Graphs Introduction

Definitions

Structure

Properties

Graphs Definition G = (V, E)

- V = **set** of **nodes**

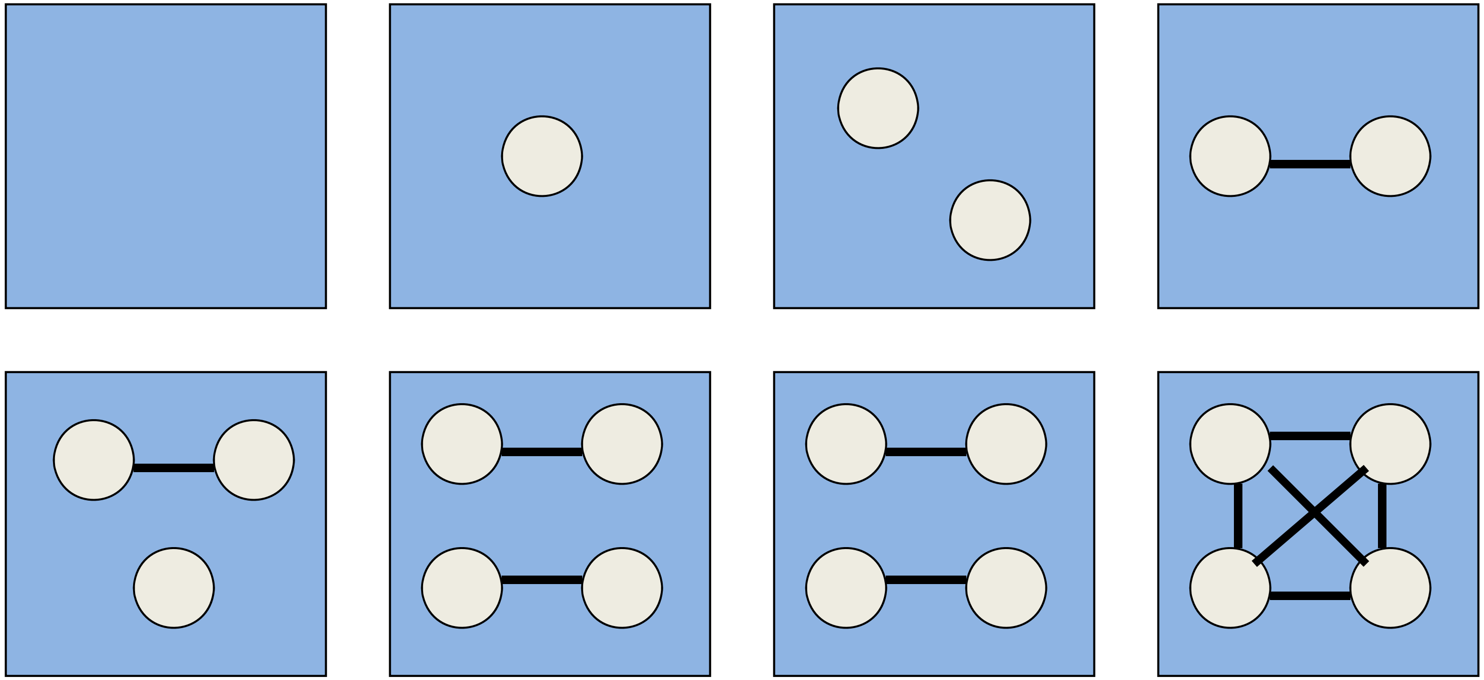
set → unique, unordered

- E = **set** of **edges** (v,w) v,w nodes in V
- **Degree of a node = number of incident edges**
  - **Directed graph – in-degree / out-degree**
- **Edges** may be
  - **Undirected** v ←→ u – **undirected graph**
  - **Directed** v → u – **directed graph**
  - An edge connects 2 nodes – represents a **relationship**
  - An edge may be **weighted** – i.e. have a value attribute
- **A graph may be empty; contain nodes only; contain nodes & edges**
- **Remember this definition G = (V, E)!!!**

