

# Graphs

Data Structures and Algorithms for Computational Linguistics III  
(ISCL-BA-07)

Çağrı Çöltekin

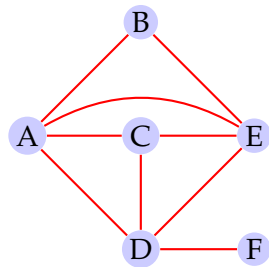
`ccoltekin@sfs.uni-tuebingen.de`

University of Tübingen  
Seminar für Sprachwissenschaft

Winter Semester 2021/22

# Introduction

- A graph is collection of **vertices** (**nodes**) connected pairwise by **edges** (**arcs**).
- A graph is a useful abstraction with many applications
- Most problems on graphs are challenging



# Example applications

## City map

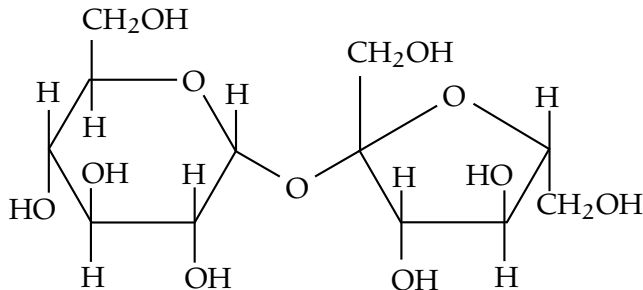
- City maps
- Chemical formulas
- Neural networks
- Artificial neural networks
- Electronic circuits
- Computer networks
- Infectious diseases
- Probability distributions
- Word semantics



# Example applications

## City map

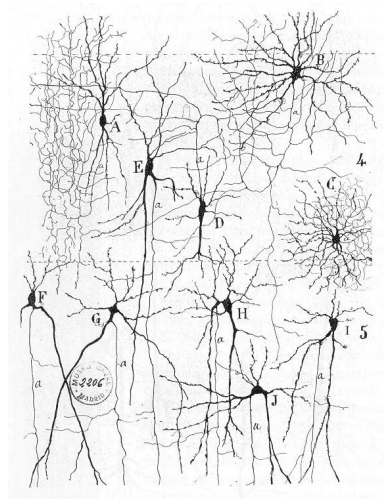
- City maps
- Chemical formulas
- Neural networks
- Artificial neural networks
- Electronic circuits
- Computer networks
- Infectious diseases
- Probability distributions
- Word semantics



# Example applications

## City map

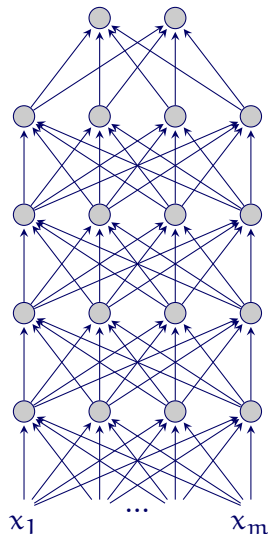
- City maps
- Chemical formulas
- **Neural networks**
- Artificial neural networks
- Electronic circuits
- Computer networks
- Infectious diseases
- Probability distributions
- Word semantics



# Example applications

## City map

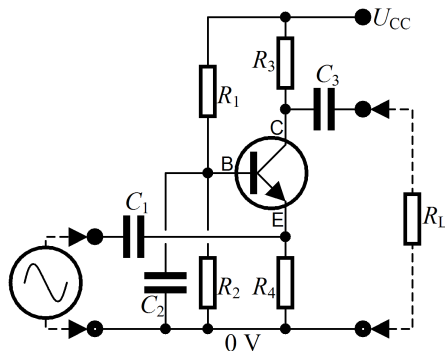
- City maps
- Chemical formulas
- Neural networks
- **Artificial neural networks**
- Electronic circuits
- Computer networks
- Infectious diseases
- Probability distributions
- Word semantics



