4.2 Directed Graphs

- digraph API
- digraph search
- topological sort
- strong components
**Digraph.** Set of vertices connected pairwise by *directed* edges.
Road network

Vertex = intersection; edge = one-way street.
The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005

Vertex = political blog; edge = link.
Overnight interbank loan graph

Vertex = bank; edge = overnight loan.

The Topology of the Federal Funds Market, Bech and Atalay, 2008
Implication graph

Vertex = variable; edge = logical implication.

If $x_5$ is true, then $x_0$ is true.
Vertex = logical gate; edge = wire.
Vertex = synset; edge = hypernym relationship.

http://wordnet.princeton.edu
The McChrystal Afghanistan PowerPoint slide

## Digraph applications

<table>
<thead>
<tr>
<th>digraph</th>
<th>vertex</th>
<th>directed edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>
### Digraph-processing: algorithms of the day

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-source reachability</td>
<td>DFS</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>DFS (from each vertex)</td>
</tr>
<tr>
<td>Topological sort (DAG)</td>
<td>DFS</td>
</tr>
<tr>
<td>Strong components</td>
<td>Kosaraju DFS (twice)</td>
</tr>
</tbody>
</table>
‣ digraph API
‣ digraph search
‣ topological sort
‣ strong components
# Digraph API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Digraph</td>
<td></td>
</tr>
<tr>
<td>Digraph(int V)</td>
<td>create an empty digraph with V vertices</td>
</tr>
<tr>
<td>Digraph(In in)</td>
<td>create a digraph from input stream</td>
</tr>
<tr>
<td>void addEdge(int v, int w)</td>
<td>add a directed edge v→w</td>
</tr>
<tr>
<td>Iterable&lt;Integer&gt; adj(int v)</td>
<td>vertices pointing from v</td>
</tr>
<tr>
<td>int V()</td>
<td>number of vertices</td>
</tr>
<tr>
<td>int E()</td>
<td>number of edges</td>
</tr>
<tr>
<td>Digraph reverse()</td>
<td>reverse of this digraph</td>
</tr>
<tr>
<td>String toString()</td>
<td>string representation</td>
</tr>
</tbody>
</table>

```java
In in = new In(args[0]);
Digraph G = new Digraph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```
Digraph API

In in = new In(args[0]);
Digraph G = new Digraph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);

% java Digraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
⋮
11->4
11->12
12->9

read digraph from input stream
print out each edge (once)
Set-of-edges digraph representation

Store a list of the edges (linked list or array).
Adjacency-matrix digraph representation

Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v \rightarrow w$ in the digraph: $\text{adj}[v][w] = \text{true}$.
Maintain vertex-indexed array of lists.
Adjacency-lists graph representation: Java implementation

```java
public class Graph {
    private final int V;
    private final Bag<Integer>[] adj;

    public Graph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```

- adjacency lists
- create empty graph with V vertices
- add edge v–w
- iterator for vertices adjacent to v
Adjacency-lists digraph representation: Java implementation

```java
public class Digraph {
    private final int V;
    private final Bag<Integer>[] adj;

    public Digraph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from \( v \).
- Real-world digraphs tend to be sparse.

### Digraph representations

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from ( v ) to ( w )</th>
<th>edge from ( v ) to ( w )?</th>
<th>iterate over vertices pointing from ( v )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>( E )</td>
<td>1</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V )</td>
<td>1</td>
<td>( 1 )</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>( E + V )</td>
<td>1</td>
<td>outdegree(( v ))</td>
<td>outdegree(( v ))</td>
</tr>
</tbody>
</table>

† disallows parallel edges

huge number of vertices, small average vertex degree
- digraph API
- digraph search
- topological sort
- strong components
Reachability

Problem. Find all vertices reachable from \( s \) along a directed path.
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

**DFS (to visit a vertex v)**

- Mark v as visited.
- Recursively visit all unmarked vertices w pointing from v.

- See Depth-first search in digraphs demo
Depth-first search (in undirected graphs)

Recall code for **undirected** graphs.

