Data Structures & Algorithms 2

Topic 7 – Graphs (part 1)

Lecturer: Dr. Hadi Tabatabaee
Materials: Dr. Phil Maguire & Dr. Hadi Tabatabaee
Maynooth University
Online at http://moodle.maynoothuniversity.ie

Overview

Aims

- Introducing graphs and their properties

Learning outcomes: You should be able to…

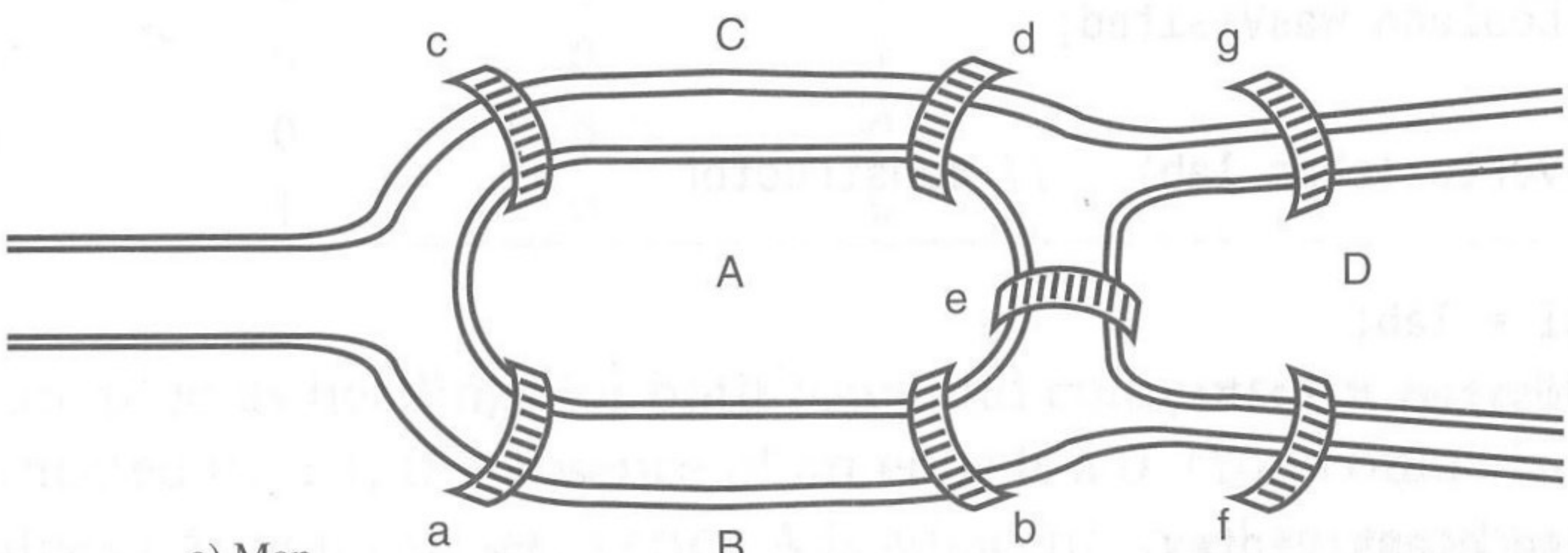
- Learn different types of graphs and their properties
- Learn Graph ADT and different presentations of graphs
- Build edge list structure
- Build adjacency list
- Build adjacency matrix

Graphs

- Graphs are one of the most versatile structures used in computer programming and are used to solve the more interesting problems.
- There are two types of graphs:
  - Unweighted graphs where the links between nodes are equal.
  - Weighted graphs where the links are each associated with an individual weight.
- The shape of the graph is dictated by the problem you want to solve and represents some real-world situation.