# Example applications

## City map

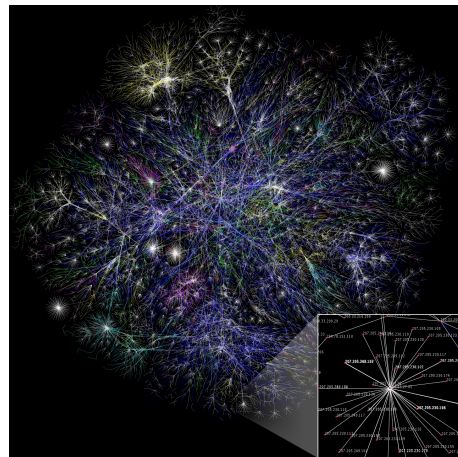
- City maps
- Chemical formulas
- Neural networks
- Artificial neural networks
- **Electronic circuits**
- Computer networks
- Infectious diseases
- Probability distributions
- Word semantics



# Example applications

## City map

- City maps
- Chemical formulas
- Neural networks
- Artificial neural networks
- Electronic circuits
- **Computer networks**
- Infectious diseases
- Probability distributions
- Word semantics

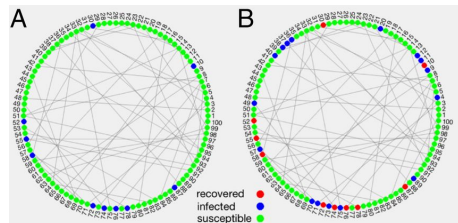




# Example applications

## City map

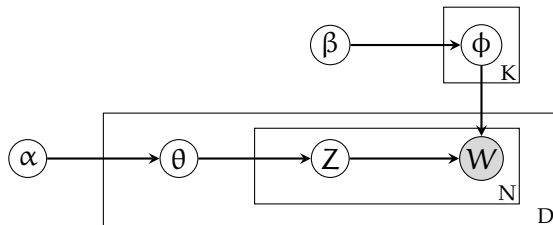
- City maps
- Chemical formulas
- Neural networks
- Artificial neural networks
- Electronic circuits
- Computer networks
- **Infectious diseases**
- Probability distributions
- Word semantics



# Example applications

## City map

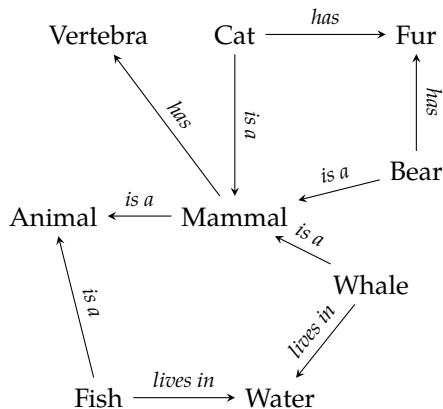
- City maps
- Chemical formulas
- Neural networks
- Artificial neural networks
- Electronic circuits
- Computer networks
- Infectious diseases
- **Probability distributions**
- Word semantics



# Example applications

## City map

- City maps
- Chemical formulas
- Neural networks
- Artificial neural networks
- Electronic circuits
- Computer networks
- Infectious diseases
- Probability distributions
- Word semantics



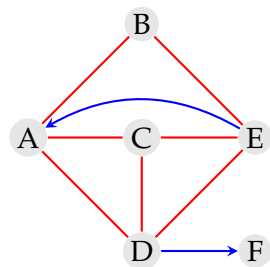
# Example applications

many more...

- Food web
- Course dependencies
- Social media
- Scheduling
- Games
- Academic networks
- Inheritance relations in object-oriented programming
- Flow charts
- Financial transactions
- World's languages
- PageRank algorithm
- ...

