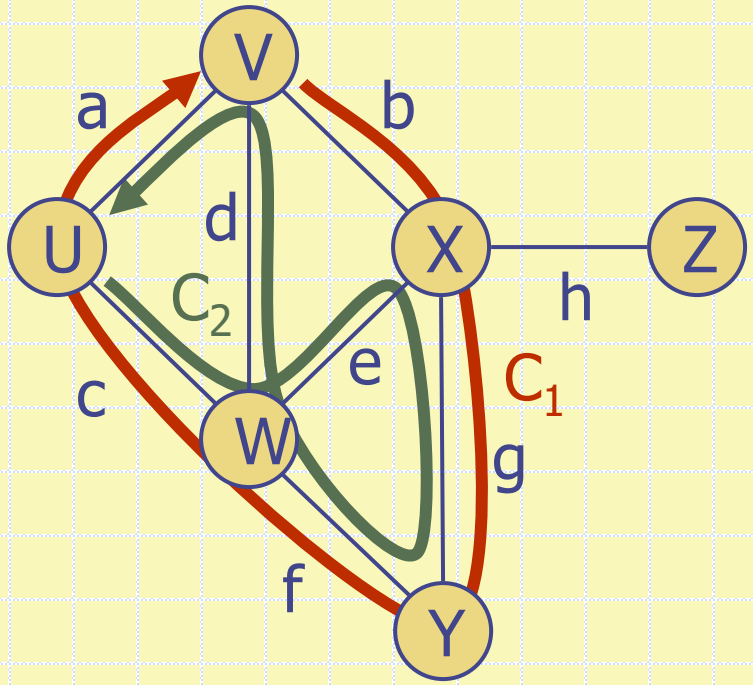
Terminology (cont.)

- *Path*
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- *Directed path*
  - path such that all edges are directed and are traversed along their directions
- *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



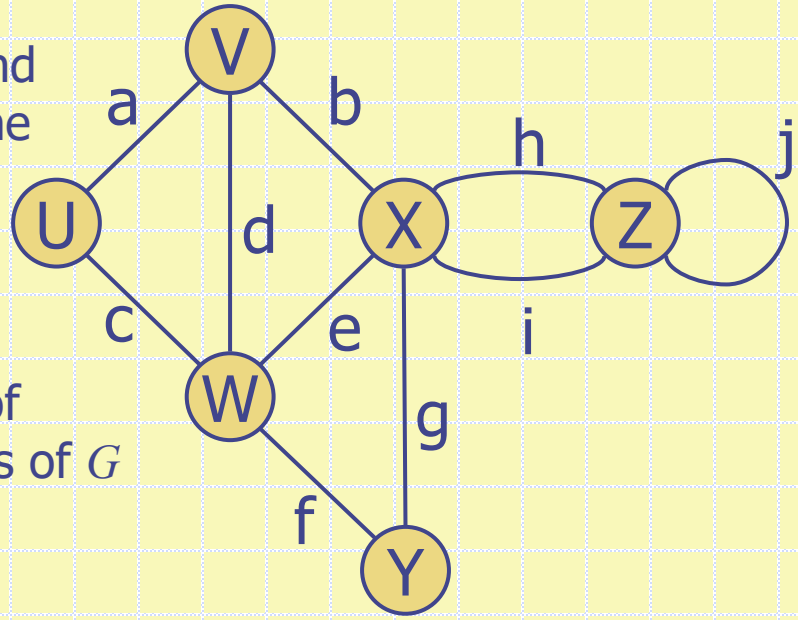
Terminology (cont.)

- *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.)