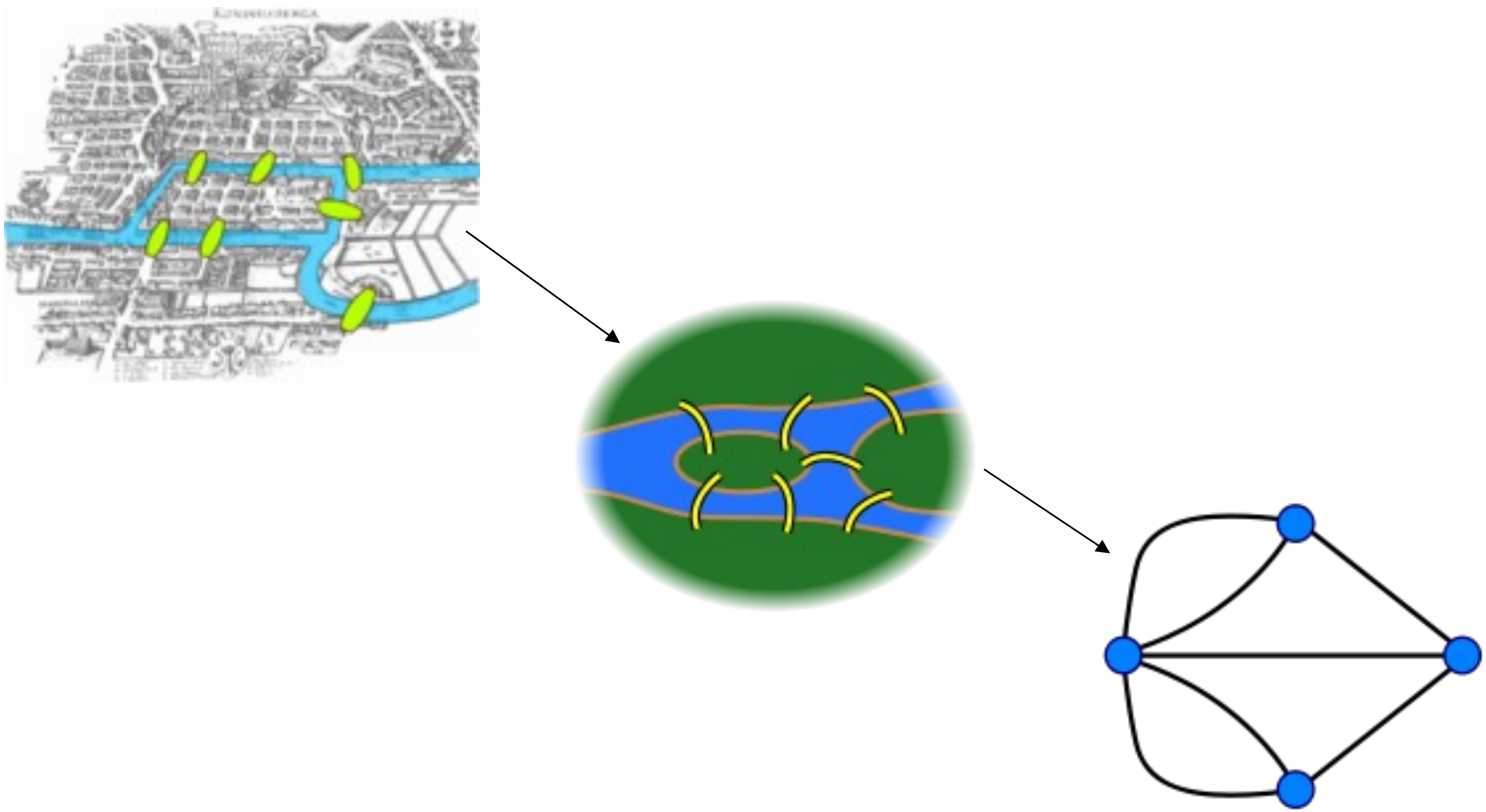
Real-world problems

- Graphs are used to represent real-world problem such as airline routes, electrical circuits and job scheduling.
- For example, can you find a way to cross all seven bridges in Konigsberg only once?



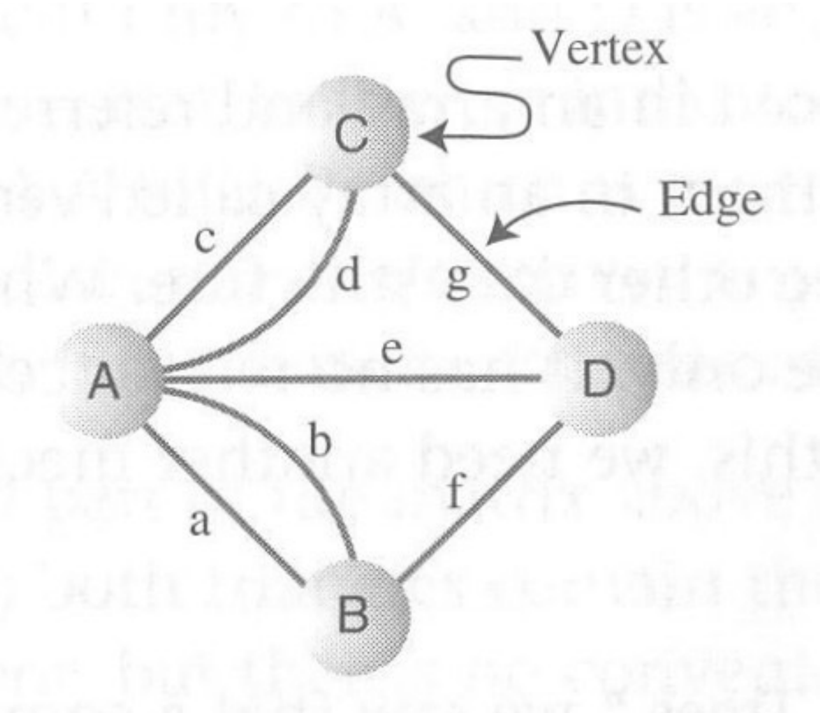
a) Map

Abstraction



Solution

- This problem inspired Leonhard Euler to invent graph theory during the 18th century.
- The trick was to represent the bridges in the form of a graph.



