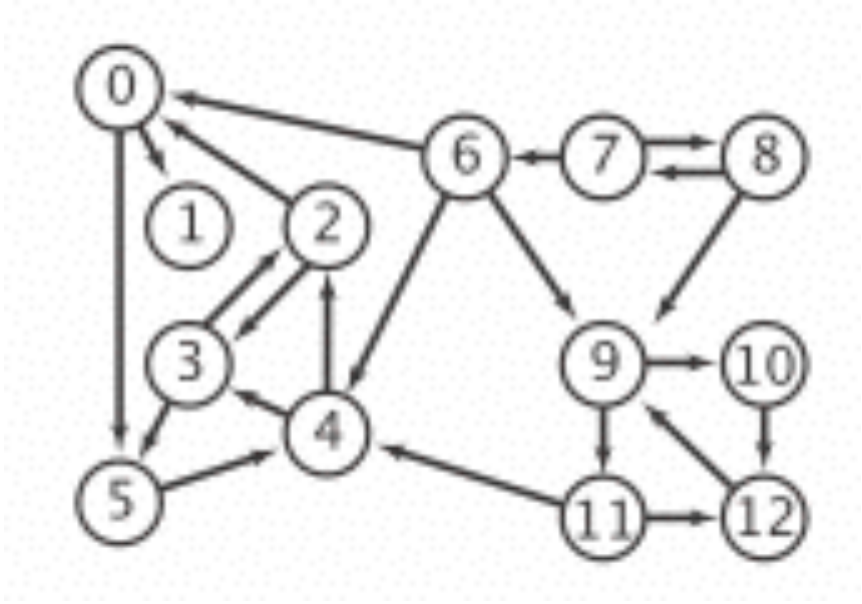
ID1020: Directed Graphs

Dr. Johan Karlander
karlan@kth.se

kap. 4.1-4.2 from Algorithms 4th Edition, Sedgewick.

Digraphs

Digraphs. A directed graph (or digraph) is a set of vertices and a collection of directed edges that each connects an ordered pair of vertices. We say that a directed edge points from the first vertex in the pair and points to the second vertex in the pair. 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 ordered 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 outdegree of a vertex is the number of edges pointing from it. The indegree of a vertex is the number of edges pointing to it.
- A subgraph is a subset of a digraph's edges (and associated vertices) that constitutes a digraph.

Paths in digraphs

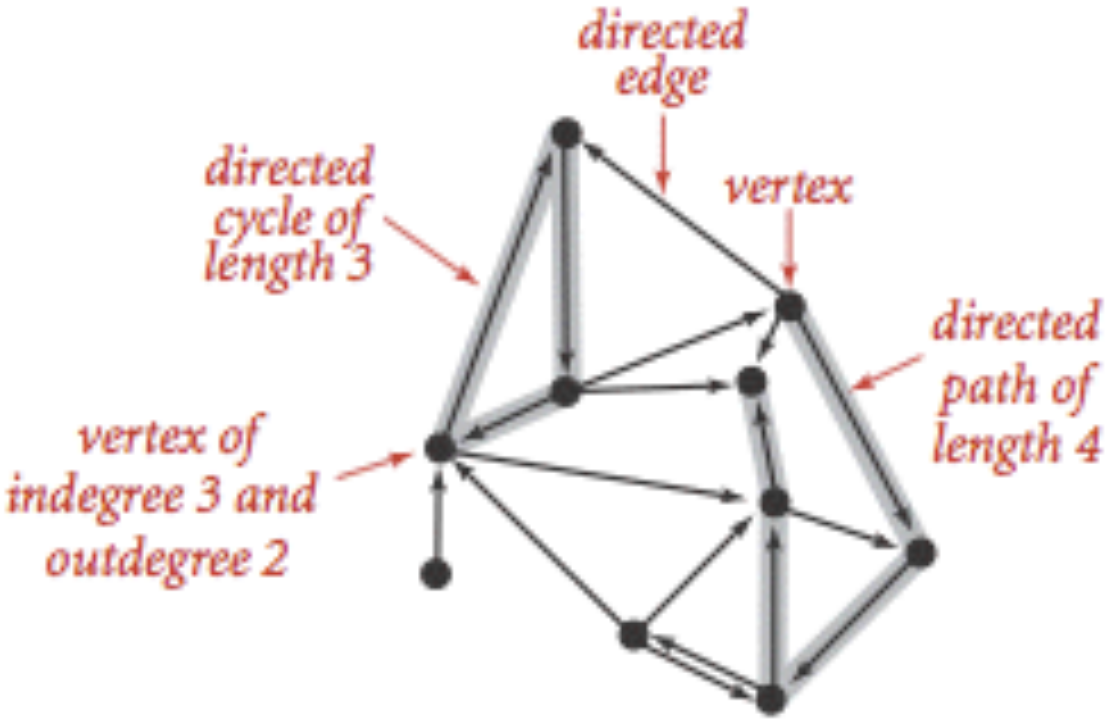
A *directed path* in a digraph is a sequence a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence. A *simple path* is one with no repeated vertices.

