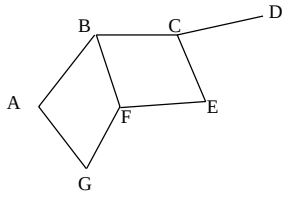
Graphs

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

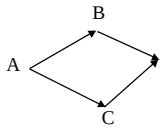
Graphs can be used to represent

- networks (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

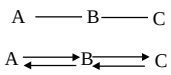
- undirected (edges don't have direction)
- directed (edges have direction)



directed graph

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).



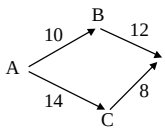
undirected graph

corresponding directed graph

Weighted and Unweighted Graphs

Graphs can also be

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

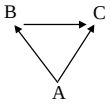


weighted graph

Notation

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

Example:



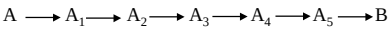
$V = \{A, B, C\}$, $E = \{(A,B), (A,C), (B,C)\}$

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 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

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, or ping times, or the QOS offered by the link ...
- nodes could represent towns and weighted edges road distances between them, or train journey times or ticket prices ...

Some 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

More graph problems

- Topological sort (finding a linear sequence of vertices which agrees with the direction of edges in the graph, e.g., 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, e.g., in network routing or circuit design.)

How to implement a graph

As with lists, there are several approaches, e.g.:

- using a static indexed data structure
- using a dynamic data structure

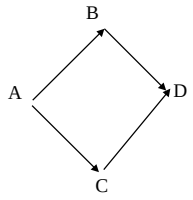
Static implementation:Adjacency Matrix

- Store node labels in the array: each label 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



	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 (if there is no edge we use a value which can't be confused with a weight, e.g., -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 node labels (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[][] adjacent;  // adjacency matrix
  Object[] labels;   // node labels

  public ArrayGraph(Object[] labels) {
    this.labels = labels;
    // fills adjacency matrix with 0s
    adjacent = new int[labels.length][labels.length];
   }
  // other methods ...
```

ArrayGraph class

```
// Example method to add an edge to the graph.
// Does nothing if nodes do not exist.
public void addEdge(Object x, Object y) {
  if (indexOf(x)== -1 || indexOf(y) == -1) return;
  else adjacent[indexOf(x)][indexOf(y)] = 1;
}
// indexOf method follows ...
```

ArrayGraph class

```
// To find o's index in the matrix...
protected int indexOf(Object label) {
  for(int i = 0; i < labels.length; i++){
    if(labels[i].equals(label)) {
      return i;
    }
  }
  return -1; // label is not in the graph
}
```

Notes on ArrayGraph class

- Keeping node 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) though we can sort the nodes
- Alternative solution would be to use a `HashMap`: the `Object` (node label) is the key, and an `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();
```

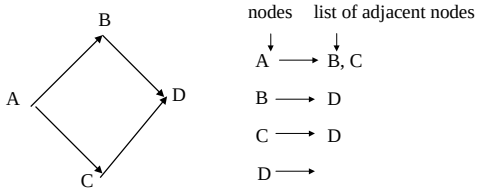
Disadvantages of adjacency matrices

- Sparse graphs with few edges for number of vertices 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.
- Represent a graph as a list or array of such lists: this is called an adjacency list implementation.

Adjacency list



Example 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 has an **`Object`** for the node label and a neighbours list (of **`GraphNode`** objects) for adjacent nodes.

`GraphNode` class

```
import java.util.*;
public class LinkedGraph {

  // Inner class for nodes ...
  class GraphNode {
    Object label;
    LinkedList neighbours;

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

`LinkedGraph` class

```
import java.util.*;
public class LinkedGraph {

  // GraphNode class definition omitted …

  LinkedList nodes;

  // Constructor sets fields to null
  public LinkedGraph() {}

  public void addNode(Object label) {
    nodes.add(new GraphNode(label));
  }
  // other methods ...
```

`LinkedGraph` class

```
// Example method ...
public boolean containsNode(Object label) {
  ListIterator li = nodes.listIterator();
  while (li.hasNext()) {
    GraphNode n = (GraphNode) li.next();
    if (n.label.equals(label)) {
       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 label`** is awkward.
- One option (there are many others) is to use a **`HashMap`**: **`Object label`** is the key, and its neighbours list (of **`Objects`**) is the value. This way, we 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 in many different ways, e.g., 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).