b) Graph

Konigsberg → Kaliningrad

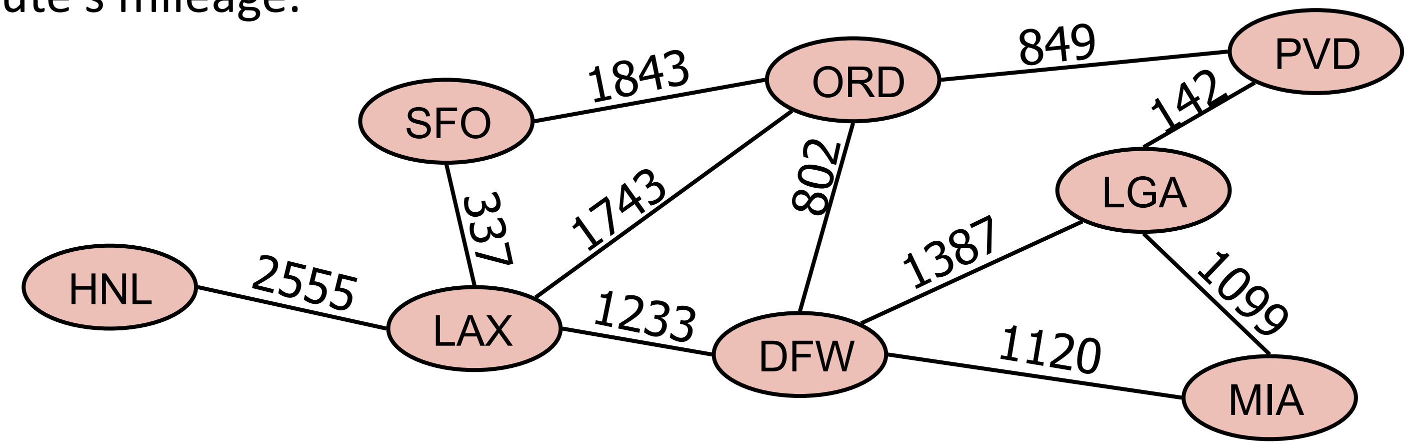
- Birthplace of Christian Goldbach (1690)
- Home of Immanuel Kant (1724)
- Bridges were used by Euler to develop graph theory (1736)
- Birthplace of David Hilbert (1862)

Graphs

- Nodes are called *vertices.*
- Edges link vertices.
- Vertices are labeled in some way, often with letters.
- Each edge has two vertices at its ends.
- The graph represents which vertices are connected to each other.

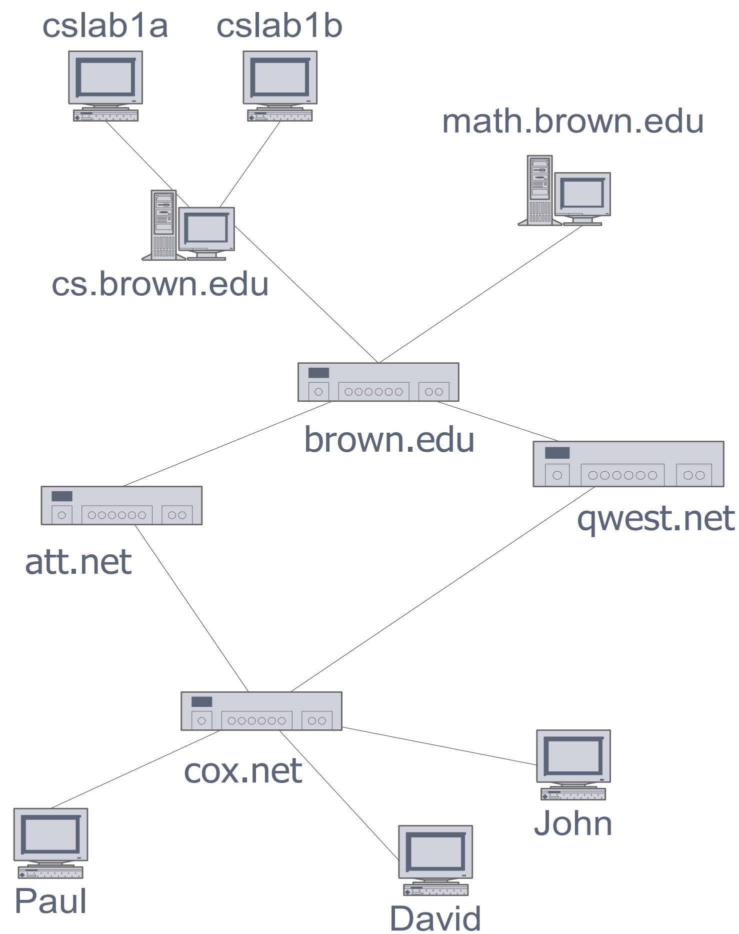
Graphs (cont.)

