4.2 Directed Graphs

- digraph API
- digraph search
- topological sort
- strong components
Directed graphs

**Digraph.** Set of vertices connected pairwise by directed edges.
Road network

Vertex = intersection; edge = one-way street.
Political blogosphere graph

Vertex = political blog; edge = link.

The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005
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.
Combinational circuit

Vertex = logical gate; edge = wire.
Vertex = synset; edge = hypernym relationship.
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>
Some digraph problems

Path. Is there a directed path from \( s \) to \( t \)?

Shortest path. What is the shortest directed path from \( s \) to \( t \)?

Topological sort. Can you draw the digraph so that all edges point upwards?

Strong connectivity. Is there a directed path between all pairs of vertices?

Transitive closure. For which vertices \( v \) and \( w \) is there a path from \( v \) to \( w \)?

PageRank. What is the importance of a web page?
- digraph API
- digraph search
- topological sort
- strong components
## Digraph API

**public class** Digraph

<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>

```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

% java TestDigraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
4->3
4->2
5->4
6->9
6->4
6->0
...
11->4
11->12
12->9

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);
Maintain vertex-indexed array of lists (use Bag abstraction).

Adjacency-list digraph representation

Digraph input format and adjacency-lists representation
Adjacency-lists digraph representation: Java implementation

Same as graph, but only insert one copy of each edge.

```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];
    }
}
```
**Digraph representations**

**In practice.** Use adjacency-list 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 list</td>
<td>(E + V)</td>
<td>1</td>
<td>outdegree(v)</td>
<td>outdegree(v)</td>
</tr>
</tbody>
</table>

\(\dagger\) disallows parallel edges
- 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 adjacent from v.
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 marked(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)

Digraph version 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 marked(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
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.
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.
  • Transitive closure.

**Basis for solving difficult digraph problems.**
• 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.

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

---

**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.
Breadth-first search in digraphs application: web crawler

**Goal.** Crawl web, starting from some root web page, say www.princeton.edu.

**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

Queue<String> queue = new Queue<String>();
SET<String> visited = new SET<String>();

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

while (!q.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 (!visited.contains(w))
        {
            visited.add(w);
            queue.enqueue(w);
        }
    }
}
- digraph API
- digraph search
- topological sort
- strong components
**Goal.** Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

**Graph model.** vertex = task; edge = precedence constraint.

0. Algorithms  
1. Complexity Theory  
2. Artificial Intelligence  
3. Intro to CS  
4. Cryptography  
5. Scientific Computing  
6. Advanced Programming

(tasks)

(precedence constraint graph)

(feasible schedule (read up!))
Topological sort

**DAG.** Directed acyclic graph.

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

<table>
<thead>
<tr>
<th>Directed edges</th>
<th>DAG</th>
<th>Topological order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0→5</td>
<td>0→1</td>
<td>0→5</td>
</tr>
<tr>
<td>0→2</td>
<td>3→6</td>
<td>0→1</td>
</tr>
<tr>
<td>3→5</td>
<td>3→4</td>
<td>3→6</td>
</tr>
<tr>
<td>5→4</td>
<td>6→4</td>
<td>3→4</td>
</tr>
<tr>
<td>6→0</td>
<td>3→2</td>
<td>5→4</td>
</tr>
<tr>
<td>1→4</td>
<td>6→0</td>
<td>1→4</td>
</tr>
</tbody>
</table>

**Solution.** DFS. What else?
Depth-first search order

```java
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;
    }
}
```

returns all vertices in “reverse DFS postorder”
**Reverse DFS postorder in a DAG**

```
dfs(0)  1 0 0 0 0 0 0 -
dfs(1)  1 1 0 0 0 0 0 -
  dfs(4)  1 1 0 0 1 0 0 -
    4 done  1 1 0 0 1 0 0 4
    1 done  1 1 0 0 1 0 0 4 1
  dfs(2)  1 1 1 0 1 0 0 4 1
    2 done  1 1 1 0 1 0 0 4 1 2
  dfs(5)  1 1 1 0 1 1 0 4 1 2
    check 2  1 1 1 0 1 1 0 4 1 2
    5 done  1 1 1 0 1 1 0 4 1 2 5
  0 done  1 1 1 0 1 1 0 4 1 2 5 0
check 1  1 1 1 0 1 1 0 4 1 2 5 0
check 2  1 1 1 0 1 1 0 4 1 2 5 0
  dfs(3)  1 1 1 1 1 1 0 4 1 2 5 0
    check 2  1 1 1 1 1 1 0 4 1 2 5 0
    check 4  1 1 1 1 1 1 0 4 1 2 5 0
    check 5  1 1 1 1 1 1 0 4 1 2 5 0
  dfs(6)  1 1 1 1 1 1 1 4 1 2 5 0 6
    6 done  1 1 1 1 1 1 1 4 1 2 5 0 6
  3 done  1 1 1 1 1 1 1 4 1 2 5 0 6 3
check 4  1 1 1 1 1 1 0 4 1 2 5 0 6 3
check 5  1 1 1 1 1 1 0 4 1 2 5 0 6 3
check 6  1 1 1 1 1 1 0 4 1 2 5 0 6 3
  done  1 1 1 1 1 1 1 4 1 2 5 0 6 3
```

The reverse DFS postorder is a topological order.
Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge \( v \rightarrow w \). When \( \text{dfs}(G, v) \) is called:

- **Case 1:** \( \text{dfs}(G, w) \) has already been called and returned. Thus, \( w \) was done before \( v \).

- **Case 2:** \( \text{dfs}(G, w) \) has not yet been called. It will get called directly or indirectly by \( \text{dfs}(G, v) \) and will finish before \( \text{dfs}(G, v) \). Thus, \( w \) will be done before \( v \).

- **Case 3:** \( \text{dfs}(G, w) \) has already been called, but has not returned. Can’t happen in a DAG: function call stack contains path from \( w \) to \( v \), so \( v \rightarrow w \) would complete a cycle.
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.

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?

![Diagram of course table]

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!)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;=B1 + 1&quot;</td>
<td>&quot;=C1 + 1&quot;</td>
<td>&quot;=A1 + 1&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Microsoft Excel cannot calculate a formula. Cell references in the formula refer to the formula's result, creating a circular reference. Try one of the following:

- If you accidentally created the circular reference, click OK. This will display the Circular Reference toolbar and help for using it to correct your formula.
- To continue leaving the formula as it is, click Cancel.
Directed cycle detection application: symbolic links

The Linux file system does not do cycle detection.