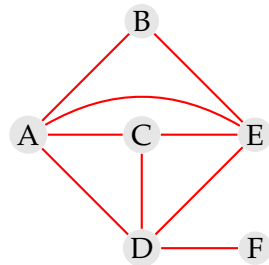
# Definition

- A (simple) *graph*  $G$  is a pair  $(V, E)$  where
  - $V$  is a set of *nodes* (or vertices),
  - $E \subseteq \{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$  is a set of ordered or unordered pairs
- A graph represent a set of objects (nodes) and the relations between them (edges)
- Edges in a graph can be either **directed**, or **undirected**
  - directed edges are 2-tuples, or *ordered pairs* (order is important)
  - undirected edges are unordered pairs, or pair sets (order is not important)



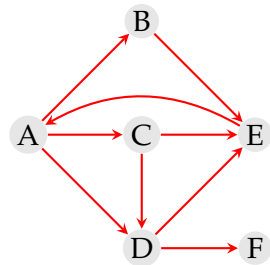
# Types of graphs

- An *undirected graph* is a graph with only undirected edges
  - Transportation (e.g., railway) networks
- A *directed graph* (digraph) is a graph with only directed edges
- A *mixed graph* contains both directed and undirected edges



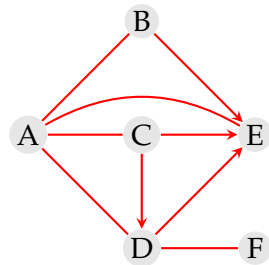
# Types of graphs

- An *undirected graph* is a graph with only undirected edges
  - Transportation (e.g., railway) networks
- A *directed graph* (digraph) is a graph with only directed edges
  - course dependencies
- A *mixed graph* contains both directed and undirected edges



# Types of graphs

- An *undirected graph* is a graph with only undirected edges
  - Transportation (e.g., railway) networks
- A *directed graph* (digraph) is a graph with only directed edges
  - course dependencies
- A *mixed graph* contains both directed and undirected edges
  - a city map



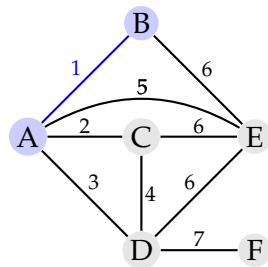


# More graphs types

- A graph is *simple* if there is only a single edge between two nodes (our earlier definition)
- If the edges of a graph has associated weights, it is called a *weighted graph*
- A *complete graph* contains edges from each node to every other node
- A *bipartite graph* has two disjoint sets of nodes, where edges are always across the sets
- A graph is called a *multi-graph* if there are multiple edges (with the same direction) between a pair of nodes
- A graph is called a *hyper-graph* if a single edge can link more than two nodes

# More definitions

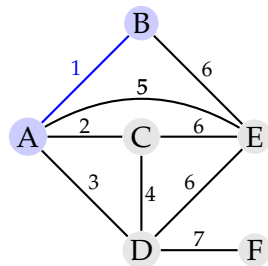
- Two nodes joined by an edge are called the *endpoints* of the edge
- An edge is called *incident* to a node if the node is one of its endpoints. Two nodes are *adjacent* (or they are neighbors) if they are incident to the same edge
- The *degree* (or valency) of a node is the number of its incident edges
- In a digraph *indegree* of a node is the number of incoming edges, and *outdegree* of a node is the number of outgoing edges



A and B are endpoints of edge 1

# More definitions

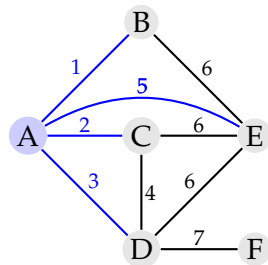
- Two nodes joined by an edge are called the *endpoints* of the edge
- An edge is called *incident* to a node if the node is one of its endpoints. Two nodes are *adjacent* (or they are neighbors) if they are incident to the same edge
- The *degree* (or valency) of a node is the number of its incident edges
- In a digraph *indegree* of a node is the number of incoming edges, and *outdegree* of a node is the number of outgoing edges



edge 1 is incident to A and B

# More definitions

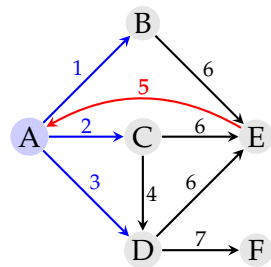
- Two nodes joined by an edge are called the *endpoints* of the edge
- An edge is called *incident* to a node if the node is one of its endpoints. Two nodes are *adjacent* (or they are neighbors) if they are incident to the same edge
- The *degree* (or valency) of a node is the number of its incident edges
- In a digraph *indegree* of a node is the number of incoming edges, and *outdegree* of a node is the number of outgoing edges