- 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 route's mileage.



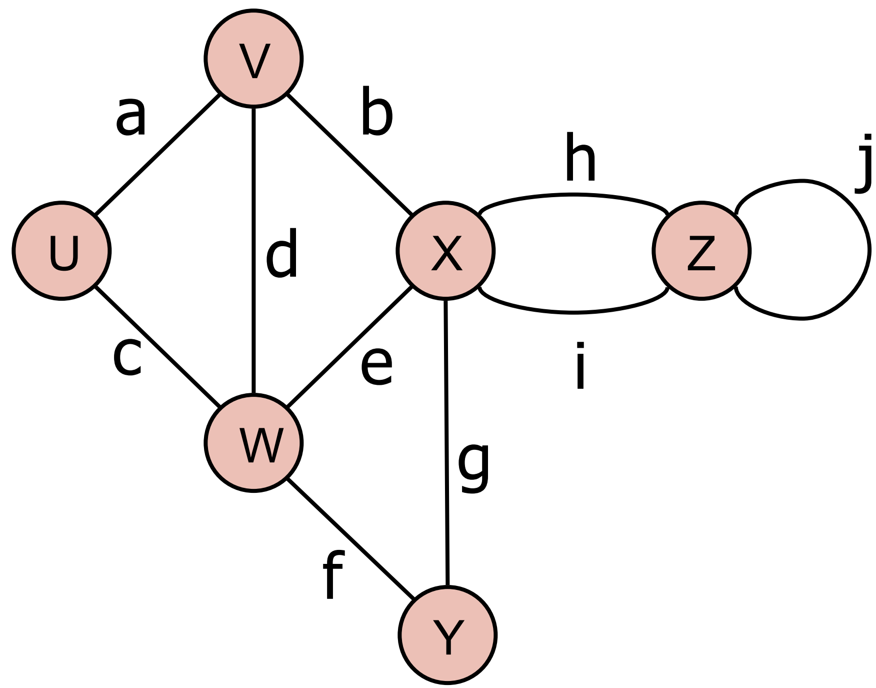
Applications

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



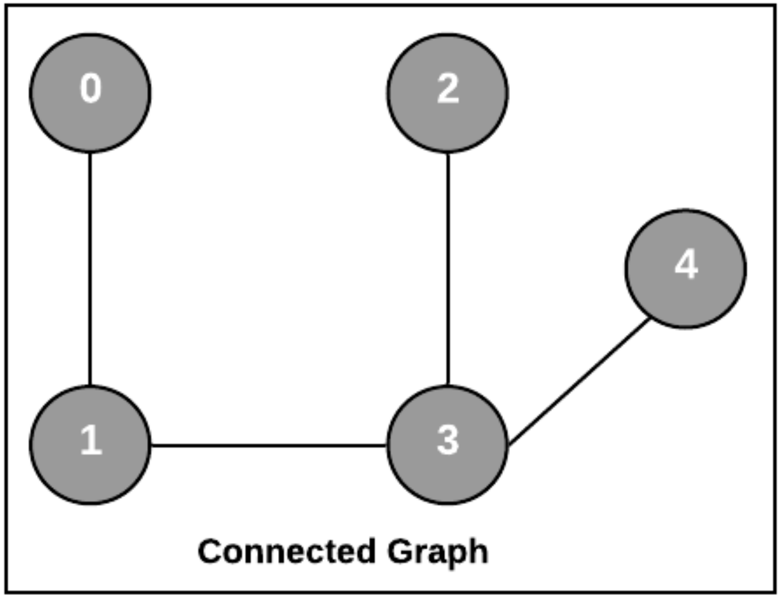
Terminology

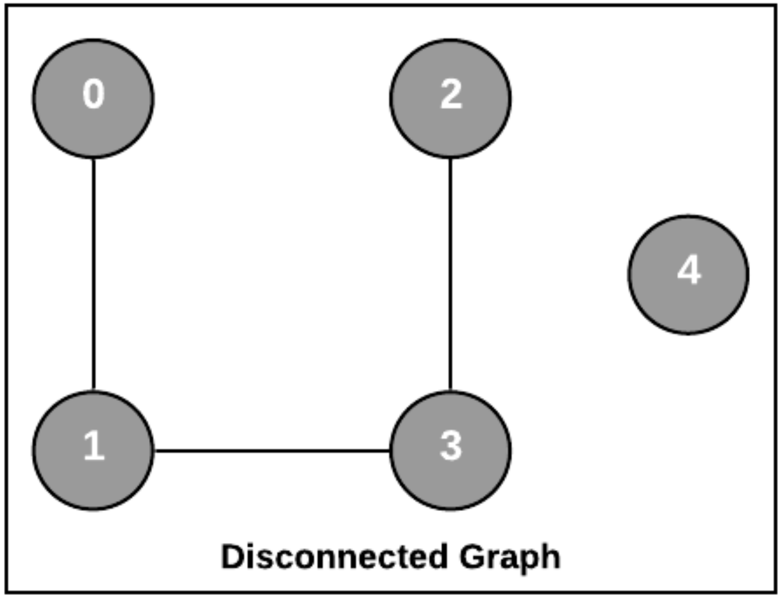
- End vertices (or endpoints) of an edge
  - *U* and *V* are the endpoints of *a*
- Edges incident on a vertex
  - *a*, *d*, and b are incident on *V*
- Adjacent vertices
  - *U* and *V* are adjacent
