# Graphs

## Objectives

- to define graphs + graph ADT
- to describe different graph representations
- to define graph traversal methods
- to present and solve some graph-related problems

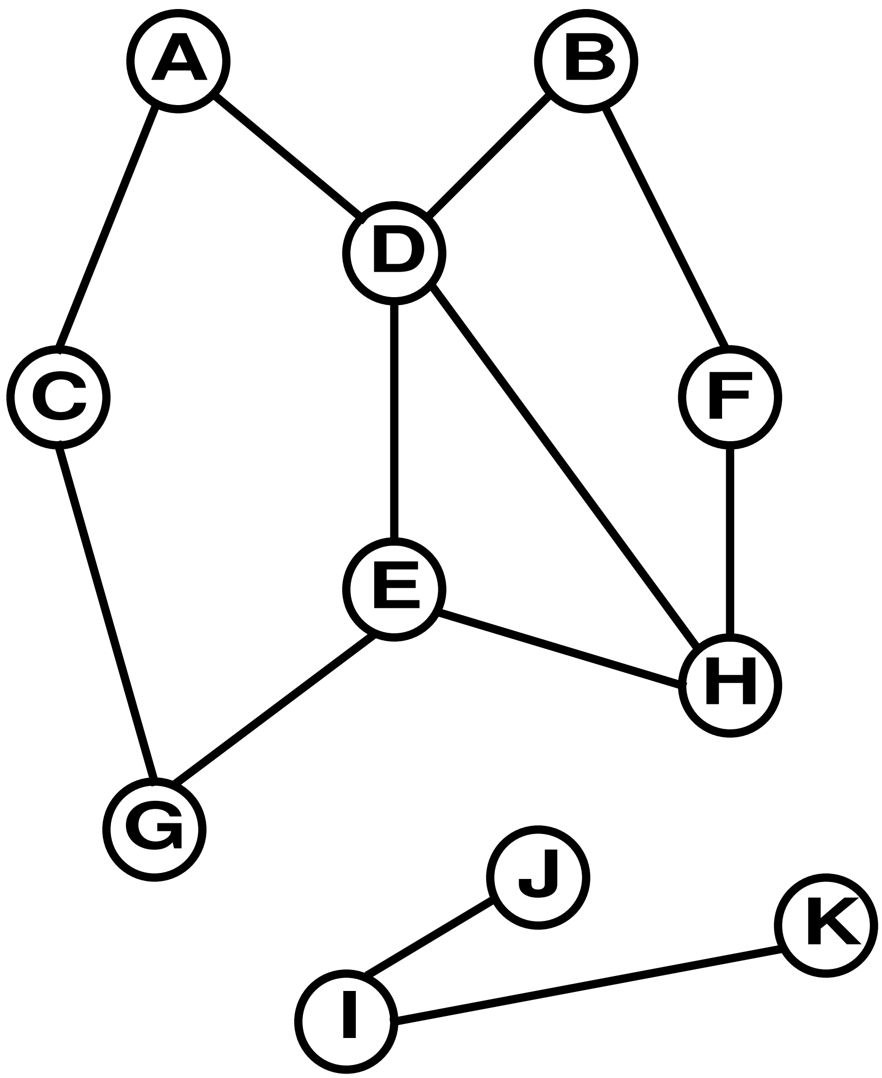
## Definition and Terminology

**Definition.** A graph G=(V,E) consists of a set V of vertices and a set E of edges connecting vertices

- **directed graphs** - have their edges directed
- **undirected graphs** - there is no direction associated with its edges
- **labeled graph** - there is a label associated with each node
- **weighted graph** - there are weights (numbers) associated with each vertex

- **adjacent vertices** - connected by an edge (neighbors, incident edge)
- **path** - a sequence of vertices such that any two consecutive vertices are connected by an edge
- **simple path** - all vertices on it are distinct
- **length of a path** - the number of edges it contains
- **cycle** - a path that connects some vertex to itself
- **acyclic graph** - a graph with no cycles

- **connected graph** - if for any two vertices there is at least one path connecting them



## Graph traversals

- **graph traversal** means the processing (visiting) the vertices of a graph in a systematic way, in some specific order (based on the topology of the graph)
- graph traversing algorithms typically begin with a start vertex and attempt to visit the vertices connected to it