$$\deg(A) = 4$$

## More definitions

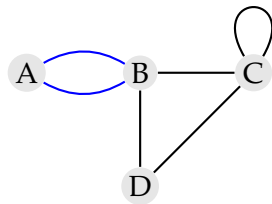
- Two nodes joined by an edge are called the *endpoints* of the edge
- An edge is called *incident* to a node if the node is one of its endpoints. Two nodes are *adjacent* (or they are neighbors) if they are incident to the same edge
- The *degree* (or valency) of a node is the number of its incident edges
- In a digraph *indegree* of a node is the number of incoming edges, and *outdegree* of a node is the number of outgoing edges



$\text{indeg}(A) = 1, \text{outdeg}(A) = 3$

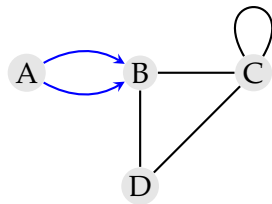
# More definitions

- Two edges are *parallel* if their both endpoints are the same
- For a directed graph parallel edges are ones with the same direction
- A self-loop is an edge from a node to itself
- A *path* is an sequence of alternating edges and nodes
- A *cycle* is a path that starts and ends at the same node
- A path or a cycle is a *simple* if every node on the path is visited only once



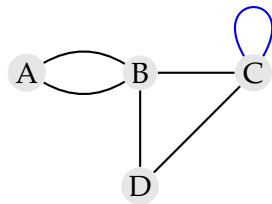
# More definitions

- Two edges are *parallel* if their both endpoints are the same
- For a directed graph parallel edges are ones with the same direction
- A self-loop is an edge from a node to itself
- A *path* is an sequence of alternating edges and nodes
- A *cycle* is a path that starts and ends at the same node
- A path or a cycle is a *simple* if every node on the path is visited only once



# More definitions

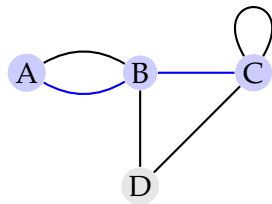
- Two edges are *parallel* if their both endpoints are the same
- For a directed graph parallel edges are ones with the same direction
- A self-loop is an edge from a node to itself
- A *path* is an sequence of alternating edges and nodes
- A *cycle* is a path that starts and ends at the same node
- A path or a cycle is a *simple* if every node on the path is visited only once





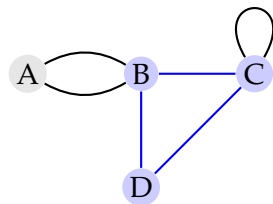
# More definitions

- Two edges are *parallel* if their both endpoints are the same
- For a directed graph parallel edges are ones with the same direction
- A self-loop is an edge from a node to itself
- A *path* is an sequence of alternating edges and nodes
- A *cycle* is a path that starts and ends at the same node
- A path or a cycle is a *simple* if every node on the path is visited only once



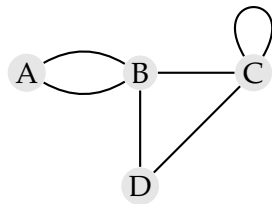
# More definitions

- Two edges are *parallel* if their both endpoints are the same
- For a directed graph parallel edges are ones with the same direction
- A self-loop is an edge from a node to itself
- A *path* is an sequence of alternating edges and nodes
- A *cycle* is a path that starts and ends at the same node
- A path or a cycle is a *simple* if every node on the path is visited only once



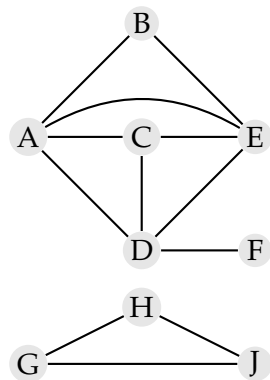
## More definitions

- Two edges are *parallel* if their both endpoints are the same
- For a directed graph parallel edges are ones with the same direction
- A self-loop is an edge from a node to itself
- A *path* is an sequence of alternating edges and nodes
- A *cycle* is a path that starts and ends at the same node
- A path or a cycle is a *simple* if every node on the path is visited only once