- Degree of a vertex
  - *X* has degree *5*
- Parallel edges
  - *h* and *i* are parallel edges
- Self-loop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



Connectivity in graphs

- **Connected graphs**: A graph is said to be connected if there is at least one path from every vertex to every other vertex.
- **Disconnected graph**

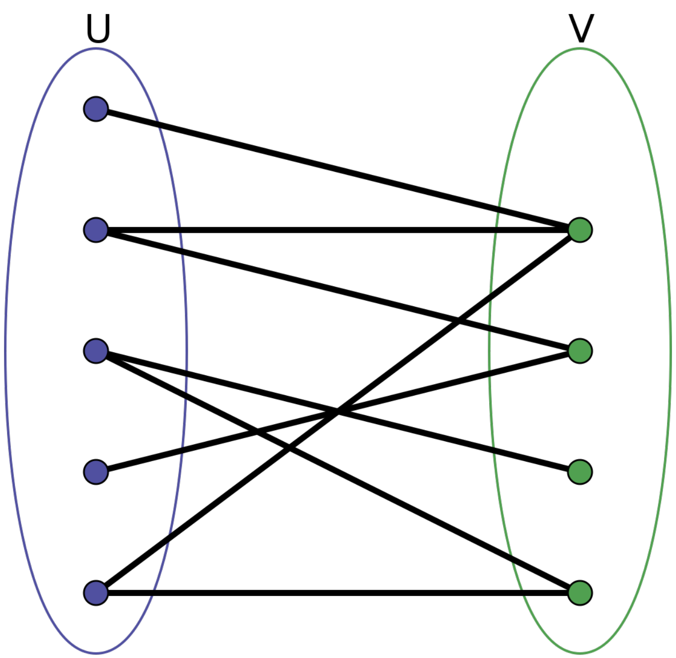


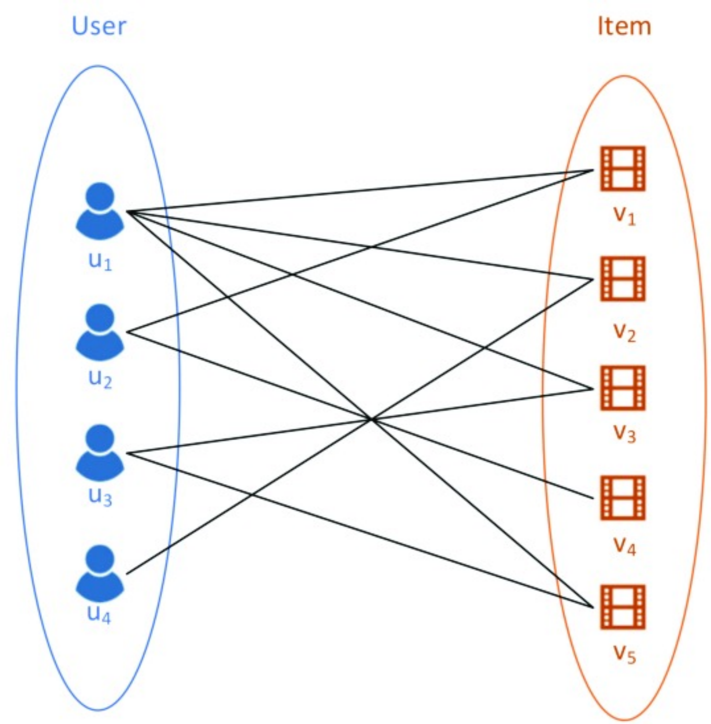


The algorithms we will consider all assume a connected graph.