- ***Subgraph***
  - A graph H is a subgraph of graph G if all vertices and edges of H are subsets of the edges and vertices of G
  - For example, vertices W, Y and X and edges e and f form a subgraph of the shown graph

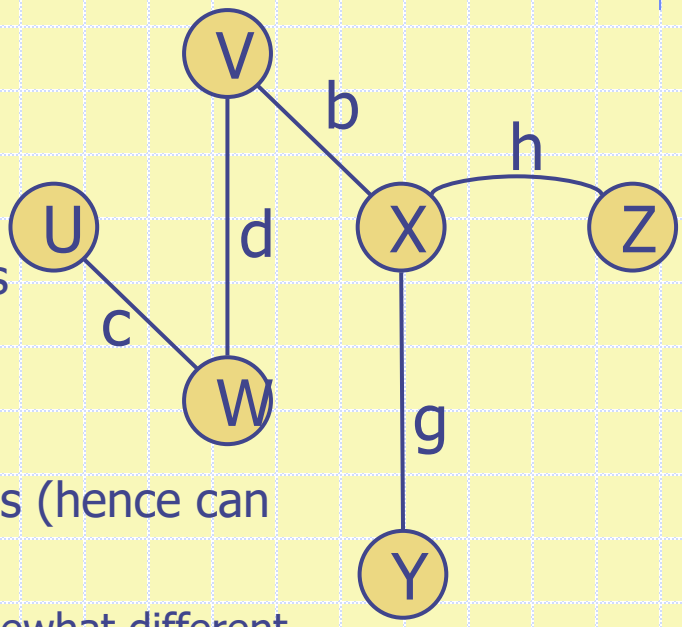


- ***Spanning Subgraph***
  - A graph H is a spanning subgraph of graph G if H contains all the vertices of G

- ***Connected Graph***
  - A graph is connected if there is a possible path between any two of its vertices. In other words, it is all in one piece.

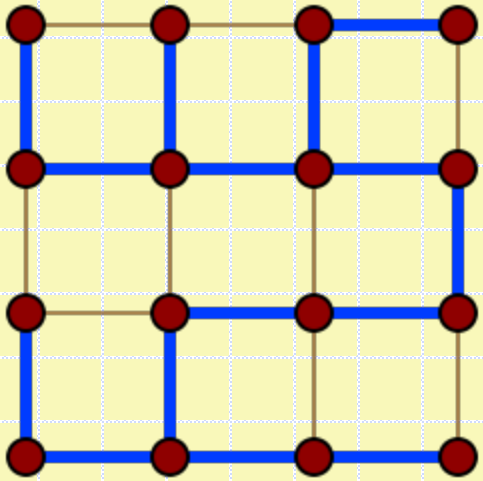
Terminology (cont.)

- ***Connected Components***
  - If a graph G is a *disconnected* graph, then its maximal connected subgraphs are called the connected components of G.
- ***Forest***
  - A forest is a graph without cycles. Notice that a forest may have many components
- ***Tree***
  - A tree is a connected graph without cycles (hence can be though of as a connected forest)
  - NOTE: In the context of graphs, a tree is somewhat different than what we have previously introduced as trees
  - In the context of graphs, a tree here has no root
  - To eliminate ambiguity, trees as we defined before are referred to as ***rooted trees***
  - Trees as defined here are referred to as ***free trees***



Terminology (cont.)

- ***Spanning Trees***
  - A spanning tree T of a graph G is a spanning subgraph of G that is a free tree
  - In other words, a spanning tree T of a graph G is:
    - A subgraph that has all the vertices of G,
    - All vertices are connected (that is one piece; a path exists between any two vertices)
    - Has no cycles
  - A spanning tree of a graph G can then be defined as:
    - A maximal set of edges of G that contains no cycle, or as
    - A minimal set of edges that connect all vertices



A spanning tree (blue heavy edges) of a grid graph*

*Source: http://en.wikipedia.org/wiki/Spanning_tree

Graph Properties (Assume Simple Graphs Here)

- Property 1:

  The sum of all degrees in a graph is equal to twice the number of edges in the graph

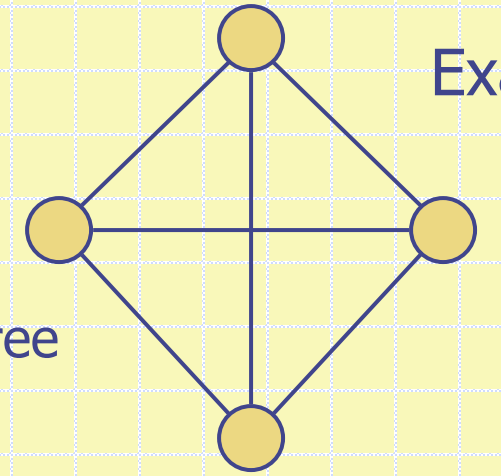
$$\sum_v \deg(v) = 2m$$

Proof:

