Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~ajp/

G52ADS 2014-15
Graphs

Introduction:

Basic definitions and concepts

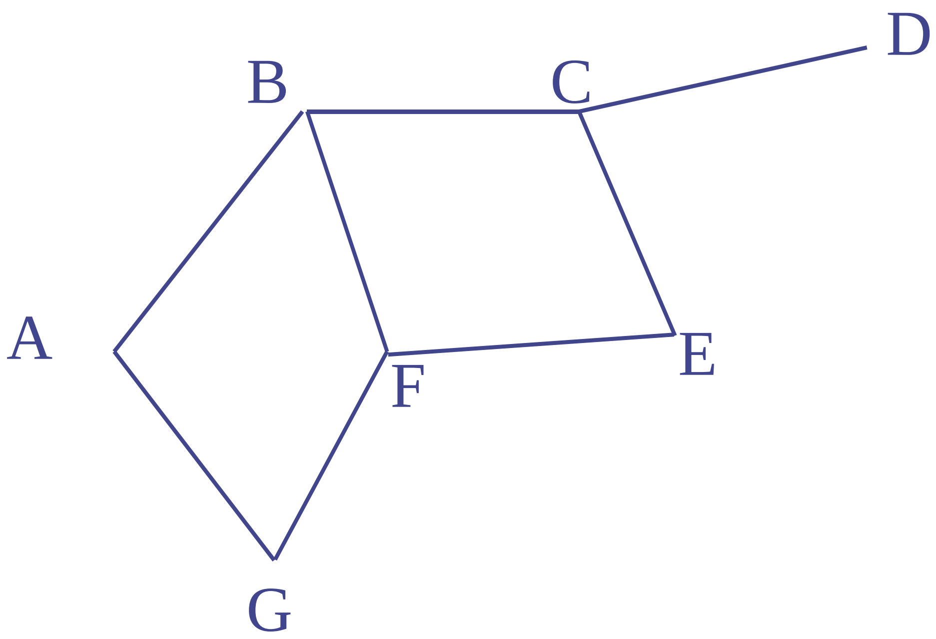
Contents:

Plan of the lecture:

- What is a graph?
- What are they used for?
- Graph problems.
- Two ways of implementing graphs.

Definition of a graph

A graph is a set of ***nodes***, or ***vertices***, connected by ***edges***.



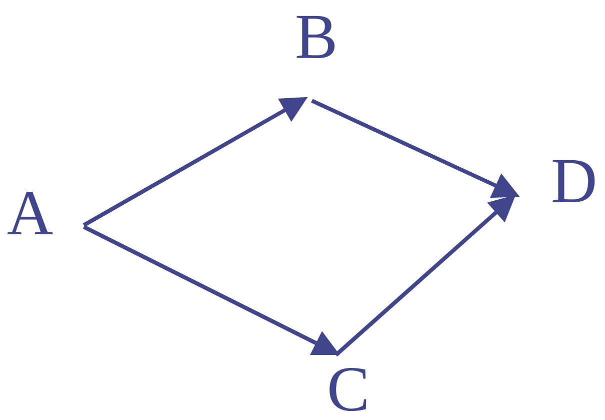
Some Applications of Graphs

- Graphs can be used to represent:
  - Networks (e.g., of computers or roads)
  - Flow charts
  - Tasks in some project (some of which should be completed before others), so edges correspond to prerequisites
  - States of an automaton / program

Directed and Undirected Graphs

Graphs can be

- undirected – edges don't have direction
- directed – edges have direction



directed graph
("digraph" for short)

Directed and Undirected Graphs

Undirected graphs can be represented as directed graphs where for each edge (X,Y) there is a corresponding edge (Y,X).

A ——— B ——— C

undirected graph

$A \rightleftarrows B \rightleftarrows C$

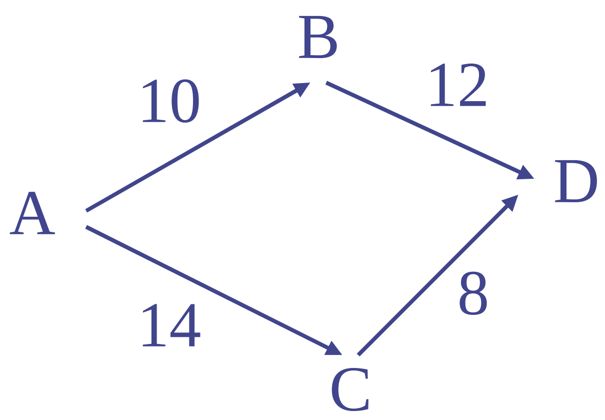
corresponding

directed graph

Weighted and Unweighted Graphs

Graphs can also be

- unweighted (as in the previous examples)
- weighted (edges have weights)

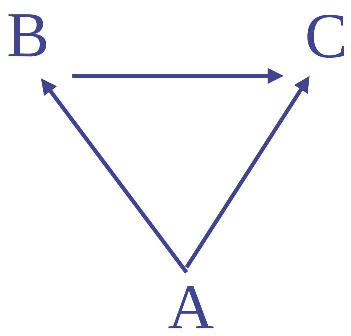


weighted (directed) graph

Notation

- Set V of *vertices* (nodes)
- Set E of *edges* ($E \subseteq V \times V$)

Example:



$$V = \{A, B, C\}, \quad E = \{(A,B), (A,C), (B,C)\}$$

Adjacency relation

- Node B is *adjacent* to A if there is an edge from A to B.

