4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
- strong components

see videos
4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
Road network

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

The Political blogosphere digraph

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

Vertex = bank; edge = overnight loan.

The Topology of the Federal Funds Market, Bech and Atalay, 2008
Uber taxi digraph

Vertex = taxi pickup; edge = taxi ride.

http://blog.uber.com/2012/01/09/uberdata-san-franciscomics/
## 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>
**Directed graph terminology**

**Digraph.** Set of vertices connected pairwise by directed edges.
## Some digraph problems

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s→t path</td>
<td>Is there a path from s to t ?</td>
</tr>
<tr>
<td>shortest s→t path</td>
<td>What is the shortest path from s to t ?</td>
</tr>
<tr>
<td>directed cycle</td>
<td>Is there a directed cycle in the graph ?</td>
</tr>
<tr>
<td>topological sort</td>
<td>Can the digraph be drawn so that all edges point upwards?</td>
</tr>
<tr>
<td>strong connectivity</td>
<td>Is there a directed path between every pairs of vertices ?</td>
</tr>
<tr>
<td>transitive closure</td>
<td>For which vertices v and w is there a directed path from v to w ?</td>
</tr>
<tr>
<td>PageRank</td>
<td>What is the importance of a web page ?</td>
</tr>
</tbody>
</table>
4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
## Digraph API

Almost identical to Graph API.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 adjacent 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>
Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.
Which is order of growth of running time of the following code fragment if the digraph uses the adjacency-lists representation, where $V$ is the number of vertices and $E$ is the number of edges?

A. $V$

B. $E + V$

C. $V^2$

D. $VE$

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

prints each edge exactly once
Digraph representations

In practice. Use adjacency-lists representation.

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

<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 adjacent 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^2 )</td>
<td>1 ( ^\dagger )</td>
<td>1</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>( E + V )</td>
<td>1</td>
<td>( \text{outdegree}(v) )</td>
<td>( \text{outdegree}(v) )</td>
</tr>
</tbody>
</table>

\( ^\dagger \) disallows parallel edges

huge number of vertices, small average vertex outdegree
Adjacency-lists graph representation (review): Java implementation

```java
public class Graph {
    private final int V;
    private 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>(v);
    }

    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 digraph representation: Java implementation

```java
public class Digraph {
    private final int V;
    private 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];
    }
}
```
4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
Reachability

**Problem.** Find all vertices reachable from $s$ along a directed path.
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 vertex v.
- Recursively visit all unmarked vertices w adjacent from v.
Depth-first search demo

To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent from $v$.

A directed graph
To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent from $v$. 

### Depth-first search demo

![Graph diagram]

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>

reachable from vertex 0
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 connected 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.

```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 connected to s
- constructor marks vertices connected to s
- recursive DFS does the work
- client can ask whether any vertex is connected to s
Reachability application: program control-flow analysis

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.
Reachability application: mark-sweep garbage collector

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).
Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).
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.

---

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$, for some constants $k_1$, $k_2$, and $k_3$, where $V$ is the number of vertices and $E$ is the number of edges of the graph being examined.

* SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

† ROBERT TARJAN
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 adjacent from v:
  add to queue and mark as visited.

**Proposition.** BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$. 
Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex \( v \) from queue.
- Add to queue all unmarked vertices adjacent from \( v \) and mark them.
Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex \( v \) from queue.
- Add to queue all unmarked vertices adjacent from \( v \) and mark them.

<table>
<thead>
<tr>
<th>( v )</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

done
MULTIPLE-SOURCE SHORTEST PATHS

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

Ex. \( S = \{1, 7, 10\} \).
- Shortest path to 4 is \(7 \rightarrow 6 \rightarrow 4\).
- Shortest path to 5 is \(7 \rightarrow 6 \rightarrow 0 \rightarrow 5\).
- Shortest path to 12 is \(10 \rightarrow 12\).

Q. How to implement multi-source shortest paths algorithm?
Suppose that you want to design a web crawler. Which graph search algorithm should you use?

A. depth-first search
B. breadth-first search
C. either A or B
D. neither A nor B
<table>
<thead>
<tr>
<th>BFS crawl</th>
<th>DFS crawl</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org">http://www.w3.org</a></td>
<td><a href="http://deimos.apple.com">http://deimos.apple.com</a></td>
</tr>
<tr>
<td><a href="http://ogp.me">http://ogp.me</a></td>
<td><a href="http://www.youtube.com">http://www.youtube.com</a></td>
</tr>
<tr>
<td><a href="http://giving.princeton.edu">http://giving.princeton.edu</a></td>
<td><a href="http://www.google.com">http://www.google.com</a></td>
</tr>
<tr>
<td><a href="http://library.princeton.edu">http://library.princeton.edu</a></td>
<td><a href="http://googlenewsblog.blogspot.com">http://googlenewsblog.blogspot.com</a></td>
</tr>
<tr>
<td><a href="http://tigernet.princeton.edu">http://tigernet.princeton.edu</a></td>
<td><a href="http://groups.google.com">http://groups.google.com</a></td>
</tr>
<tr>
<td><a href="http://princetonusg.com">http://princetonusg.com</a></td>
<td><a href="http://fusion.google.com">http://fusion.google.com</a></td>
</tr>
<tr>
<td><a href="http://artmuseum.princeton.edu">http://artmuseum.princeton.edu</a></td>
<td><a href="http://insidesearch.blogspot.com">http://insidesearch.blogspot.com</a></td>
</tr>
<tr>
<td><a href="http://jobs.princeton.edu">http://jobs.princeton.edu</a></td>
<td><a href="http://agoogleaday.com">http://agoogleaday.com</a></td>
</tr>
<tr>
<td><a href="http://odoc.princeton.edu">http://odoc.princeton.edu</a></td>
<td><a href="http://static.googleusercontent.com">http://static.googleusercontent.com</a></td>
</tr>
<tr>
<td><a href="http://twitter.com">http://twitter.com</a></td>
<td><a href="http://www.dot.ca.gov">http://www.dot.ca.gov</a></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Breadth-first search in digraphs application: web crawler


