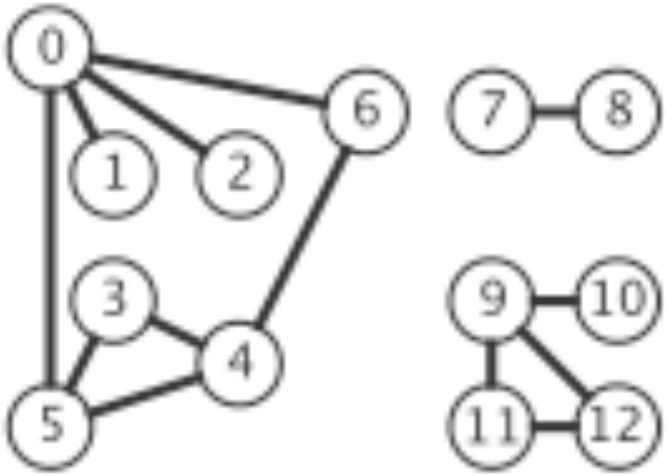
ID1020: Undirected Graphs

Dr. Johan Karlander
karlan@kth.se

kap 4.1 from Algorithms 4th Edition, Sedgewick.

Undirected graphs

Graphs. A *graph* is a set of *vertices* and a collection of *edges* that each connect a pair of vertices. We use the names 0 through V-1 for the vertices in a V-vertex graph.

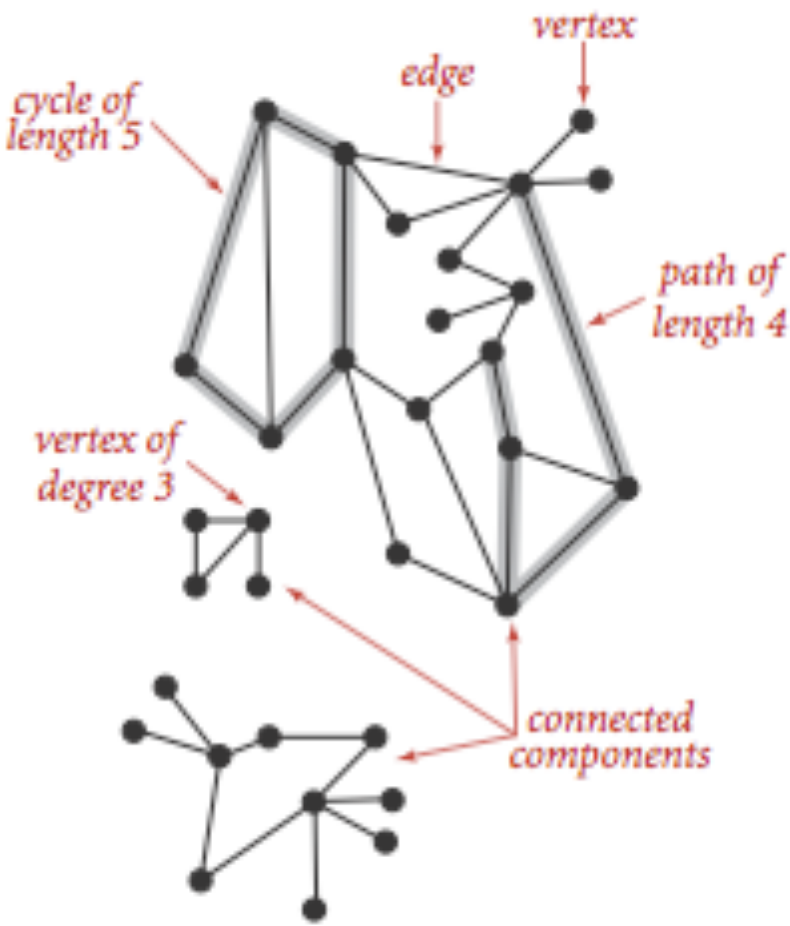


Terminology

- A *self-loop* is an edge that connects a vertex to itself.
- Two edges are *parallel* if they connect the same pair of vertices.
- When an edge connects two vertices, we say that the vertices are *adjacent to* one another and that the edge is *incident on* both vertices.
- The *degree* of a vertex is the number of edges incident on it.
- A *subgraph* is a subset of a graph's edges (and associated vertices) that constitutes a graph.