Connectivity in graphs (cont.)

Bi-partite graphs: a bipartite graph is a graph whose vertices can be divided into two disjoint and independent sets U and V, that is every edge connects a vertex in U to one in V.

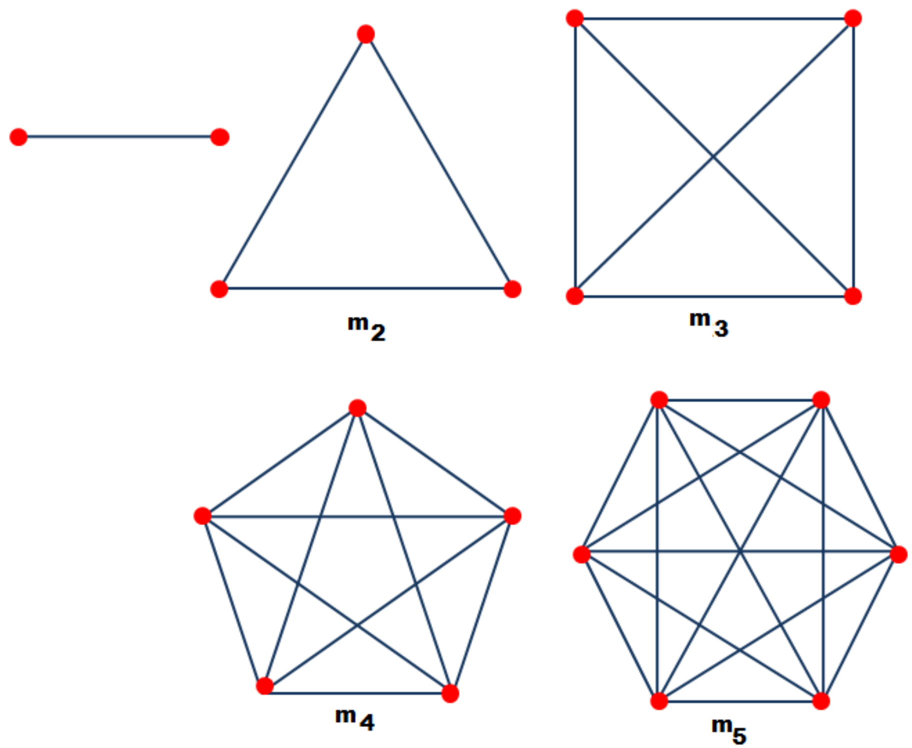




Connectivity in graphs (cont.)

Complete graphs

- In a complete graph every vertex has an edge to all other.
- The complete graph with *n* graph vertices is denoted *mn*.
- The complete graph on *n* vertices has *n(n−1)/2*.



Terminology

Path

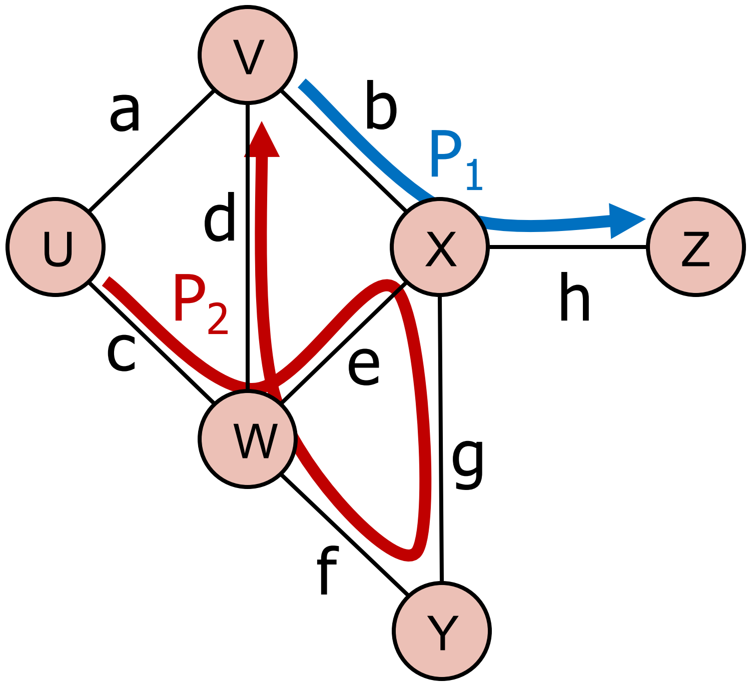
- The sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge is preceded and followed by its endpoints