Solution. [BFS with implicit digraph]
- 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).
Bare-bones web crawler: Java implementation

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

String root = "http://www.princeton.edu";
queue.enqueue(root);
marked.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+)(\w+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);

    while (matcher.find())
    {
        String w = matcher.group();
        if (!marked.contains(w))
        {
            marked.add(w);
            q.enqueue(w);
        }
    }
}
```

- queue of websites to crawl
- set of marked 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 unmarked, mark and enqueue
4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
Combinational circuit

Vertex = logical gate; edge = wire.
WordNet digraph

Vertex = synset; edge = hypernym relationship.

http://wordnet.princeton.edu
Vertex = revision of repository; edge = revision relationship.
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.

\[
\begin{align*}
0 \rightarrow 5 & \quad 0 \rightarrow 2 \\
0 \rightarrow 1 & \quad 3 \rightarrow 6 \\
3 \rightarrow 5 & \quad 3 \rightarrow 4 \\
5 \rightarrow 2 & \quad 6 \rightarrow 4 \\
6 \rightarrow 0 & \quad 3 \rightarrow 2 \\
1 \rightarrow 4
\end{align*}
\]

Directed edges

DAG

Topological order
Suppose that you want to find a topological order of a DAG. Which graph search algorithm should you use?

A. depth-first search
B. breadth-first search
C. either A or B
D. neither A nor B
Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

A directed acyclic graph

tinyDAG7.txt

7
11
0  5
0  2
0  1
3  6
3  5
3  4
5  2
6  4
6  0
3  2
Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

```
postorder
4 1 2 5 0 6 3

topological order
3 6 0 5 2 1 4
```

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

    public DepthFirstOrder(Digraph G) {
        reversePostorder = 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);
        reversePostorder.push(v);
    }

    public Iterable<Integer> reversePostorder() {
        return reversePostorder;
    }
}
Why does topological sort algorithm work?

- First vertex in postorder has outdegree 0.
- Second-to-last vertex in postorder can only point to last vertex.
- ...

Topological sort in a DAG: intuition

```
postorder
4 1 2 5 0 6 3

topological order
3 6 0 5 2 1 4
```
Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge $v \rightarrow w$. When $dfs(v)$ is called:

- **Case 1**: $dfs(w)$ has already been called and returned.
  - thus, $w$ appears before $v$ in postorder

- **Case 2**: $dfs(w)$ has not yet been called.
  - $dfs(w)$ will get called directly or indirectly by $dfs(v)$
  - so, $dfs(w)$ will finish before $dfs(v)$
  - thus, $w$ appears before $v$ in postorder

- **Case 3**: $dfs(w)$ has already been called, but has not yet returned.
  - function-call stack contains path from $w$ to $v$
  - edge $v \rightarrow w$ would complete a cycle
  - contradiction (this case can't happen in a DAG)
**Directed cycle detection**

**Proposition.** A digraph has a topological order iff no directed cycle.

**Pf.**
- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.

![A digraph with a directed cycle](image)

**Goal.** Given a digraph, find a directed cycle.

**Solution.** DFS. What else? See textbook.
Directed cycle detection application: precedence scheduling

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

![Table Example](http://xkcd.com/754)

**Remark.** A directed cycle implies scheduling problem is infeasible.
Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

```java
public class A extends B {
    ...
}

public class B extends C {
    ...
}

public class C extends A {
    ...
}
```

```
% javac A.java
A.java:1: cyclic inheritance involving A
public class A extends B {
   ^
1 error
```
Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)
Observation. DFS visits each vertex exactly once. The order in which it does so can be important.

Orderings.

- Preorder: order in which dfs() is called.
- Postorder: order in which dfs() returns.
- Reverse postorder: reverse order in which dfs() returns.

```java
private void dfs(Graph G, int v) {
    marked[v] = true;
    preorder.enqueue(v);
    for (int w : G.adj(v))
        if (!marked[w]) dfs(G, w);
    postorder.enqueue(v);
    reversePostorder.push(v);
}
```
# Digraph-processing summary: algorithms of the day

<table>
<thead>
<tr>
<th>Operation</th>
<th>Digraph</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-source reachability in a digraph</td>
<td><img src="image1.png" alt="Graph" /></td>
<td>DFS/BFS</td>
</tr>
<tr>
<td>Shortest path in a digraph</td>
<td><img src="image2.png" alt="Graph" /></td>
<td>BFS</td>
</tr>
<tr>
<td>Topological sort in a DAG</td>
<td><img src="image3.png" alt="Graph" /></td>
<td>DFS</td>
</tr>
</tbody>
</table>