- A *directed cycle* is a directed 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 a vertex w is *reachable from* a vertex v if there exists a directed path from v to w.

Strongly connected components

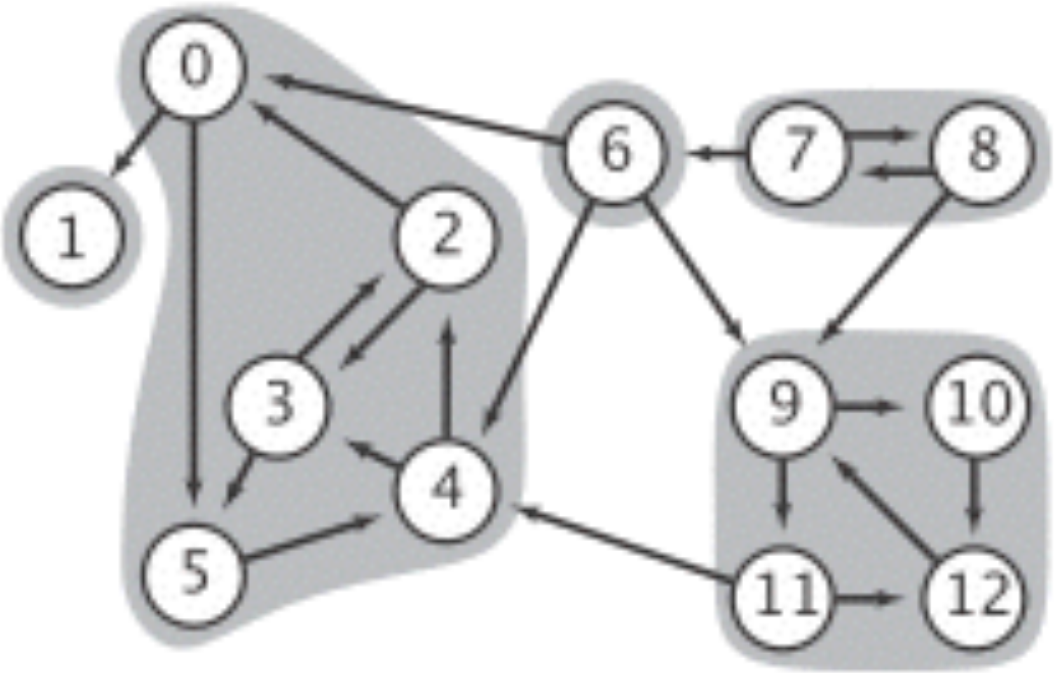
- We say that two vertices v and w are strongly connected if they are mutually reachable: there is a directed path from v to w and directed path from w to v.
- A digraph is strongly connected if there is a directed path from every vertex to every other vertex.
- A digraph that is not strongly connected consists of a set of strongly-connected components, which are maximal strongly-connected subgraphs.

Paths and Directed Cycles



Anatomy of a digraph

Strongly connected components



A digraph and its strong components

API

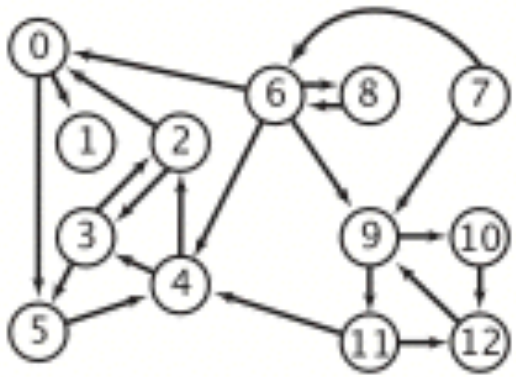
public class	Digraph	
	Digraph(int V)	*create a V-vertex digraph with no edges*
	Digraph(In in)	*read a digraph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v->w *to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to*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

The key method adj() allows client code to iterate through the vertices adjacent from a given vertex.