**Problems** with graph traversals:

- not all the vertices of the graph may be reachable from the start vertex
- the graph may contain cycles which may lead to infinite loops

Both of these problems can be taken care of by marking the nodes already visited and testing for this mark each time a nodes is about to be visited

## Depth-first search (DFS)

- whenever a vertex is visited during the traversal, all its unvisited neighbors are visited recursively

## Breadth-first search

- the vertices adjacent to the start vertex are visited before other vertices are visited recursively

# Representing graphs

## Adjacency matrix

**Definition.** For a graph G=(V,E) with n vertices ($V=\{v_1,v_2,\ldots,v_n\}$) the adjacency matrix is an n by n matrix M defined over {0,1} such that M[i,j]=1 a if there is a vertex connecting $v_i$ and $v_j$, and M[i,j]=0 otherwise.

- adjacency matrices can also be used for representation of weighted graphs

## Adjacency list

**Definition.** The adjacency list is an array of linked lists corresponding to each of the vertices of the graph. A list corresponding to a given vertex contains all the vertices adjacent to it.

## Comparing representations

From the point of view of **space requirement**:

- an adjacency matrix doesn’t use pointers
- an adjacency list only stores information about edges that appear in the graph
- adjacency matrices tend to be more efficient for dense matrices (high number of edges)

From the point of view of **time requirement**:

- for problems which only involve testing the existence of certain edges adjacency matrices are more efficient (due to indexing)
- for problems which require processing paths the adjacency list implementation is more efficient

# The graph ADT

**Type:** a set of graphs

**Operations:**

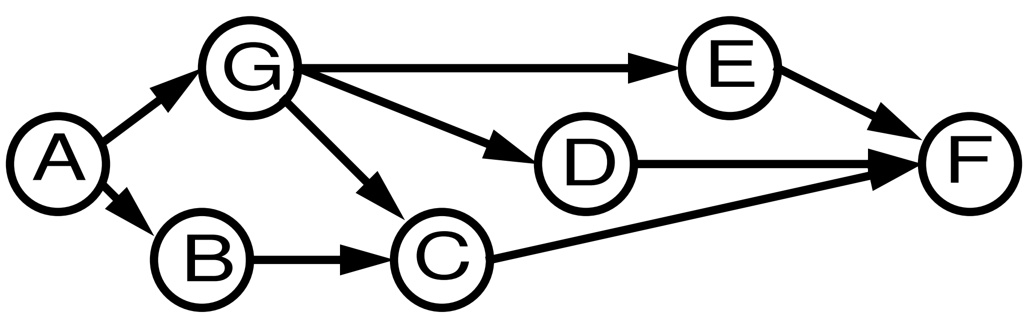
- insert a node
- add an edge
- remove an edge
- remove a node
- find a node

# Graph problems

## Topological sorting

**The problem:** given a directed acyclic graph order the vertices such that $(v_i,v_j)$ is an edge, than $v_i$ precedes $v_j$ in the ordering

- **solution** - use depth-first search to list the vertices
- **alternative solution** - use a queue to store all the vertices with no preceding vertices



## Shortest-path problems

### 1. **Single source shortest path**

**The problem:** given a (weighted) graph G and a vertex in G, find the shortest paths from x to all the other vertices in G