- each edge is counted twice when we calculate the degree (since each connection gives/adds 1 value to the degree at both ends)
- In other words, each edge adds 2 to the degree

Notation

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



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$
- $\sum_v \deg(v) = 3 * 4 = 12 = 2m$

Graph Properties (Assume Simple Graphs Here)

- Property 2:

  If G is a directed graph then the sum of in-degree equals to the sum of the out-degree and equals to the number of edges

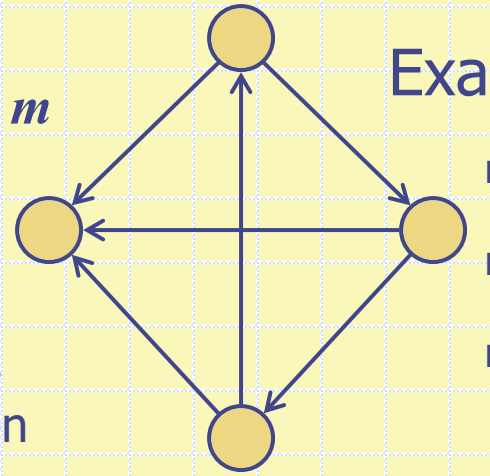
$$\Sigma_v \text{indeg}(v) = \Sigma_v \text{outdeg}(v) = m$$

Proof:

- each directed edge is counted once when we calculate the in-degree and counted once when we calculate the out-degree (since each connection gives/adds 1 value to the in-degree and 1 value to the out-degree)

Notation

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



Example

- $n = 4$
- $m = 6$
- $\Sigma_v \text{indeg}(v) = \Sigma_v \text{outdeg}(v) = 6 = m$

Graph Properties (Assume Simple Graphs Here)

Property 3

The number of edges of an undirected simple graph is less than or equal to $n(n-1)/2$, where n is the number of vertices

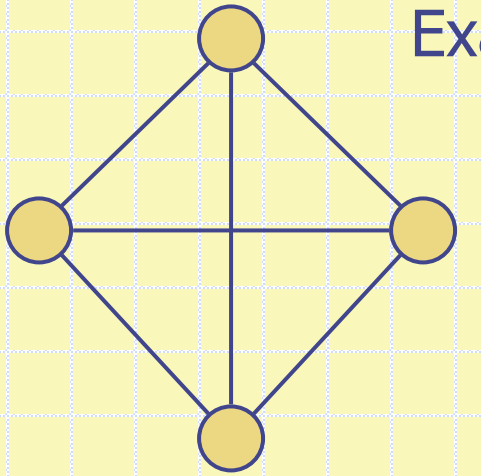
$$m \leq n(n-1)/2$$

Proof:

- each vertex has degree of at most $(n-1)$; this is the case when every other vertex has an edge with that vertex
- The maximum possible total (sum) degree is hence $n(n-1)$
- Using Property 1, $2m <= n(n-1)$ ➔ $m \leq n(n-1)/2$

Notation

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



Example

- $n = 4$
- $m = 6$
- $2m = 12 <= 4 * 3 = 12$

Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements
- **Accessor methods**
  - endVertices(e): return an array of the two end vertices of e
  - opposite(v, e): return the vertex opposite of v on e; error if e is not incident of v
  - areAdjacent(v, w): return *true* if v and w are adjacent; *false* otherwise
  - replace(v, x): replace element at vertex v with x
  - replace(e, x): replace element at edge e with x

Main Methods of the Graph ADT

- **Update methods**
  - insertVertex(x): insert and return a new vertex storing element x
  - insertEdge(v, w, x): insert and return a new edge (v,w) storing element x
  - removeVertex(v): remove vertex v and all its incident edges, and return the element stored at v
  - removeEdge(e): remove edge e and return the element stored at e
- **Iterable collection methods**
  - vertices(): return an iterable collection of all the vertices of the graph
  - edges(): return an iterable collection of all the edges of the graph
  - incidentEdges(v): return an iterable collection of all the edges incident upon vertex v