Representations

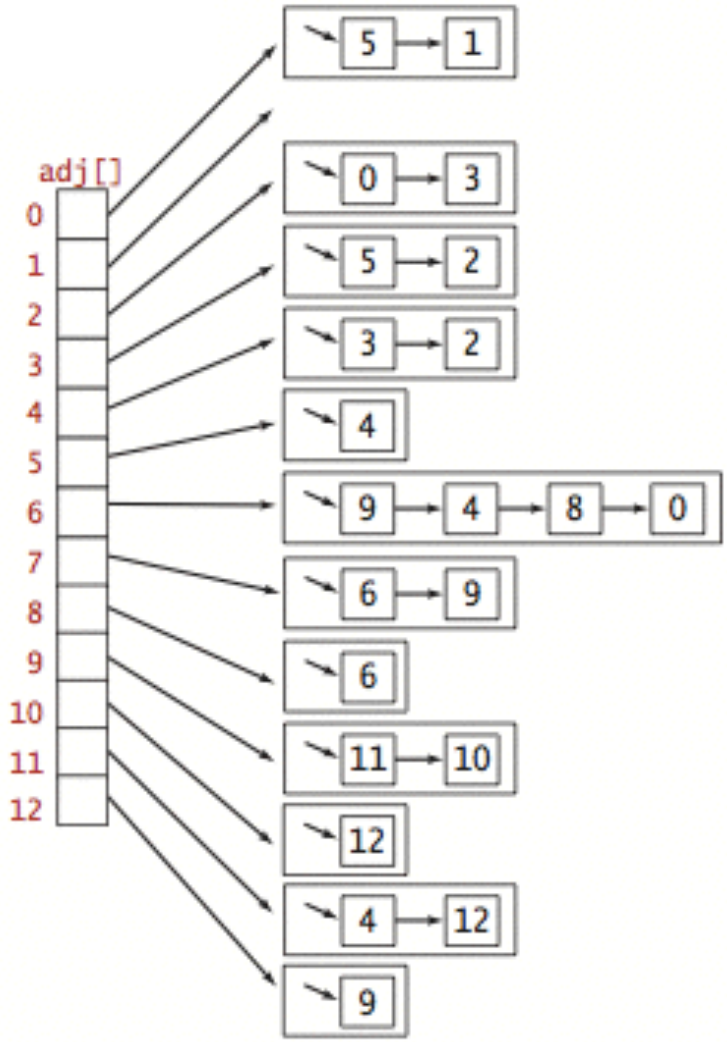
tinyDG.txt

V
13
E
22
4 2
2 3
3 2
6 0
0 1
2 0
11 12
12 9
9 10
9 11
7 9
10 12
11 4
4 3
3 5
6 8
8 6
5 4
0 5
6 4
6 9
7 6



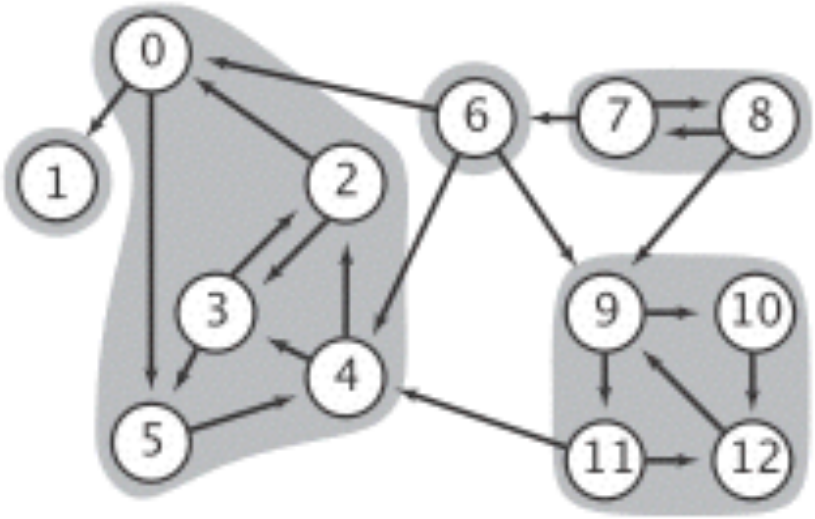
Representing using adjacency lists

Graph representation.
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.



Directed and undirected graphs

The concept reachability has a stricter meaning for digraphs.



A digraph and its strong components

In this graph the vertices 1, 2, 3, 4, 5 are reachable from 0. If we remove the directions on the edges we get an undirected graph. In this graph all vertices are reachable from 0.