**Dijkstra's algorithm** - at any time maintain the shortest distance estimate for each node

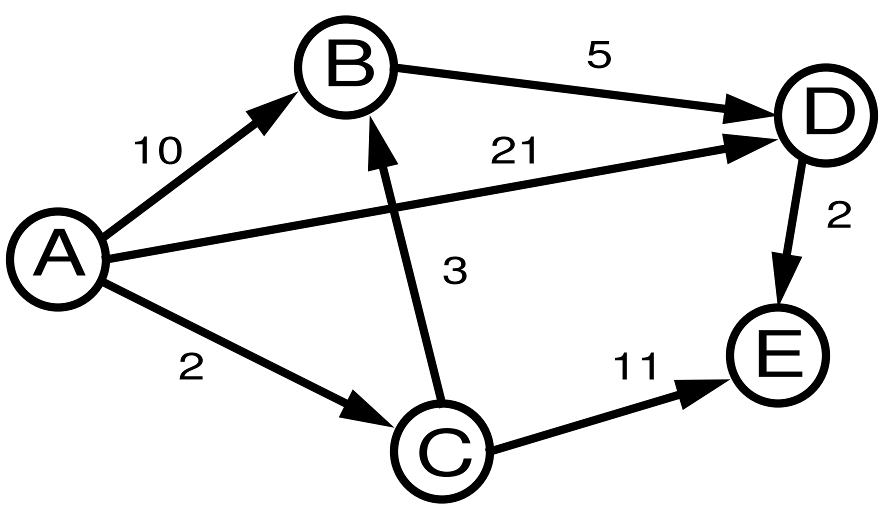
- finding the unvisited vertex with minimum estimated distance calls for selecting a data structure for organizing the vertices (list, priority queue)

## 2. **All-pair shortest path**

**The problem:** given a (weighted) graph, find the shortest distance between all pairs of vertices

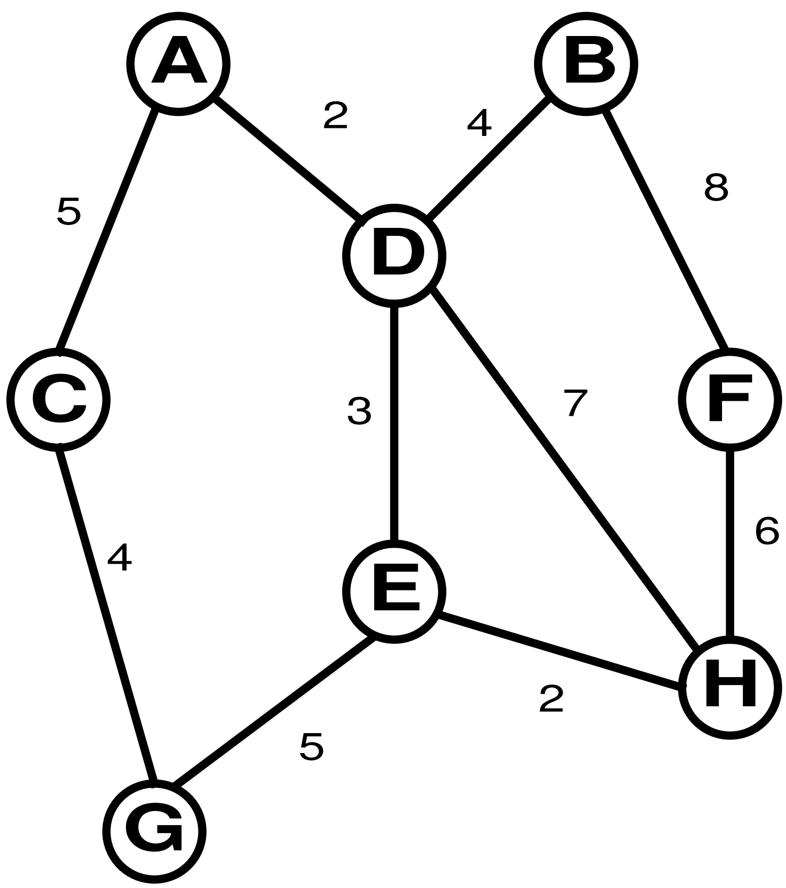
**Floyd's algorithm -** keep the shortest k-path for all pairs

- **k-path** from vertex x to vertex y is any path between x and y that has at most k intermediate vertices (0-path is an edge)



# Minimum cost spanning tree

**The problem:** a minimum-cost spanning tree of a weighted graph G is a subgraph of G that contains all the vertices of G, is a tree, and has the sum of the weights minimal



## 1. Prim's algorithm

- start from any vertex of the graph and add the lowest cost edge to the tree so far constructed till all the vertices had been added
- Prim's algorithm is a greedy algorithm

**Theorem.** Prim's algorithm produces a minimum-cost spanning tree

## 2. Kruskal's algorithm

- maintains a set of subtrees and adds edges (combines equivalence classes) one at a time by selecting the shortest one
- edges can be processed in order of weight by storing them into a min-heap