```java
public class DepthFirstSearch {
    private boolean[] marked;

    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v) { return marked[v]; }
}
```

- true if path to s
- constructor marks vertices connected to s
- recursive DFS does the work
- client can ask whether any vertex is connected to s
Depth-first search (in directed graphs)

Code for directed graphs identical to undirected one.
[substitute Digraph for Graph]

```java
public class DirectedDFS {
    private boolean[] marked;

    public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v) {
        return marked[v];
    }
}
```
true if path from s
constructor marks vertices reachable from s
recursive DFS does the work
client can ask whether any vertex is reachable from s
Every program is a digraph.
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.
Find (and remove) unreachable code.

Infinite-loop detection.
Determine whether exit is unreachable.
Every data structure is a digraph.
- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.

- Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.
Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

**BFS (from source vertex s)**

Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- remove the least recently added vertex v
- for each unmarked vertex pointing from v:
  add to queue and mark as visited.

**Proposition.** BFS computes shortest paths (fewest number of edges).
Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex.

Ex. Shortest path from \{ 1, 7, 10 \} to 5 is 7→6→4→3→5.
Breadth-first search in digraphs application: web crawler


Solution. BFS with implicit graph.

BFS.
- Choose root web page as source $s$.
- Maintain a queue of websites to explore.
- Maintain a set of discovered websites.
- Dequeue the next website and enqueue websites to which it links (provided you haven't done so before).

Q. Why not use DFS?
Bare-bones web crawler: Java implementation

```java
Queue<String> queue = new Queue<String>();
SET<String> discovered = new SET<String>();

String root = "http://www.princeton.edu";
queue.enqueue(root);
discovered.add(root);

while (!queue.isEmpty())
{
    String v = queue.dequeue();
    StdOut.println(v);
    In in = new In(v);
    String input = in.readAll();

    String regexp = "http://(\w+\.)*(\w+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);
    while (matcher.find())
    {
        String w = matcher.group();
        if (!discovered.contains(w))
        {
            discovered.add(w);
            queue.enqueue(w);
        }
    }
}
```

queue of websites to crawl
set of discovered websites

start crawling from root website

read in raw html from next website in queue

use regular expression to find all URLs in website of form http://xxx.yyy.zzz
[crude pattern misses relative URLs]

if undiscovered, mark it as discovered and put on queue
- digraph API
- digraph search
- topological sort
- strong components
Precedence scheduling

**Goal.** Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

**Digraph model.** vertex = task; edge = precedence constraint.
Topological sort

**DAG.** Directed **acyclic** graph.

**Topological sort.** Redraw DAG so all edges point upwards.

\[
\begin{array}{c}
0 \rightarrow 5 \\
0 \rightarrow 1 \\
3 \rightarrow 5 \\
5 \rightarrow 4 \\
6 \rightarrow 0 \\
1 \rightarrow 4 \\
0 \rightarrow 2 \\
3 \rightarrow 6 \\
3 \rightarrow 4 \\
6 \rightarrow 4 \\
3 \rightarrow 2 \\
\end{array}
\]

directed edges

\[
\begin{array}{c}
0 \\
2 \\
3 \\
6 \\
4 \\
5 \\
1 \\
\end{array}
\]

DAG

topological order
Depth-first search order

public class DepthFirstOrder
{
    private boolean[] marked;
    private Stack<Integer> reversePost;

    public DepthFirstOrder(Digraph G)
    {
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
        reversePost.push(v);
    }

    public Iterable<Integer> reversePost()
    {
        return reversePost;
    }
}
- digraph API
- digraph search
- topological sort
- strong components
Connected components vs. strongly-connected components

Analog to connectivity in undirected graphs.

V and w are **connected** if there is a path between v and w

V and w are **strongly connected** if there is a directed path from v to w and a directed path from w to v

3 connected components

5 strongly-connected components

Connected component id (easy to compute with DFS)

```
cc[] = [0 0 0 0 0 0 1 1 1 2 2 2 2]
```

Strongly-connected component id (how to compute?)

```
scc[] = [1 0 1 1 1 1 3 4 3 2 2 2 2]
```

Constant-time client connectivity query

```
public int connected(int v, int w) {
    return cc[v] == cc[w];
}
```

Constant-time client strong-connectivity query

```
public int stronglyConnected(int v, int w) {
    return scc[v] == scc[w];
}
```
Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.

http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif
Strong component application: software modules

Software module dependency graph.
- Vertex = software module.
- Edge: from module to dependency.

Strong component. Subset of mutually interacting modules.
Approach 1. Package strong components together.

Firefox

Internet Explorer
Strong components algorithms: brief history

1960s: Core OR problem.
• Widely studied; some practical algorithms.
• Complexity not understood.

1972: linear-time DFS algorithm (Tarjan).
• Classic algorithm.
• Level of difficulty: Algs4++. 
• Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).
• Forgot notes for lecture; developed algorithm in order to teach it!
• Later found in Russian scientific literature (1972).

1990s: more easy linear-time algorithms.
• Gabow: fixed old OR algorithm.
• Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.
Kosaraju's algorithm: intuition

Reverse graph. Strong components in $G$ are same as in $G^R$.

Kernel DAG. Contract each strong component into a single vertex.

Idea.
- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.