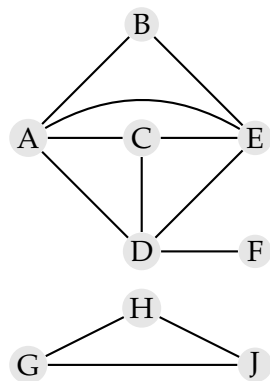
## More definitions

- A node  $X$  is *reachable* from another ( $Y$ ) if there is a (directed) path from  $Y$  to  $X$
- A graph is *connected* if all nodes are reachable from each other
- A directed graph is *strongly connected* if all nodes are reachable from each other
- A *subgraph* a graph formed by a subset of nodes and edges of a graph
- If a graph is not connected, the maximally connected subgraphs are called the connected components



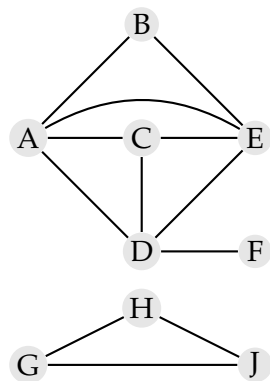
## More definitions

- A node  $X$  is *reachable* from another ( $Y$ ) if there is a (directed) path from  $Y$  to  $X$
- A graph is *connected* if all nodes are reachable from each other
- A directed graph is *strongly connected* if all nodes are reachable from each other
- A *subgraph* a graph formed by a subset of nodes and edges of a graph
- If a graph is not connected, the maximally connected subgraphs are called the connected components



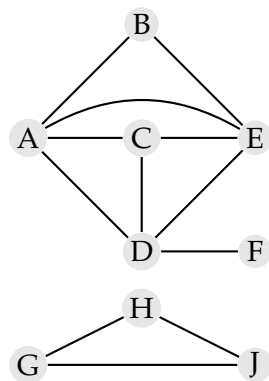
## More definitions

- A node  $X$  is *reachable* from another ( $Y$ ) if there is a (directed) path from  $Y$  to  $X$
- A graph is *connected* if all nodes are reachable from each other
- A directed graph is *strongly connected* if all nodes are reachable from each other
- A *subgraph* a graph formed by a subset of nodes and edges of a graph
- If a graph is not connected, the maximally connected subgraphs are called the connected components



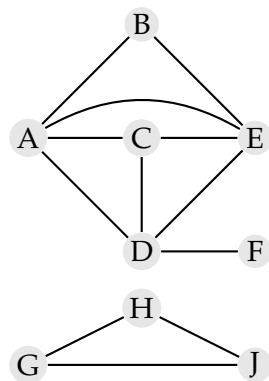
# More definitions

- A node  $X$  is *reachable* from another ( $Y$ ) if there is a (directed) path from  $Y$  to  $X$
- A graph is *connected* if all nodes are reachable from each other
- A directed graph is *strongly connected* if all nodes are reachable from each other
- A *subgraph* a graph formed by a subset of nodes and edges of a graph
- If a graph is not connected, the maximally connected subgraphs are called the connected components



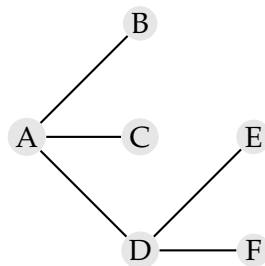
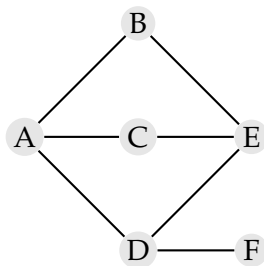
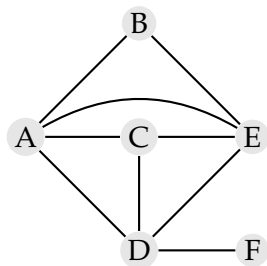
# More definitions