Data Structures for Graphs

- There are three popular ways of representing graphs, which are referred to as:
  - **Edge List Structure**
  - **Adjacency List Structure**
  - **Adjacency Matrix Structure**
- All three use a collection to store the vertices of the graph
- However, for edges, there is a fundamental difference between the first two structures and the latter
- The first two structures only store edges actually present in the graph, resulting in a $O(n + m)$ space complexity (n vertices and m edges)
- The adjacency matrix structure stores an edge placeholder for each pair of vertices whether or not the edge exists, resulting in a $O(n^2)$ space complexity

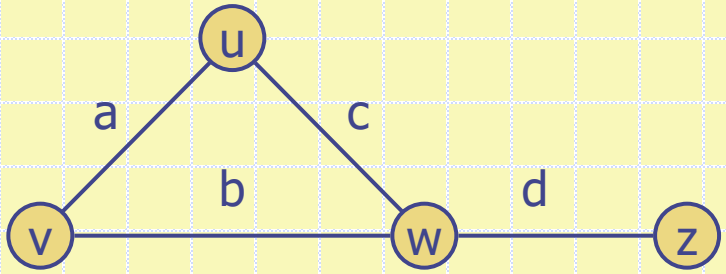
Edge List Structure

- In this representation of a graph, both vertices and edges are represented as objects
- The vertices objects are stored in a collection/sequence V, and the edges are stored in another collection E
- V and E can be array lists, node lists or other types
- Each entry in V points to a vertex object
- Each entry in E points to an edge object
- A vertex object stores the value of that vertex and a pointer to its location in V
- An edge object stores the value of that edge, two pointers to its two vertices, and a pointer to its location in E

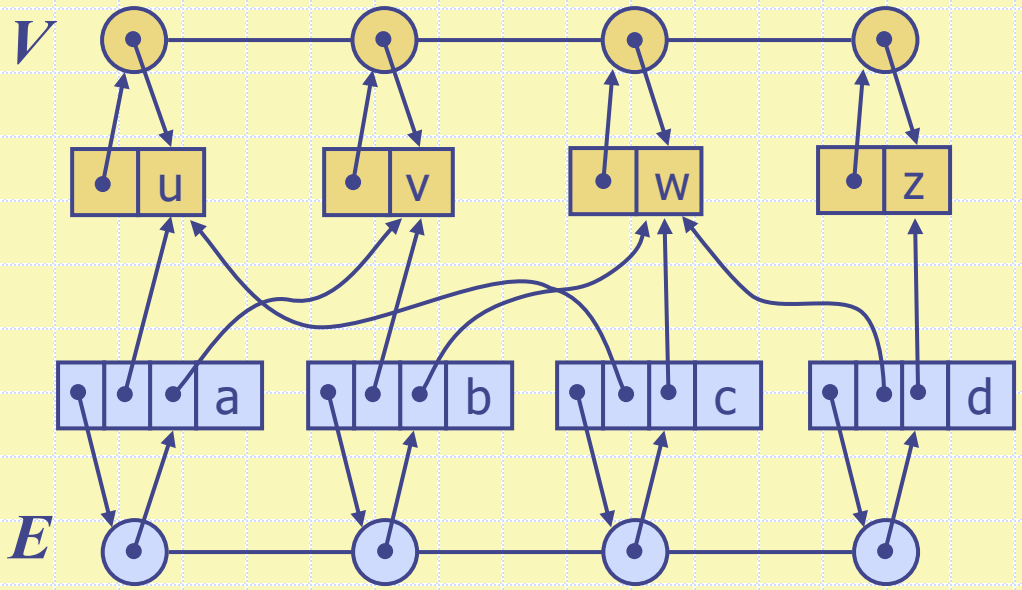
Edge List Structure

- 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

Graph



Edge List



Edge List Structure Analysis

- Edge list is simple to implement, however it has significant performance limitations