$$A \longrightarrow B$$

Paths and reachability

- A *path* from A to B is a sequence of vertices $A_1,\ldots,A_n$ such that there is an edge from A to A_1, from A_1 to A_2, …, from A_n to B.

$$A \longrightarrow A_1 \longrightarrow A_2 \longrightarrow A_3 \longrightarrow A_4 \longrightarrow A_5 \longrightarrow B$$

- A vertex B is *reachable* from A if there is a path from A to B

More Terminology

- A *cycle* is a path from a vertex to itself
- Graph is *acyclic* if it does not have cycles
- Graph is *connected* if there is a path between every pair of vertices
- Graph is *strongly connected* if there is a path in both directions between every pair of vertices (only relevant to digraphs)

Applications of Graphs

For example,

- nodes could represent positions in a board game, and edges the moves that transform one position into another …
- nodes could represent computers (or routers) in a network and weighted edges the bandwidth between them
- nodes could represent towns and weighted edges road distances between them, or train journey times or ticket prices …

Some Elementary Graph Problems

- Searching a graph for a vertex
- Searching a graph for an edge
- Finding a path in the graph (from one vertex to another)
- Finding the shortest path between two vertices
- Cycle detection

There are many advanced problems on graphs, and many real problems contain graph problems.

How to implement a graph

As with lists, there are several approaches, but most common options are:

- static indexed data structure
  - “Adjacency Matrix”
- dynamic data structure
  - “Adjacency Lists”

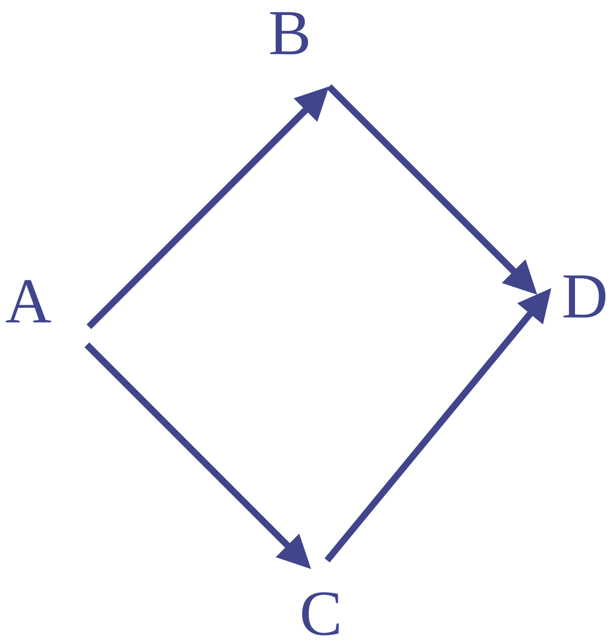
Static Implementation: Adjacency Matrix

- Store node in an array: each node is associated with an integer (array index)
- Represent information about the edges using a two dimensional array, where

```
array[i][j] == 1
```

iff there is an edge **from** node with index *i* **to** the node with index *j*.

Example



A	B	C	D
0	1	2	3

node indices

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	0	0	1
3	0	0	0	0

adjacency matrix

Weighted graphs

- For weighted graphs, place weights in the matrix
  - if there is no edge we use a value which can’t be confused with a weight, e.g., -1 or **`Integer.MAX_VALUE`**

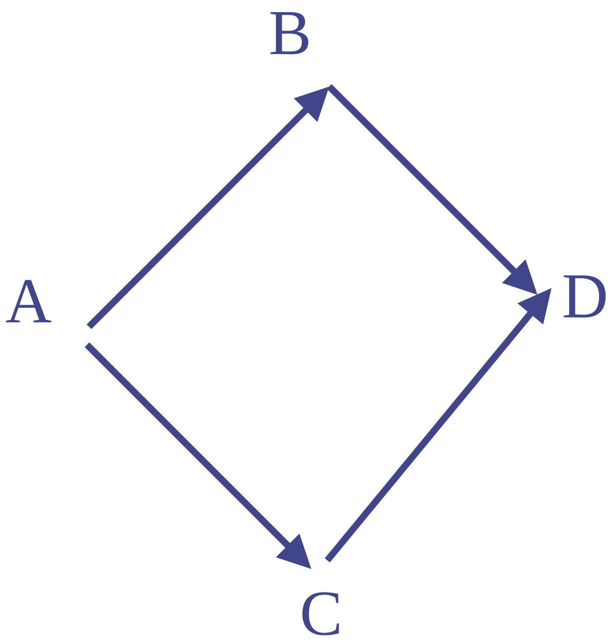
Disadvantages of adjacency matrices

- Sparse graphs with few edges for number that are possible result in many zero entries in adjacency matrix
  - This wastes space and makes many algorithms less efficient
  - e.g., to find nodes adjacent to a given node, we have to iterate through the whole row even if there are few 1s there
- Also, if the number of nodes in the graph may change, matrix representation is too inflexible
  - especially if we don't know the maximal size of the graph.

Adjacency List

- For every vertex, keep a list of adjacent vertices.
- Keep a list of vertices, or keep vertices in a Map (e.g. HashMap) as keys and lists of adjacent vertices as values.
- (The best choice depends on what the graph algorithm needs to do.)

Adjacency list



nodes — list of adjacent nodes

A → B, C

B → D

C → D

D →

Reading

- Goodrich and Tamassia (Ch. 13) have a somewhat different Graph implementation, where edges are first-class objects.
- In general, choice of implementation depends on what we want to do with a graph.