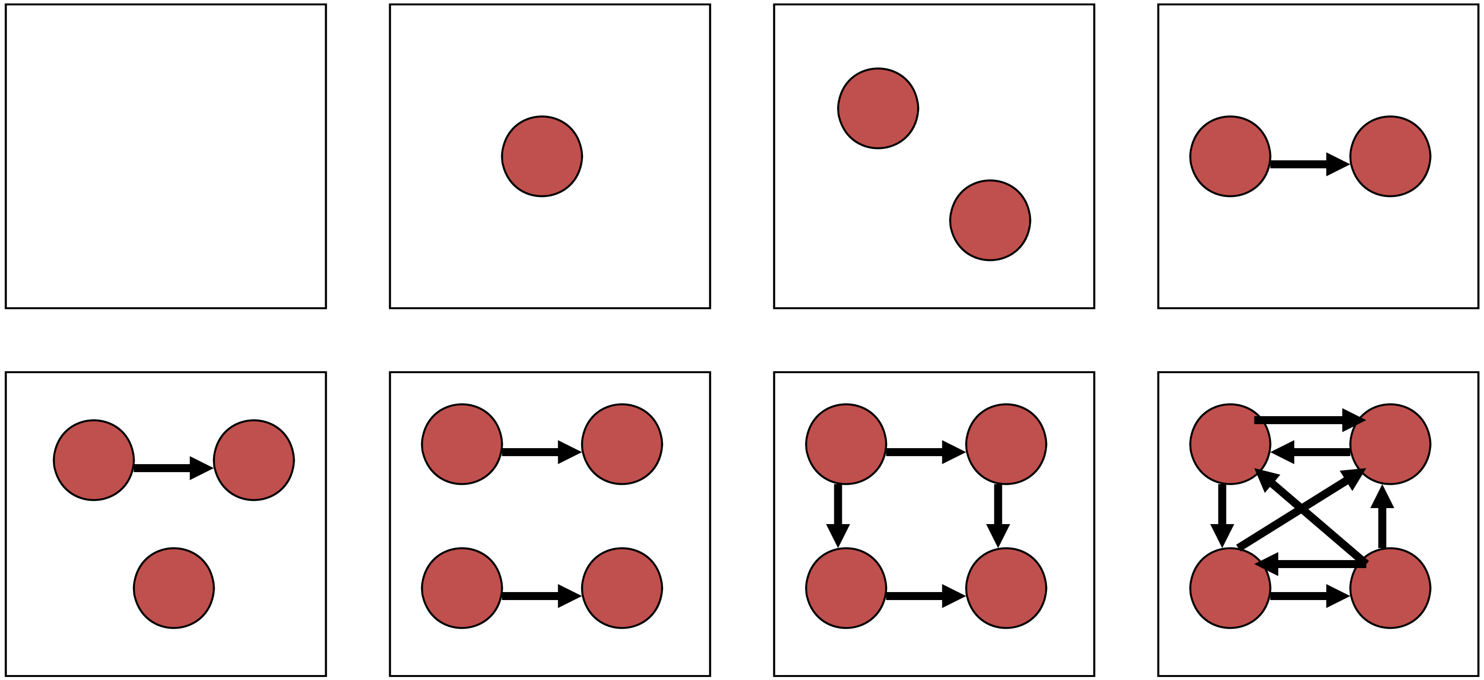
Meaning & Use

- A graph is used to represent arbitrary relationships among data objects
- e.g. undirected graphs
  - communications network
  - transport network (road, rail, air, sea) with costs/distances
  - (travelling salesman problem)
- e.g. directed graphs (digraph)
  - flow of control in computer programs
  - University course planning (dependency graph)
  - state transition diagrams