Some search problems

- Single-source reachability: Given a digraph and source s, is there a directed path from s to v? If so, find such a path. DirectedDFS.java uses depth-first search to solve this problem.
- Multiple-source reachability: Given a digraph and a set of source vertices, is there a directed path from any vertex in the set to v? DirectedDFS.java uses depth-first search to solve this problem.
- Single-source directed paths: given a digraph and source s, is there a directed path from s to v? If so, find such a path. DepthFirstDirectedPaths.java uses depth-first search to solve this problem.
- Single-source shortest directed paths: given a digraph and source s, is there a directed path from s to v? If so, find a shortest such path. BreadthFirstDirectedPaths.java uses breadth-first search to solve this problem.

Modifications of DFS and BFS

Our algorithms can be used practically unchanged. The only difference is that ”v is a neighbor of u” should mean that there is a directed edge from u to v.

DFS:

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

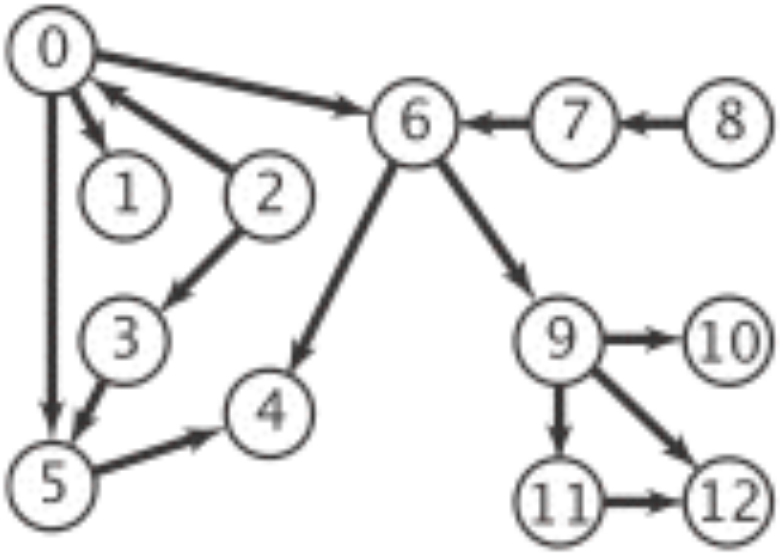
```
DFS-Visit(u)
vis(u) ← 1
foreach neighbor v to u
if vis(v) = 0
    p[v] = u
    DFS-Visit(v)
```

BFS again

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

Directed Acyclic Graphs (DAG)

A DAG is directed graph without any directed cycles. It is an important problem to decide if a graph is a DAG or not. It can be solved with a modified version of DFS.



A modified DFS

The idea is to mark vertices as ”halfdone” if they have been visited but still are processed.

```
DFS-DAG(V, E)
foreach u ∈ V
    vis(u) ← 0
foreach u ∈ V
    if vis(u) = 0
    DFS-Visit(u)
return ”The graph is a DAG"
```

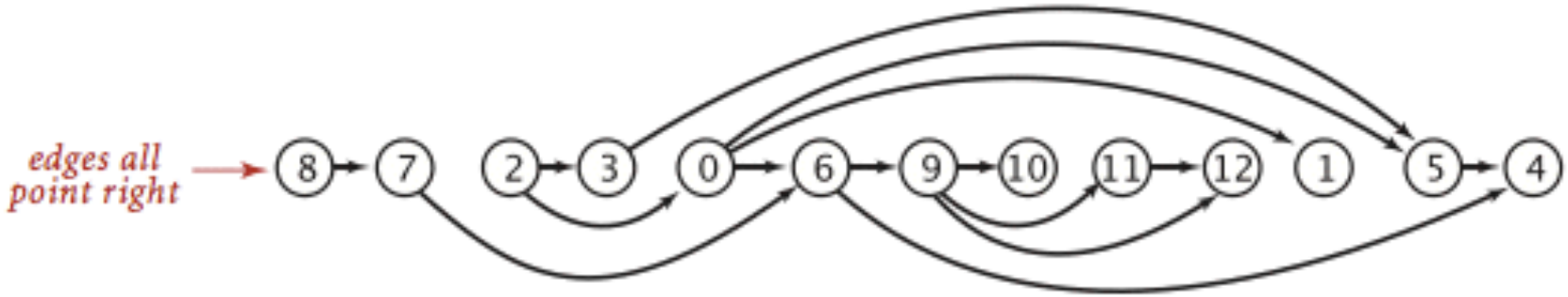
```
DFS-Visit(u)
vis(u) ← 0.5

foreach neighbor v to u
    if vis(v) = 0.5

    return ”The graph is not a DAG"
    if vis(v) = 0
        DFS-Visit(v)
vis(u) ← 1
```

Topological sort

Topological sort: given a digraph, put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order (or report that doing so is not possible). Topological.java solves this problem using depth-first search. Remarkably, a reverse postorder in a DAG provides a topological order.



