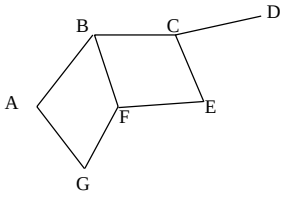
Graphs

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***.



Applications of Graphs

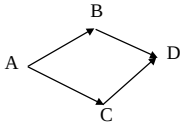
Graphs can be used to represent

- networks
- flow charts
- states of automaton / program
- tasks in some project (some of which should be completed before others), so edges correspond to prerequisites.

Directed and Undirected Graphs

Graphs can be

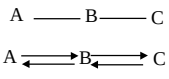
- undirected (edges don't have direction).
- directed (edges have direction).



directed graph

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).



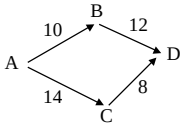
undirected graph

corresponding directed graph

Weighted and Unweighted Graphs

Graphs can also be

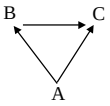
- unweighted (as in the previous examples)
- weighted (edges have weights).



Notation

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

Example: V = {A, B, C}, E = {(A,B), (A,C),(B,C)}:

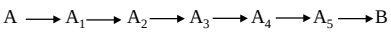


Adjacency relation

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

Paths and reachability

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



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

More Terminology

- A *cycle* is a path from u 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

Some graph problems

- searching a graph for a vertex
- finding a path in the graph (from one vertex to another) ; finding the shortest path between two vertices
- cycle detection

More graph problems

- topological sort (finding a linear sequence of vertices which agrees with the direction of edges in the graph; applied for scheduling tasks in a project)
- minimum spanning tree (deleting as many edges in a graph as possible, so that all vertices are still connected by shortest possible edges. Useful in circuit design.)

How to implement a graph

- As with lists, two approaches:
  - using a static indexed data structure
  - using a dynamic data structure

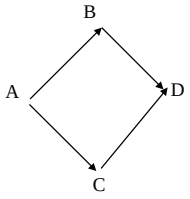
Static implementation:Adjacency Matrix

- Store nodes in the 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 i to node j.

Example



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

A	B	C	D
0	1	2	3

node indices

adjacency matrix

Weighted graphs

- For weighted graphs, place weights in matrix (for no edge, -1 or Integer.MAX_VALUE)

Example implementation

- If we know which nodes the graph is going to have, and the number of nodes is not going to change, we can implement a graph as follows.
- **`ArrayGraph`** class has an array to keep nodes (Objects) and a two dimensional array to keep 0s and 1s depending on whether there is an edge between corresponding nodes.

`ArrayGraph` class

```
public class ArrayGraph {
   int[][] matrix;  // adjacency matrix
   Object[] positions;
   public ArrayGraph(Object[] nodes) {
      positions = nodes;
      matrix = new
         int[nodes.length][nodes.length];
   }// this fills matrix with 0s
```

`ArrayGraph` class

```
public void addEdge(Object x, Object y) {
  if (indexOf(x)== -1 | indexOf(y) == -1)
  return;
  else matrix[indexOf(x)][indexOf(y)]= 1;
}
/* This method does nothing if nodes do
  not exist. indexOf method follows. */
```

`ArrayGraph` class

```
// to find o's position in the matrix...
public int indexOf(Object o) {
   for(int i=0; i<positions.length; i++){
     if(positions[i].equals(o)) {
        return i;
     }
   }
   return -1; // o is not in the graph
}
```

Notes on `ArrayGraph` class

- Keeping edge labels in an array is a bit awkward (have to look through the array each time to find out node's position in the matrix)
- Alternative solution could be using a HashMap: the Object (node) is the key, and Integer object holding its index is the value.
- When we need to know the node o's position we could do

```
int pos =
((Integer) map.get(o)).intValue();
```

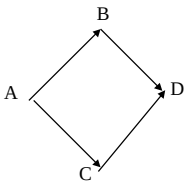
Disadvantages of adjacency matrices

- Sparse graphs: few edges for number of vertices.
- Leads to many zero entries in adjacency matrix: wastes space, makes many algorithms less efficient (to find nodes adjacent to a given node, 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.
- Represent a graph as a list or array of such lists: this is called an adjacency list implementation.

Adjacency list



nodes list of adjacent nodes

A → B, C

B → D

C → D

D →

Adjacency List Implementation

- Graph can be implemented as a list of lists, or any other structure holding vertices and lists of their neighbours.
- The simplest example is the **`LinkedGraph`** class: it keeps a list of **`GraphNode`** objects. Each **`GraphNode`** object keeps an **`Object`** and a neighbours list (of **`GraphNode`** objects).

`GraphNode` class

```
class GraphNode {
    Object label;
    LinkedList neighbours;

    GraphNode(Object o) {
        label = o;
        neighbours = new LinkedList();
     }
}
```

`LinkedGraph` class

```
import java.util.*;
public class LinkedGraph {
   LinkedList nodes;
   public LinkedGraph() {}
// constructor sets fields to null
   public void addNode(Object o) {
      nodes.add(new GraphNode(o));
   }
```

`LinkedGraph` class

```
public boolean containsNode(Object o) {
  ListIterator li = nodes.listIterator();
  while (li.hasNext()) {
    GraphNode n = (GraphNode) li.next();
    if (n.label.equals(o)) {
       return true;
    }
  }
  return false;
}
```

`LinkedGraph` class

- … other methods you would have to write yourself, but the idea should be clear:
- given an object (or two objects), find their corresponding **`GraphNode`** objects in the list, and then modify their neighbours lists etc.
- To implement graph traversal algorithms, we may need to add extra fields to the **`GraphNode`** class, for example a boolean flag to say that we have seen this node before.

Notes on **`LinkedGraph`** class

- Again, iterating through the list of nodes every time we need to find a **`GraphNode`** object corresponding to an **`Object o`** is awkward.
- One option (there are many others) is to use a **`HashMap`**: **`Object o`** is the key, and its neighbours list (of **`Objects`**) is the value. This way, don't need a **`GraphNode`** class.

HashMap implementation

Key	Value
A	Linked list containing B,C
B	Linked list containing D
C	Linked list containing D
D	Empty linked list

Summary and Reading

- Graphs can be used in many applications (anywhere where diagrams and maps are used).
- Graphs can be implemented with adjacency matrices or adjacency lists.
- For the formal coursework, have a look at Java Collections API to choose a suitable data structure for your implementation.
- Have a look at Shaffer, chapter 7 for graph terminology. His implementation of graphs assumes fixed size graphs (storing ints).