Paths and Cycles

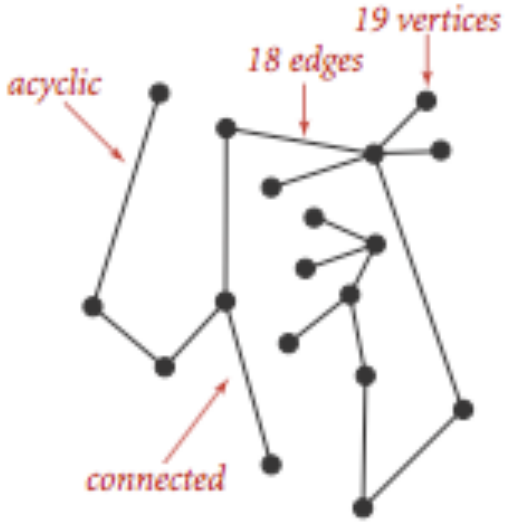
- A *path* in a graph is a sequence of vertices connected by edges. A *simple path* is one with no repeated vertices.
- A *cycle* is a path (with at least one edge) whose first and last vertices are the same. A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).
- The *length* of a path or a cycle is its number of edges.
- We say that one vertex is *connected to* another if there exists a path that contains both of them.
- A graph is *connected* if there is a path from every vertex to every other vertex.
- A graph that is not connected consists of a set of *connected components*, which are maximal connected subgraphs.



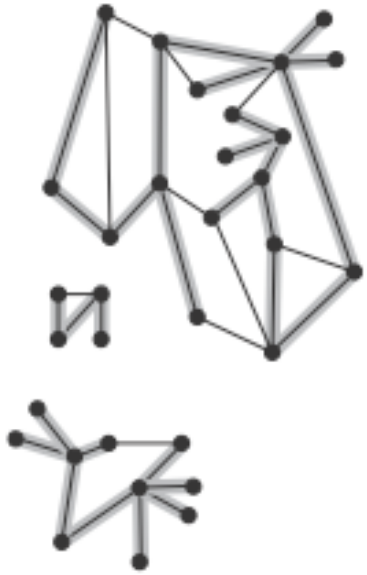
Anatomy of a graph

Trees

- An *acyclic graph* is a graph with no cycles.
- A *tree* is an acyclic connected graph.
- A *forest* is a disjoint set of trees.
- A *spanning tree* of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A *spanning forest* of a graph is the union of the spanning trees of its connected components.
- A *bipartite graph* is a graph whose vertices we can divide into two sets such that all edges connect a vertex in one set with a vertex in the other set.



A tree



A spanning forest

API for graphs

public class	Graph	
	Graph(int V)	*create a V-vertex graph with no edges*
	Graph(In in)	*read a graph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v-w *to this graph*
Iterable<Integer>	adj(int v)	*vertices adjacent to* v
String	toString()	*string representation*

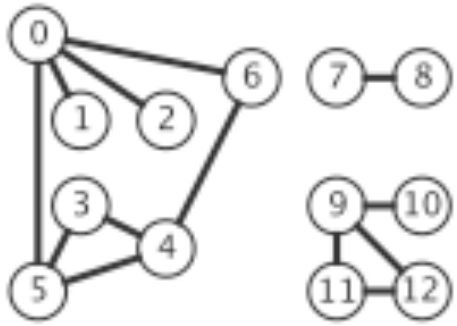
API for an undirected graph

The key method `adj()` allows client code to iterate through the vertices adjacent to a given vertex. Remarkably, we can build all of the algorithms that we consider in this section on the basic abstraction embodied in `adj()`.

We prepare the test data tinyG.txt, mediumG.txt, and largeG.txt, using the following input file format.