Simple path

- A path such that all its vertices and edges are distinct

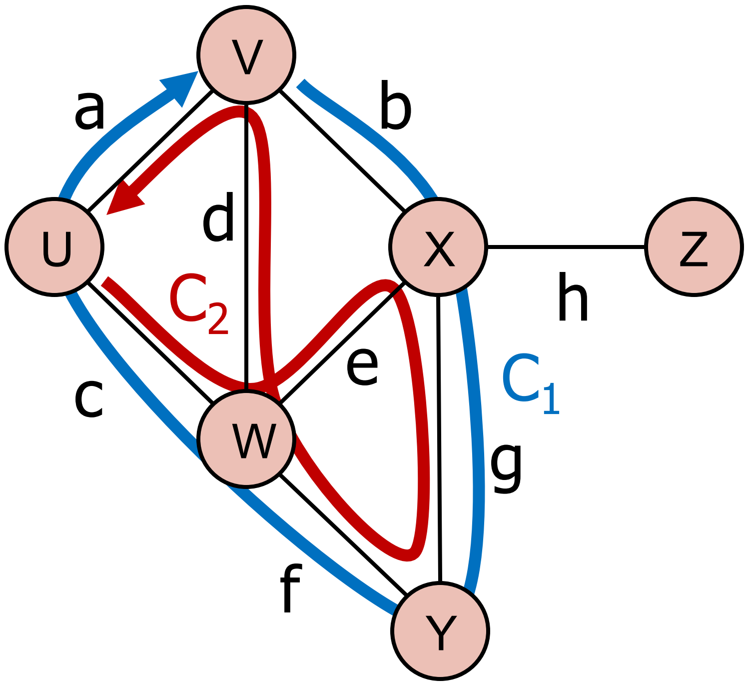
Examples

- 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



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

Directed and Weighted

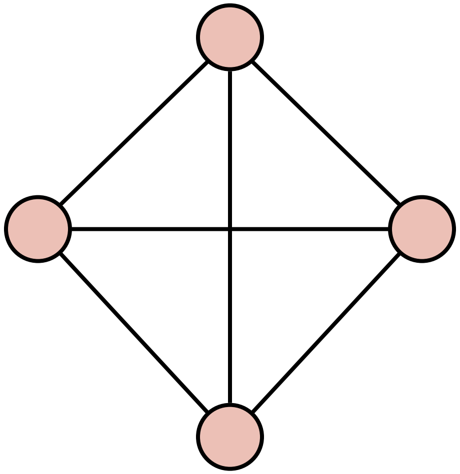
- A non-directed graph means you don’t have to go in a particular direction. You can follow an edge in both directions.
- Graphs are often used to model situations where you can go in only one direction along an edge – like a one-way street.
- These graphs are called *directed,* and the allowed direction is shown as an arrowhead.
- In some graphs, edges are given a weight that represents factors such as the physical distance between two vertices or the cost/time taken to get from one vertex to another.

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 a degree at most $(n-1)$

Notation

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



Example

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

What is the bound for a directed graph?

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.

Graph ADT

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.

Representing Edges

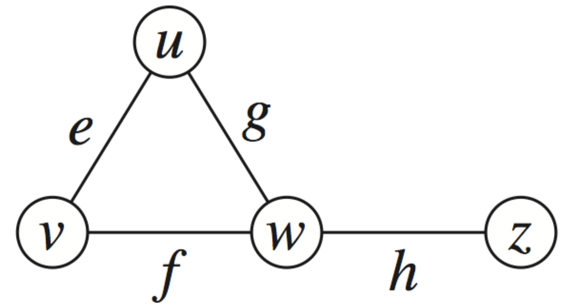
- In a binary tree, each node contains a reference to two child nodes.
- However, graphs have a more free-form organization where they have an arbitrary number of edges.
- We need a special data structure for representing connections between vertices.