- A node  $X$  is *reachable* from another ( $Y$ ) if there is a (directed) path from  $Y$  to  $X$
- A graph is *connected* if all nodes are reachable from each other
- A directed graph is *strongly connected* if all nodes are reachable from each other
- A *subgraph* a graph formed by a subset of nodes and edges of a graph
- If a graph is not connected, the maximally connected subgraphs are called the **connected components**



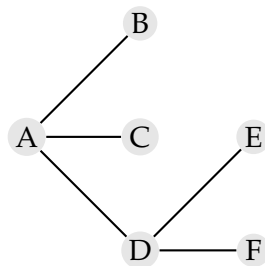
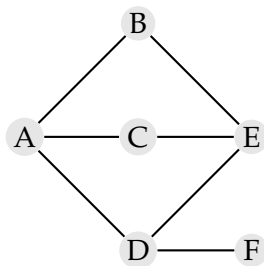
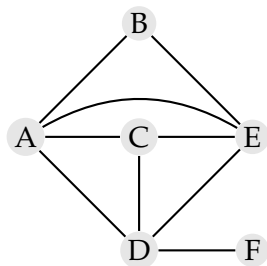


## More definitions



- A *spanning subgraph* of a graph is a subgraph that includes all nodes of the graph
- A *tree* is a connected graph without cycles
- A *spanning tree* is a spanning subgraph which is a tree
- A *forest* is a disconnected acyclic graph

## More definitions



- A *spanning subgraph* of a graph is a subgraph that includes all nodes of the graph
- A *tree* is a connected graph without cycles
- A *spanning tree* is a spanning subgraph which is a tree
- A *forest* is a disconnected acyclic graph

# Some properties

## sum of degrees

- For an undirected graph with  $m$  edges and set of nodes  $V$

$$\sum_{v \in V} \deg(v) = 2m$$

- All edges are counted twice for each node they are incident to
- The total contribution of each node is twice its degree
- For a directed graph with  $m$  edges and set of nodes  $V$

$$\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = m$$

# Some properties

relation between the number of edges and nodes

- For a simple undirected graph with  $n$  nodes and  $m$  edges

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

- If the graph is simple
  - there are no parallel edges
  - there are no self loops
  - the maximum degree of a node is  $n-1$
- Putting this together with the previous property

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

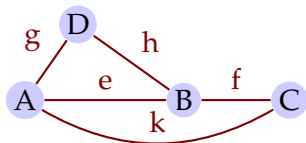
- For a directed graph with  $n$  nodes and  $m$  edges

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

# The graph ADT

- A graph is a collection of nodes and edges
- Basic operations include
  - `add_node(v)` add a new node
  - `remove_node(v)` remove an existing node
  - `adjacent(u,v)` return true if the nodes are adjacent (for a digraph true only if there is a directed link from `u` to `v`)
  - `neighbors(v)` enumerate the neighbors of the node (for a digraph we list the nodes reachable through outgoing edges by default)
  - `remove_edge(u,v)` remove an existing edge
  - `add_edge(u,v)` add a new edge
  - `nodes()` enumerate the nodes in the graph
  - `edges()` enumerate the edges in the graph

# Edge list



$$e = (A, B)$$

$$f = (B, C)$$

$$g = (A, D)$$

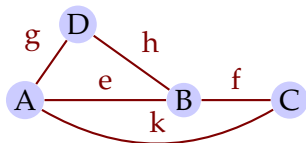
$$h = (D, B)$$

$$k = (A, C)$$

- We keep a simple a simple list of edges (and possibly nodes)
- Simple structure, but not very efficient ( $n$  nodes,  $m$  edges):

$$\text{add\_edge}(v)$$

# Edge list



$$e = (A, B)$$

$$f = (B, C)$$

$$g = (A, D)$$

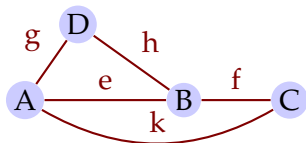
$$h = (D, B)$$

$$k = (A, C)$$

- We keep a simple a simple list of edges (and possibly nodes)
- Simple structure, but not very efficient ( $n$  nodes,  $m$  edges):