Examples: Undirected Graph

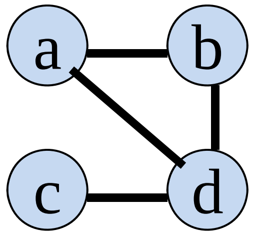


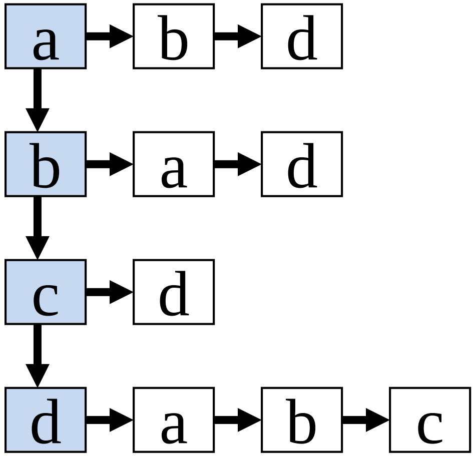
Examples: Directed Graph



Implementations: undirected

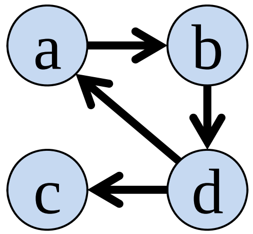
- Adjacency list $|V| + |E|$
  - Weights **(costs may be included in the edge list)**

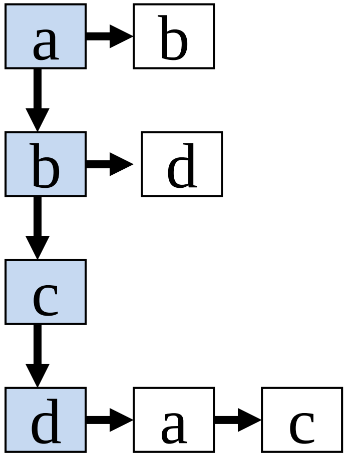




Implementations: directed

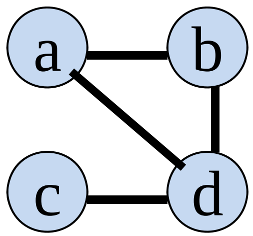
- Adjacency list |V| + |E|
  - Weights **(costs may be included in the edge list)**





Implementations: undirected

- Adjacency matrix |V| * |V|
  - 4 edges - implementation 8 directed edges

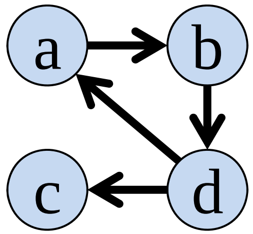


	a	b	c	d
a	0	1	0	1
b	1	0	0	1
c	0	0	0	1
d	1	1	1	0

  - **Symmetrical about the left diagonal**

Implementations: directed

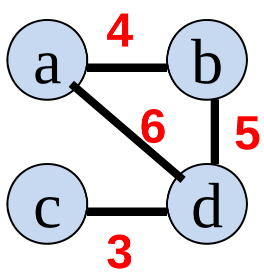
- Adjacency matrix |V| * |V|
  - 4 edges - implementation 4 directed edges