Three commonly used methods are:

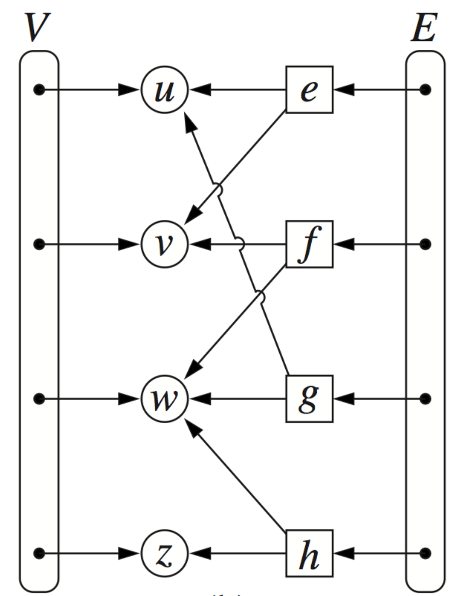
1. Edge list structure
2. Adjacency list structure
3. Adjacency matrix

Edge list structure

The ***edge list*** structure is possibly the simplest, though not the most efficient, representation of a graph *G*. All vertex objects are stored in an unordered list *V*, and all edge objects are stored in an unordered list *E*.



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



Performance of the Edge List Structure

Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v), outDegree(v), outgoingEdges(v)	$O(m)$
insertVertex(x), insertEdge(u, v, x), removeEdge(e)	$O(1)$
removeVertex(v)	$O(m)$

For a graph of n vertices and m edges.

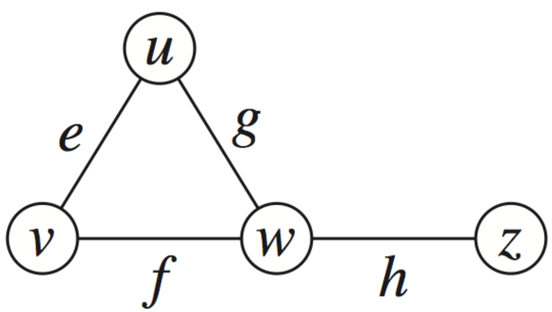
Adjacency List

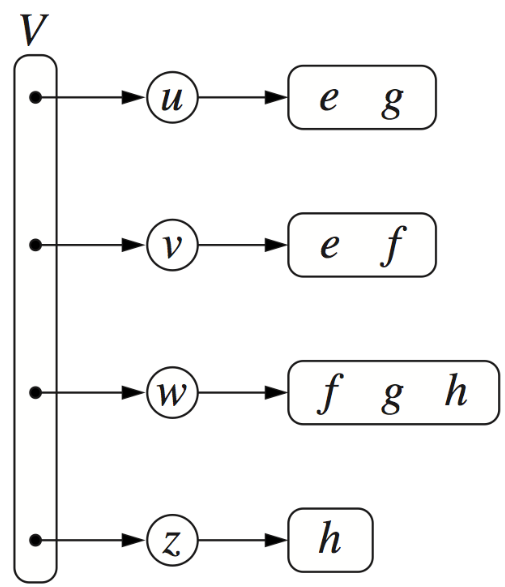
- An adjacency list uses a linked list structure to represent edges.
- There is a linked list for each vertices, which contains a list of all the other vertices it is linked to.
- The order in the linked list has no particular significance.
- For a very sparse graph (few edges between vertices), you can speed up runtime by using an adjacency list (because you don't need to examine any 0 entries).
- However, the algorithms involving the adjacency matrix are simpler, so we'll use this method.

Adjacency List Structure

The adjacency list structure for a graph adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex.

- 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





Performance of the Adjacency List Structure

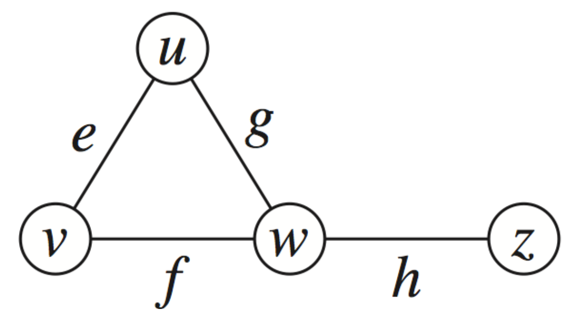
Method	Running Time
numVertices(), numEdges()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
getEdge(u, v)	$O(\min(\deg(u), \deg(v)))$
outDegree(v), inDegree(v)	$O(1)$
outgoingEdges(v), incomingEdges(v)	$O(\deg(v))$
insertVertex(x), insertEdge(u, v, x)	$O(1)$
removeEdge(e)	$O(1)$
removeVertex(v)	$O(\deg(v))$

Adjacency Matrix Structure

The adjacency matrix is a two-dimensional array in which the elements indicate whether an edge is present between two vertices.

If a graph has N vertices, then the adjacency matrix is an N x N array.

- 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 →	0		e	g	
v →	1	e		f	
w →	2	g	f		h
z →	3			h	

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

Questions