`add_edge(v)`    $O(1)$   
`remove_edge(v)`

# Edge list



$$e = (A, B)$$

$$f = (B, C)$$

$$g = (A, D)$$

$$h = (D, B)$$

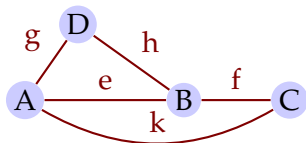
$$k = (A, C)$$

- We keep a simple a simple list of edges (and possibly nodes)
- Simple structure, but not very efficient ( $n$  nodes,  $m$  edges):

$\text{add\_edge}(v) \quad O(1)$   
 $\text{remove\_edge}(v) \quad O(m)$   
 $\text{remove\_node}(v)$



# Edge list



$$e = (A, B)$$

$$f = (B, C)$$

$$g = (A, D)$$

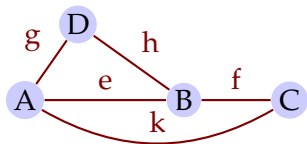
$$h = (D, B)$$

$$k = (A, C)$$

- We keep a simple a simple list of edges (and possibly nodes)
- Simple structure, but not very efficient ( $n$  nodes,  $m$  edges):

$\text{add\_edge}(v)$   $O(1)$   
 $\text{remove\_edge}(v)$   $O(m)$   
 $\text{remove\_node}(v)$   $O(m)$   
 $\text{adjacent}(u, v)$

# Edge list



$e = (A,B)$

$f = (B,C)$

$g = (A,D)$

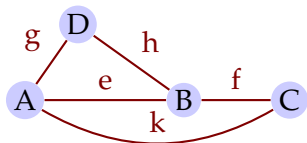
$h = (D,B)$

$k = (A,C)$

- We keep a simple a simple list of edges (and possibly nodes)
- Simple structure, but not very efficient ( $n$  nodes,  $m$  edges):

$\text{add\_edge}(v)$   $O(1)$   
 $\text{remove\_edge}(v)$   $O(m)$   
 $\text{remove\_node}(v)$   $O(m)$   
 $\text{adjacent}(u,v)$   $O(m)$   
 $\text{neighbors}(v)$

# Edge list



$e = (A, B)$

$f = (B, C)$

$g = (A, D)$

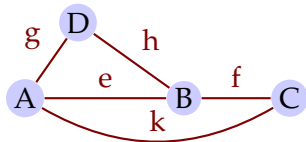
$h = (D, B)$

$k = (A, C)$

- We keep a simple a simple list of edges (and possibly nodes)
- Simple structure, but not very efficient ( $n$  nodes,  $m$  edges):

$\text{add\_edge}(v)$   $O(1)$   
 $\text{remove\_edge}(v)$   $O(m)$   
 $\text{remove\_node}(v)$   $O(m)$   
 $\text{adjacent}(u, v)$   $O(m)$   
 $\text{neighbors}(v)$   $O(m)$

# Adjacency list



A — g e k

D — g h

B — e h f

C — f k

*nodes*

e = (A,B)

f = (B,C)

g = (A,D)

h = (D,B)

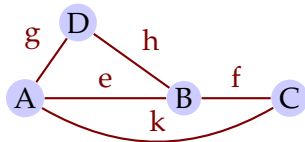
k = (A,C)

*edges*

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

`add_node(v)`

# Adjacency list



A — g e k

D — g h

B — e h f

C — f k

*nodes*

e = (A,B)

f = (B,C)

g = (A,D)

h = (D,B)

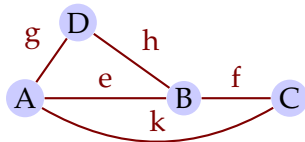
k = (A,C)

*edges*

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

add\_node(v)  $O(1)$   
remove\_node(v)

# Adjacency list



A — g e k

D — g h

B — e h f

C — f k

*nodes*

e = (A,B)

f = (B,C)

g = (A,D)

h = (D,B)

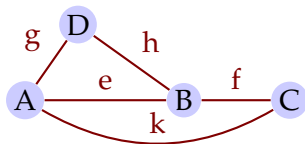
k = (A,C)

*edges*

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

add\_node(v)     $O(1)$   
 remove\_node(v)     $O(\deg(v))$   
 adjacent(u,v)