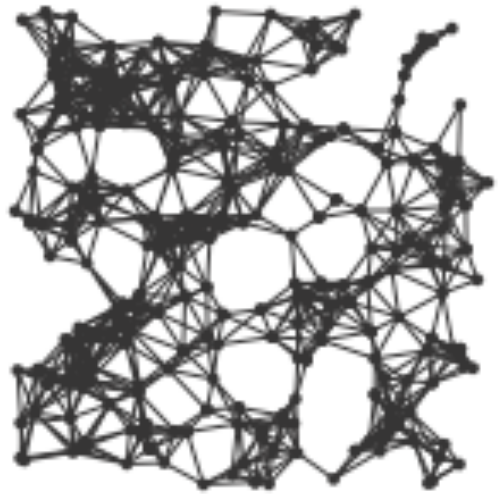
tinyG.txt

```
V → 13
    13 ← E
    0  5
    4  3
    0  1
    9  12
    6  4
    5  4
    0  2
    11 12
    9  10
    0  6
    7  8
    9  11
    5  3
```



mediumG.txt

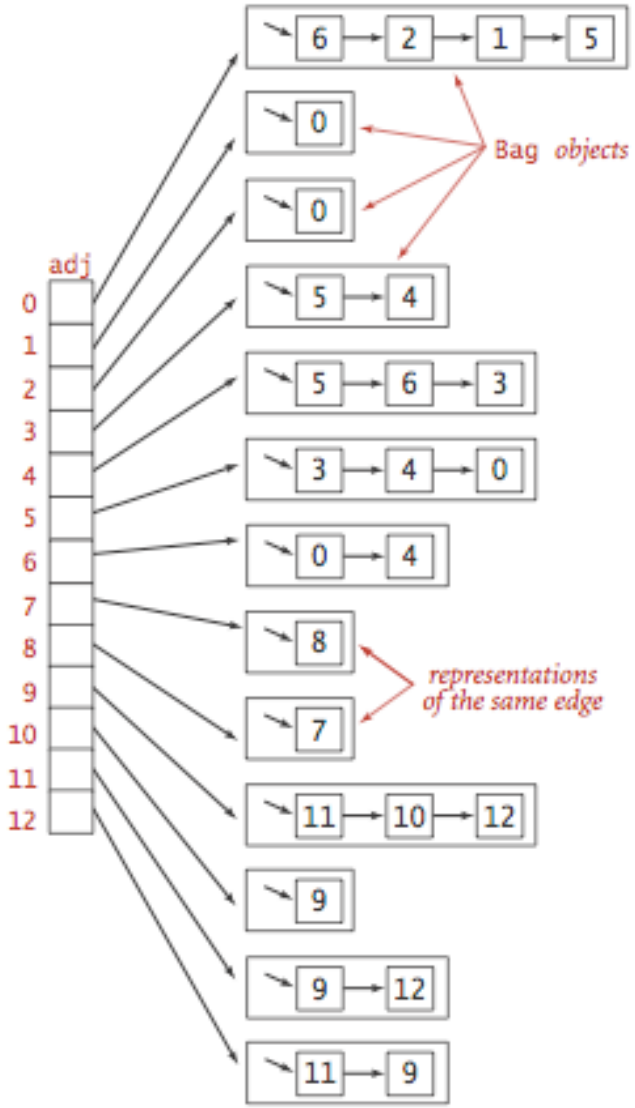
```
V → 250
    1273 ← E
    244 246
    239 240
    238 245
    235 238
    233 240
    232 248
    231 248
    229 249
    228 241
    226 231
    ...
    (1261 additional lines)
```



Input format for Graph constructor (two examples)

Representation of graphs

We use the *adjacency-lists representation*, where we maintain a vertex-indexed array of lists of the vertices connected by an edge to each vertex.



Adjacency-lists representation (undirected graph)

Dense and sparse graphs

In a dense graph we have $\Theta(|V|^2)$ edges. We can represent with a adjacency matrix.

In a sparse graph we have $\Theta(|V|)$ edges. It is usually best to represent it with adjacency lists.

Sparse graphs often occur in applications. It is often as important to know how well an algorithm works for such graphs as how it works in worst-case graphs.

DFS (Depth-First Search)

Depth-first search is a classic recursive method for systematically examining each of the vertices and edges in a graph. To visit a vertex

-
- Mark it as having been visited.
- Visit (recursively) all the vertices that are adjacent to it and that have not yet been marked.