	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Implementations: undirected

- A Cost matrix
  - Weighted graphs

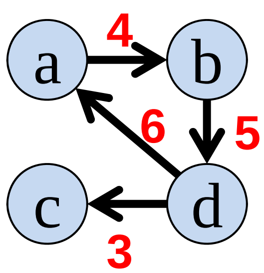


	a	b	c	d
a	0	4	0	6
b	4	0	0	5
c	0	0	0	3
d	6	5	3	0

  - **Symmetrical about the left diagonal**

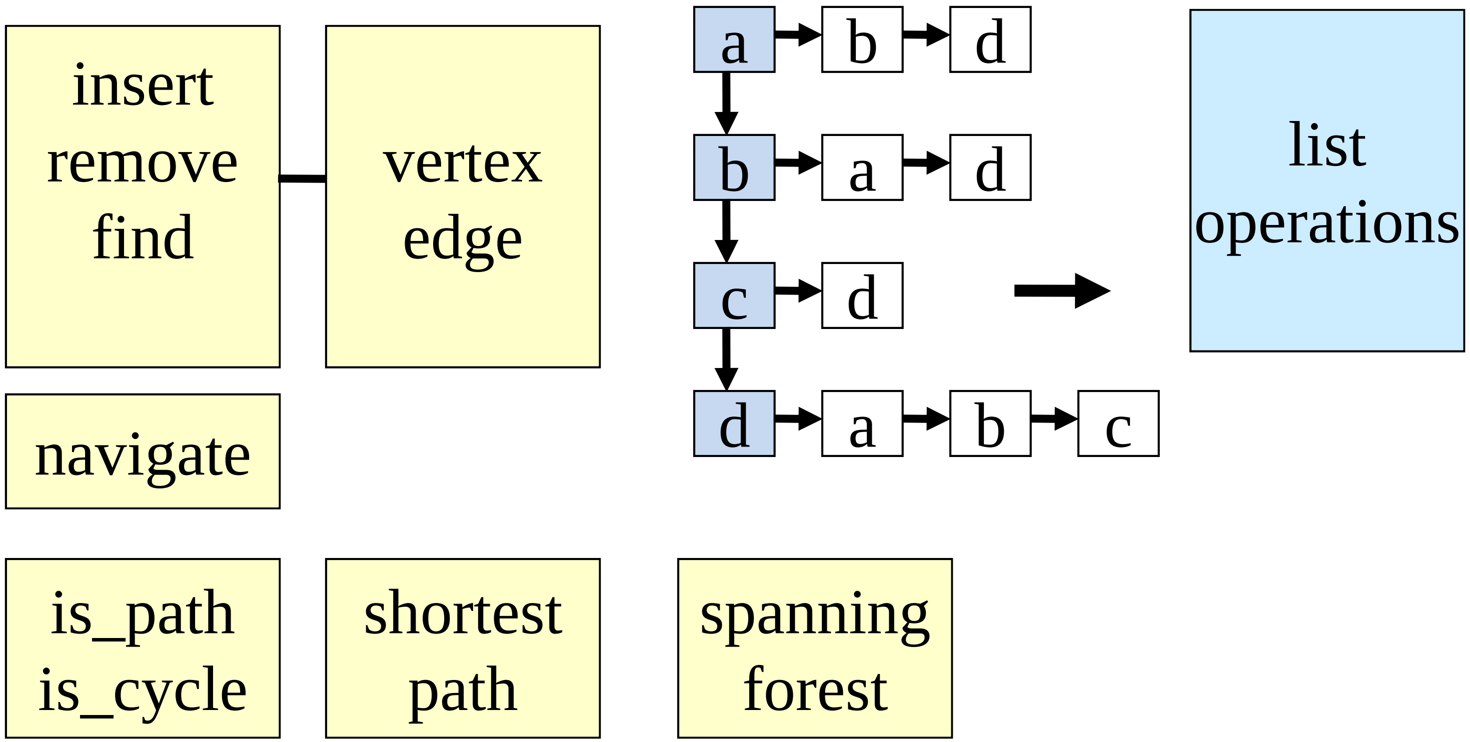
Implementations: directed

- A Cost matrix
  - Weighted graphs



	a	b	c	d
a	0	**4**	0	0
b	0	0	0	**5**
c	0	0	0	0
d	**6**	0	**3**	0

Implementation operations



Implementation operations

- add **node** - if node exists error else add node
- add **edge** – (v, w) – nodes v & w must exist else error

- find **node** – found or not found
- find **edge** – found or not found

- remove **node** – the node must exist else error

  all incident edges must also be removed
- remove **edge** – the edge must exist else error

- count **nodes /** count **edges**

Terminology

- Vertex **(source / sink)**
- Edge **(bridge)**
- Weight **(positive/negative)**
- Degree **(in / out)**
- Directed
- Undirected
- Path
- Simple path
- Cycle
- Simple cycle
- Subgraph
- Induced subgraph
- Complete graph
- Connected graph
- Connected Components
- Bipartite graph
- Spanning tree
- Spanning forest
- Shortest path tree
- DAG

Properties

- **A graph with V vertices has at most V(V-1)/2 edges**
- **A digraph with V vertices has at most V(V-1) edges**
- **Path** – a sequence of adjacent vertices
- **Simple path** – edges and vertices are distinct
- **Simple cycle** – simple path where 1st/final vertices same Graph (3 vertices) / digraph (2 vertices)
- **Connected graph** – path from every v to every other w
- An acyclic connected graph is called a TREE
  - **Spanning tree / spanning forest**
  - Note the difference between a directed acyclic graph (DAG) and a tree