- Finding the end points of an edge: $O(?)$
- Finding the opposite end point of an edge: $O(?)$
- The insertion of a vertex: $O(?)$
- The insertion of an edge: $O(?)$
- The removal of an edge: $O(?)$

- Finding the incident edges of a vertex v: $O(?)$

- Finding if a vertex w is adjacent to a vertex v: $O(?)$

- Removing a vertex: $O(?)$

Edge List Structure Analysis

- Edge list is simple to implement, however it has significant performance limitations
- Finding the end points of an edge, or the opposite vertex of an edge is $O(1)$
- The insertion of a vertex, the insertion of an edge, as well as the removal of an edge is also $O(1)$
- However, to find the incident edges of a vertex v, a complete search of the edges list is required, resulting in $O(m)$ instead of $O(deg(v))$
- Finding if a vertex w is adjacent to a vertex v would also require searching all edges to find an edge with endpoints v and w, resulting in $O(m)$
- Removing a vertex would also result in $O(m)$, since an entire search of the list is required to remove all incident edges of that vertex.

Edge List Structure Analysis

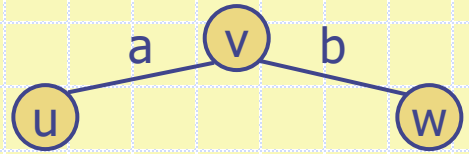
Operation	Complexity
endVertices(e), opposite(v, e)	$O(1)$
replace(v, x), replace(e, x)	$O(1)$
insertVertex(x), insertEdge(v, w, x), removeEdge(e)	$O(1)$
incidentEdges(v)	$O(m)$
areAdjacent(v, w)	$O(m)$
removeVertex(v)	$O(m)$
vertices()	$O(n)$
edges()	$O(m)$