DFS

DFS finds all nodes that can be reached from the start node s. It does this by trying to find as long paths as possible without returning to a previously visited node. When it has found such a path it back-tracks until a new possible path is found.

```
DFS(V, E, s)
foreach u ∈ V
    vis(u) ← 0
    p[u] ← NULL
DFS-Visit(s)
```

```
DFS-Visit(u)
vis(u) ← 1
foreach neighbor v to u
if vis(v) = 0
    p[v] = u
    DFS-Visit(v)
```

Time complexity: O(|V | + |E|)

Finding paths. It is easy to modify depth-first search to not only determine whether there exists a path between two given vertices but to find such a path (if one exists). We seek to implement the following API:

public class	Paths	
	Paths(Graph G, int s)	*find paths in G from source s*
boolean	hasPathTo(int v)	*is there a path from s to v?*
Iterable<Integer>	pathTo(int v)	*path from s to v; null if no such path*

To accomplish this, we remember the edge `v-w` that takes us to each vertex `w` for the *first* time by setting `edgeTo[w]` to `v`. In other words, `v-w` is the last edge on the known path from s to `w`. The result of the search is a tree rooted at the source; `edgeTo[]` is a parent-link representation of that tree. DepthFirstPaths.java implements this approach.

BFS (Breadth-First Search)

Breadth-first search. Depth-first search finds some path from a source vertex s to a target vertex v. We are often interested in finding the *shortest* such path (one with a minimal number of edges). Breadth-first search is a classic method based on this goal. To find a shortest path from `s` to `v`, we start at `s` and check for `v` among all the vertices that we can reach by following one edge, then we check for `v` among all the vertices that we can reach from `s` by following two edges, and so forth.

To implement this strategy, we maintain a queue of all vertices that have been marked but whose adjacency lists have not been checked. We put the source vertex on the queue, then perform the following steps until the queue is empty:

-
- Remove the next vertex `v` from the queue.
- Put onto the queue all unmarked vertices that are adjacent to `v` and mark them.

BreadthFirstPaths.java is an implementation of the `Paths` API that finds shortest paths. It relies on Queue.java for the FIFO queue.

BFS

BFS takes a graph and tries to visit all nodes, starting at node s. It first finds the neighbors of s, then the neighbors of neighbors and so on. A queue is used for keeping track of the nodes visited. In this implementation we also compute the shortest distance from s to all other nodes.

```
BFS(V, E, s)
foreach u∈V
    d[u]←∞
    d[s] ← 0
Q ← {s}
while Q/= ∅
    u ← Dequeue(Q)
    foreach neighbor v to u
        if d[v] = ∞
            d[v] ← d[u] + 1
            p[v] = u
        Enqueue(Q, v)
```

Connected Components

Connected components. Our next direct application of depth-first search is to find the connected components of a graph. Recall from Section 1.5 that "is connected to" is an equivalence relation that divides the vertices into equivalence classes (the connected components). For this task, we define the following API:

public class	CC	
	CC(Graph G)	*preprocessing constructor*
boolean	connected(int v, int w)	*are* v *and* w *connected?*
int	count()	*number of connected components*
int	id(int v)	*component identifier for* v (*between* 0 *and* count()-1)

Application: Bipartite graphs

We can tell if a graph is bipartite by running BFS. When we have found all distances d[u] from s to u, we call all nods with d[u] even blue nodes and all nodes with d[u] odd red nodes. We then check all edges. If we can find an edge between two blue nodes or between two red nodes, the graph is not bipartite, otherwise, it is bipartite.

Non-recursive DFS

```
private void dfs(Graph G, int s) {
    Stack<Integer> stack = new
Stack<Integer>();
    stack.push(s);
    while (!stack.isEmpty()) {
        int v = stack.pop();
        if (!marked[v]) {
            marked[v] = true;
            for (int w : G.adj(v))
                stack.push(w);
        }
    }
}
```

Problem

Is this DFS or BFS ore something else?

```
private void dfs(Graph G, int s) {
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(s);
    marked[s] = true;
    while (!stack.isEmpty()) {
        int v = stack.pop();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                stack.push(w);
                marked[w] = true;
            }
        }
    }
}
```