Issues

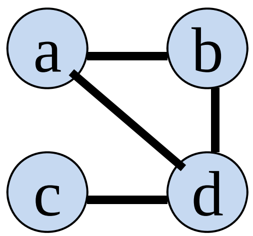
- Finding **paths** **(shortest A2B)** and **cycles**
- Finding **connectivity**
- **Separability**
  - Edge separable (**bridge** edge)
  - Vertex separable (**articulation** point)
- **Biconnectivity** (k-edge-connected)
  - Every pair of vertices connected by 2 disjoint paths
  - No articulation points

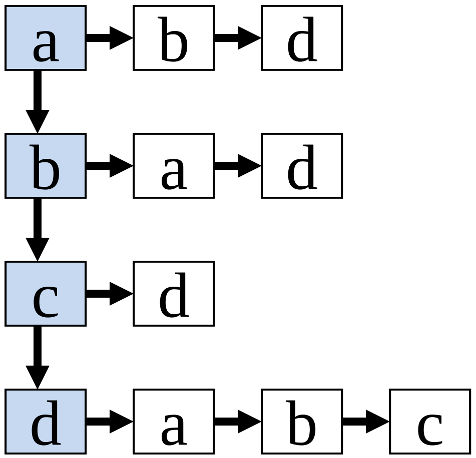
Algorithms

- Is_path(a, b)
- Is_cycle(G)
- Is_connected(G)
- Is_strongly_connected(DG) **Directed Graph**
- Transitive closure (G) **Warshall**
- Minimal spanning tree (MST) **Prim / Kruskal**
- Single-source shortest paths **Dijkstra**
- All-pairs shortest paths **Floyd**
- Topological sort (DAG) **Directed Acyclic Graph**
- Depth / breadth first search

Traversing a graph

- Mark each node when visited
- **Depth-first** search: a → b → d → c
- **Breadth-first** search: a, one step (b, d), two steps c





Visiting Nodes – 2 methods

1. **Mark as visited** (sometimes recursive calls)
2. **Cut**
   - Divide the graph into **components (n nodes)**
   - Merge the components **by adding edges**
     - Independent components **(Kruskal) (using a PQ)**
   - Choose one **start node** (in a component)
   - **2 sets**: visited **(S)** and not visited **(V-S)**
   - Merge the components **by adding edges**
     - **Tree formation** **(Prim, Dijkstra)**

Depth-first search (dfs)

- Recursive search **(stack)**
- Depth-first numbering
  - Preorder **(order that processing starts)**
  - Postorder **(order that processing finishes)**
- Cost
  - O(|V|+|E|) **(adjacency list)**
  - $O(V^2)$ **(adjacency matrix)**

Depth-first search (dfs) **used for**

- Simple path
- Simple connectivity (dfs called once)
- Topological sort **(digraphs - DAGs)**
- Finding strongly connected components
- Cycle detection **(back edges)**
- Finding bridge edges
- Finding articulation points

Digraph **(Directed Graph)** Algorithms

dfs / **bfs**	$O(\|V\|+\|E\|)$	$O(V^2)$
Dijkstra	$O(V^2)$	(cheaper variants exist)
Floyd	$O(V^3)$	(cheaper variants exist)
Warshall	$O(V^3)$	

- Alternative dfs on each node $O(V^2)$

- Topological sort (DAG) - dfs $O(|V|+|E|)$
- Strong components (dfs)
  - Kosaraju $O(|V|+|E|)$ or $O(V^2)$
  - Tarjan
  - Gabow (1999) **(we will not consider these 3 algorithms in this course)**

Graph Algorithms

- **dfs** / **bfs** $O(|V|+|E|)$ $O(V^2)$
- MST
  - **Prim** $O(V^2)$ (1961)
  - **Kruskal** $O(E \lg E)$ (1956)
- Other problems
  - Travelling Salesman (Hamiltonian path)
  - Königsberg Bridges (Euler)
  - Matching (bipartite graphs)

Summary

- Collection of nodes + relationships
- G = (V,E)   directed; undirected
- Edges may be weighted
- Implementation: adjacency list, matrix
- Digraphs: represent dependencies
- Graphs: represent networks