Space Complexity	$O(m + n)$

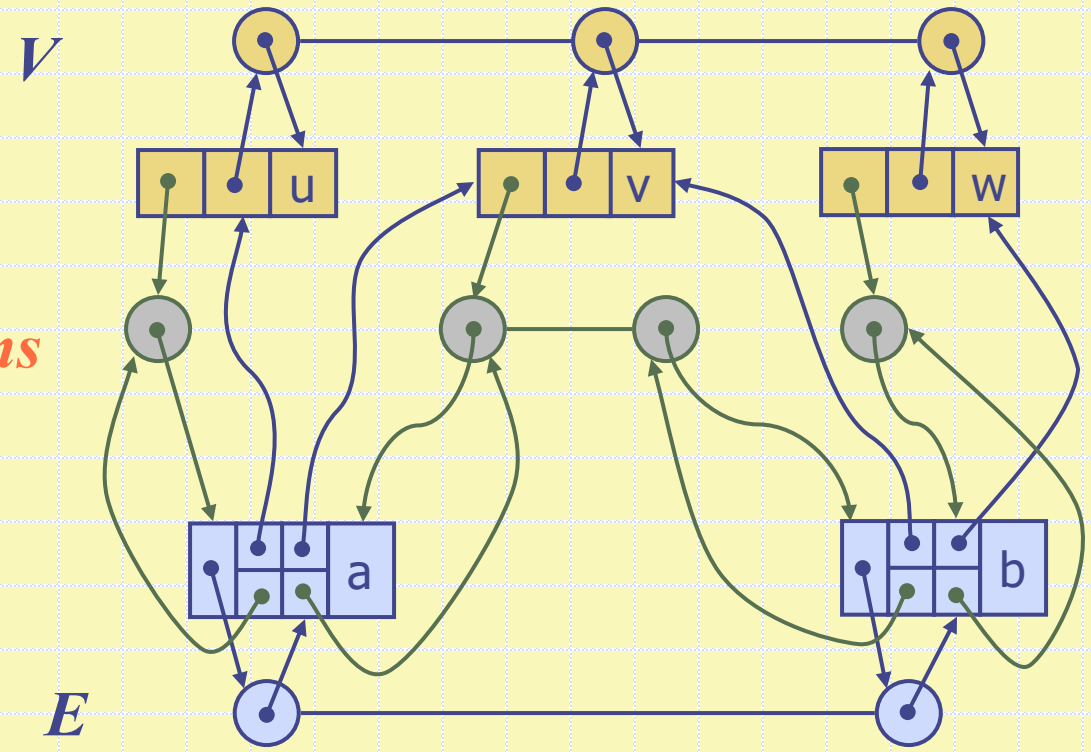
Adjacency List Structure

- The adjacency list structure includes all the structural components of the edge list structure plus:
  - A vertex object v holds a reference to a collection, referred to as the ***incidence collection*** of v or $l(v)$. The incidence collection of v is traditionally a ***l***ist whose elements store references to the edges incident on v.
  - The edge object for an object e connecting vertices v and w has an additional two references, which reference the two elements associated with that edge e in $l(v)$ and $l(w)$. This allows for finding all incident edges of a vertex from an edge to that vertex.
  - The space utilization is hence still proportional to the number of vertices and edges of the graph

Adjacency List Structure



Adjacency List



Adjacency List Structure Analysis

- Finding the end points of an edge: $O(?)$
- Finding the opposite end point of an edge: $O(?)$
- The insertion of a vertex: $O(?)$
- The insertion of an edge: $O(?)$
- The removal of an edge: $O(?)$

- Finding the incident edges of a vertex v: $O(?)$

- Finding if a vertex w is adjacent to a vertex v: $O(?)$

- Removing a vertex: $O(?)$

Adjacency List Structure Analysis

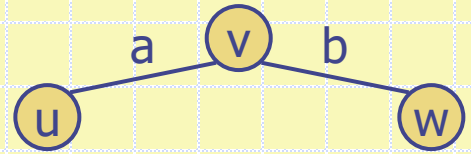
Operation	Complexity
endVertices(e), opposite(v, e)	$O(1)$
replace(v, x), replace(e, x)	$O(1)$
insertVertex(x), insertEdge(v, w, x), removeEdge(e)	$O(1)$
incidentEdges(v)	$O(deg(v))$
areAdjacent(v, w) We can find that by inspecting the incidence collection of v or the incidence collection of w. We choose to inspect the smaller one.	$O(min(deg(v), deg(w)))$
removeVertex(v) Go to the incidence collection of v and remove all edges pointed by its elements	$O(deg(v))$
vertices()	$O(n)$
edges()	$O(m)$