Another modification of DFS

A topological sorting of a DAG is a numbering topnum[u] of the vertices such that if (u, v) is a directed edge then topnum[u] < topnum[v].
We can find such a numbering by with this modified DFS:

```
DFS-TOPNUM(V, E)
foreach u ∈ V
    vis(u) ← 0
    t←|V|
foreach u ∈ V
    if vis(u) = 0
        DFS-Visit(u)
return ”The graph is a DAG”
```

```
DFS-Visit(u)
vis(u) ← 0.5

foreach neighbor v to u
    if vis(v) = 0.5
        return ”The graph is not a DAG”
    if vis(v) = 0
        DFS-Visit(v)
vis(u) ← 1
 topnum[u] ← t
 t←t−1
```

Strong connectivity

Strong connectivity. Strong connectivity is an equivalence relation on the set of vertices:

- Reflexive: Every vertex v is strongly connected to itself.
- Symmetric: If v is strongly connected to w, then w is strongly connected to v.
- Transitive: If v is strongly connected to w and w is strongly connected to x, then v is also strongly connected to x.

Strong connectivity partitions the vertices into equivalence classes, which we refer to as strong components for short.

How do you decide strong connectivity?

To tell if a directed graph G is strongly connected we do a DFS-search starting from a vertex s. Let T be the set of vertices reachable from s. If T = V we continue: Let G_{rev} be G with all edge directions reversed. Do a DFS-search on G_{rev} starting from s.

If we now get T = V we know that the graph is strongly connected.

All strongly connected components

API:

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

Algorithm

Remarkably, KosarajuSharirSCC.java implements the API with just a few lines of code added to CC.java, as follows:

-
- Given a digraph G, use DepthFirstOrder.java to compute the reverse postorder of its reverse, G^R.
- Run standard DFS on G, but consider the unmarked vertices in the order just computed instead of the standard numerical order.
- All vertices reached on a call to the recursive dfs() from the constructor are in a strong component (!), so identify them as in CC.

A sketch of an implementation

SCC[G]

S ⟵ V(G)

For all v ∈ V(G)

    vis[v] ⟵ 0

C_counter ⟵ 1

While S is not empty

    Choose s ∈ S

    DFS[G,s]

    Let A be the set of vertices with vis = 1

For all v ∈ A

    vis[v] ⟵ 0

DFS[G_{rev} , s]

Remove all markt vis = 0 from A

For all v ∈ A

    Comp_nr[v] ⟵ C_counter

S ⟵ S -A

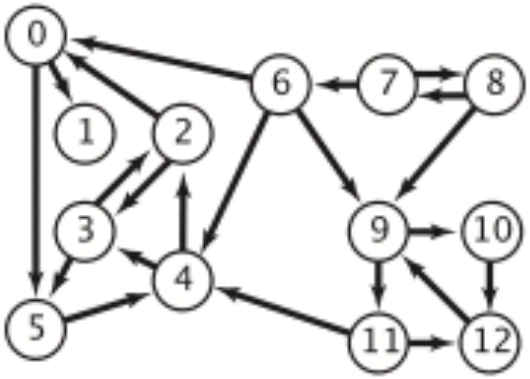
C_counter ⟵ C_counter + 1

For all v ∈ S

    vis[v] ⟵ 0

Transitive closure

Transitive closure. The transitive closure of a digraph G is another digraph with the same set of vertices, but with an edge from v to w if and only if w is reachable from v in G.



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	T	T	T	T	T	T							
1		T											
2	T	T	T	T	T	T							
3	T	T	T	T	T	T							
4	T	T	T	T	T	T							
5	T	T	T	T	T	T							
6	T	T	T	T	T	T	T			T	T	T	T
7	T	T	T	T	T	T	T	T	T	T	T	T	T
8	T	T	T	T	T	T	T	T	T	T	T	T	T
9	T	T	T	T	T	T				T	T	T	T
10	T	T	T	T	T	T				T	T	T	T
11	T	T	T	T	T	T				T	T	T	T
12	T	T	T	T	T	T				T	T	T	T

original edge (red)

self-loop (gray)

12 is reachable from 6

Algorithm

TransitiveClosure.java computes the transitive closure of a digraph by running depth-first search from each vertex and storing the results. This solution is ideal for small or dense digraphs, but it is not a solution for the large digraphs we might encounter in practice because the constructor uses space proportional to V^2 and time proportional to V (V + E).