# Adjacency list



A — g e k

D — g h

B — e h f

C — f k

*nodes*

e = (A,B)

f = (B,C)

g = (A,D)

h = (D,B)

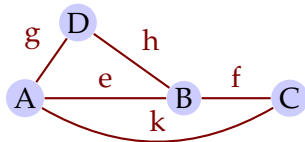
k = (A,C)

*edges*

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

add\_node( $v$ )     $O(1)$   
 remove\_node( $v$ )     $O(\deg(v))$   
 adjacent( $u, v$ )     $O(\min(\deg(u), \deg(v)))$   
 neighbors( $v$ )

# Adjacency list



A — g e k

D — g h

B — e h f

C — f k

*nodes*

e = (A,B)

f = (B,C)

g = (A,D)

h = (D,B)

k = (A,C)

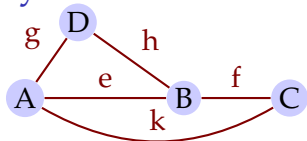
*edges*

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

add\_node( $v$ )     $O(1)$   
 remove\_node( $v$ )     $O(\deg(v))$   
 adjacent( $u, v$ )     $O(\min(\deg(u), \deg(v)))$   
 neighbors( $v$ )     $O(\deg(v))$



# Adjacency matrix

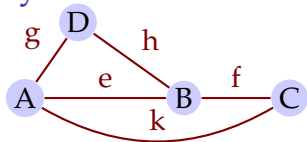


	A	B	C	D
A		e	k	g
B			f	h
C				
D				

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

`add_node(v)`

# Adjacency matrix

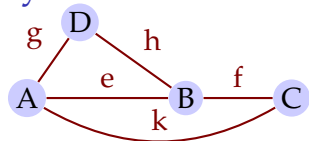


	A	B	C	D
A		e	k	g
B			f	h
C				
D				

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

`add_node(v)`  $O(n)$   
`remove_node(v)`

# Adjacency matrix

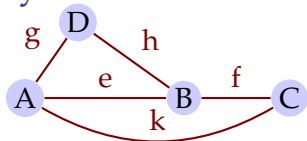


	A	B	C	D
A		e	k	g
B			f	h
C				
D				

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

`add_node(v)`    $O(n)$   
`remove_node(v)`    $O(n)$   
`adjacent(u,v)`

# Adjacency matrix

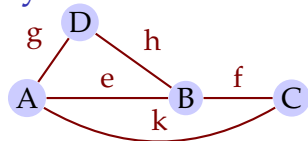


	A	B	C	D
A		e	k	g
B			f	h
C				
D				

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

`add_node(v)`     $O(n)$   
`remove_node(v)`     $O(n)$   
`adjacent(u,v)`     $O(1)$   
`neighbors(v)`

# Adjacency matrix



	A	B	C	D
A		e	k	g
B			f	h
C				
D				

- We keep simple lists for nodes and edges
- Very simple structure, but not very efficient:

`add_node(v)`  $O(n)$   
`remove_node(v)`  $O(n)$   
`adjacent(u,v)`  $O(1)$   
`neighbors(v)`  $O(n)$

# Interesting problems on graphs

- Is there a (directed) path between two nodes?
- What is the shortest path between two nodes?
- Is there a cycle in the graph?
- Is there a cycle that uses each edge exactly once? (Eulerian path)
- Is there a cycle that uses each node exactly once? (Hamiltonian path)
- Are all nodes of the graph connected?
- Is there a node that breaks the connectivity if removed?
- Is the graph planar: can it be drawn without crossing edges?
- Are two graphs isomorphic (have the same structure)?
- What is the importance of a web page, based on the links pointing to it?

# Summary

- Graphs are data structures with many applications
- Reading on graphs: Goodrich, Tamassia, and Goldwasser (2013, chapter 14),

Next:

- Graph traversals
- Reading: Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 14)

# Acknowledgments, credits, references

- The map on slide 2 is from OpenStreetMap, The other images are from Wikipedia, except the infectious disease graph which comes from Thurner et al. (2020).



Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013). *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN: 9781118476734.