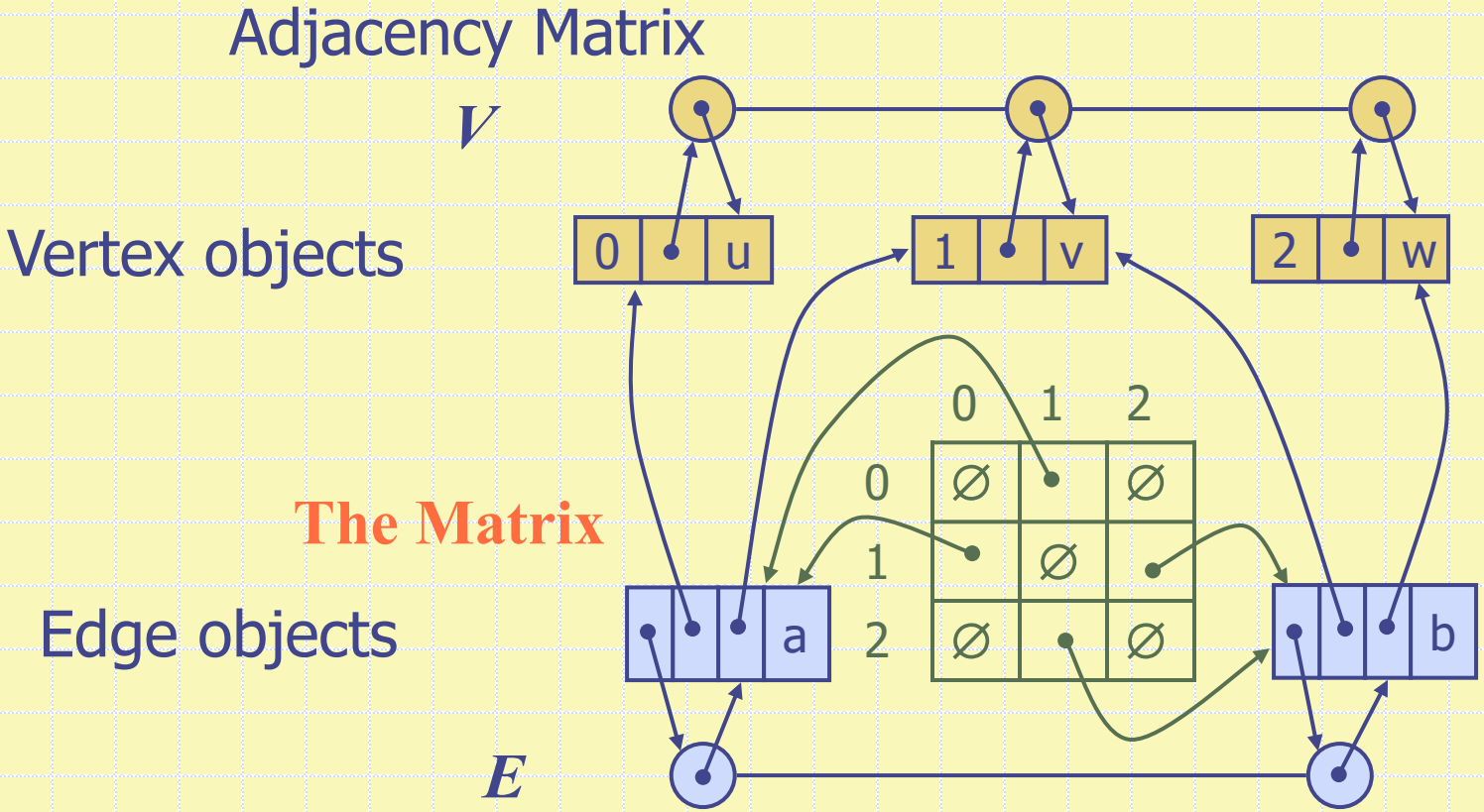
Space Complexity	$O(m + n)$

Adjacency Matrix Structure

- The adjacency matrix structure includes all the structural components of the *edge list* structure plus:
  - The vertex object is extended to include a distinct integer i in the range $[0 - N\text{-}1]$, which is referred to as the index of v
  - A matrix, which is a two-dimensional array A of size n x n, such that $A[i, j]$ holds a reference to the edge (v, w), where v is the vertex with index i and w is the vertex with index j. If the edge (v, w) does not exist, then $A[i, j]$ holds **null**.

Adjacency Matrix Structure





Adjacency Matrix Structure Analysis

- Finding the end points of an edge: $O(?)$
- Finding the opposite end point of an edge: $O(?)$
- The insertion of an edge: $O(?)$
- The removal of an edge: $O(?)$
- The insertion of a vertex: $O(?)$
- Finding the incident edges of a vertex v: $O(?)$
- Finding if a vertex w is adjacent to a vertex v: $O(?)$
- Removing a vertex: $O(?)$

Adjacency Matrix Structure Analysis

Operation	Complexity
endVertices(e), opposite(v, e)	$O(1)$
replace(v, x), replace(e, x)	$O(1)$
insertEdge(v, w, x), removeEdge(e)	$O(1)$
incidentEdges(v) We need to examine an entire row or column then follow the non-null links to find the exact edges	$O(n + deg(v)) \rightarrow O(n)$
areAdjacent(v, w)	$O(1)$
removeVertex(v), insertVertex(x) Any addition or removal require the creation/construction of a whole new matrix with larger or smaller size respectively	$O(n^2)$
vertices()	$O(n)$
edges()	$O(m)$

Space Complexity	$O(n^2)$

Performance Comparison

▪ 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

- Historically, Adjacency Matrix was proposed first to represent graphs.
- With the exception of areAdjacent(), Adjacency List performs best compared to the other two structures.