```bash
% ln -s a.txt b.txt
% ln -s b.txt c.txt
% ln -s c.txt a.txt

% more a.txt
a.txt: Too many levels of symbolic links
```
Directed cycle detection application: WordNet

The WordNet database (occasionally) has cycles.
› digraph API
› digraph search
› topological sort
› strong components
Strongly-connected components

**Def.** Vertices $v$ and $w$ are strongly connected if there is a directed path from $v$ to $w$ and a directed path from $w$ to $v$.

**Key property.** Strong connectivity is an equivalence relation:
- $v$ is strongly connected to $v$.
- If $v$ is strongly connected to $w$, then $w$ is strongly connected to $v$.
- If $v$ is strongly connected to $w$ and $w$ to $x$, then $v$ is strongly connected to $x$.

**Def.** A strong component is a maximal subset of strongly-connected vertices.
**Connected components vs. strongly-connected components**

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

Connected component id (easy to compute with DFS)

<table>
<thead>
<tr>
<th>cc[]</th>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc[]</td>
<td>0 0 0 0 0 1 1 1 2 2 2 2</td>
</tr>
</tbody>
</table>

Strongly-connected component id (how to compute?)

<table>
<thead>
<tr>
<th>scc[]</th>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>scc[]</td>
<td>1 0 1 1 1 3 4 4 2 2 2 2</td>
</tr>
</tbody>
</table>

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

constant-time client connectivity query

public int stronglyConnected(int v, int w) {
    return scc[v] == scc[w];
}

constant-time client strong-connectivity query
Strong component application: ecological food webs

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

Strong component. Subset of species with common energy flow.
**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.
**Approach 2.** Use to improve design!
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).
• 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.
Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

- Run DFS on $G^R$ to compute reverse postorder.
- Run DFS on $G$, considering vertices in order given by first DFS.

DFS in reverse digraph (ReversePost)

$G^R$

check unmarked vertices in the order
0 1 2 3 4 5 6 7 8 9 10 11 12

dfs(0)
  | dfs(6)
  |   | dfs(7)
  |   |   | dfs(8)
  |   |   |   | check 7
  |   |   |   | 8 done
  |   |   | 7 done
  | 6 done
  | dfs(2)
  |   | dfs(4)
  |   |   | dfs(11)
  |   |   |   | dfs(9)
  |   |   |   |   | dfs(12)
  |   |   |   |   |   | check 11
  |   |   |   |   |   | ...

G

Run DFS on
ReversePost

to compute reverse postorder.
Simple (but mysterious) algorithm for computing strong components.

- Run DFS on $G^R$ to compute reverse postorder.
- Run DFS on $G$, considering vertices in order given by first DFS.

**Proposition.** Second DFS gives strong components. (!!)
Kosaraju proof of correctness

**Proposition.** Kosaraju’s algorithm computes strong components.

**Pf.** We show that the vertices marked during the constructor call `dfs(G, s)` are the vertices strongly connected to `s`.

`⇐ [If `t` is strongly connected to `s`, then `t` is marked during the call `dfs(G, s)`.]`

- There is a path from `s` to `t`, so `t` will be marked during `dfs(G, s)` unless `t` was previously marked.
- There is a path from `t` to `s`, so if `t` were previously marked, then `s` would be marked before `t` finishes (so `dfs(G, s)` would not have been called in constructor).
Kosaraju proof of correctness (continued)

Proposition. Kosaraju’s algorithm computes strong components.
⇒ [If \( t \) is marked during the call \( \text{dfs}(G, s) \), then \( t \) is strongly connected to \( s \).]

- Since \( t \) is marked during the call \( \text{dfs}(G, s) \), there is a path from \( s \) to \( t \) in \( G \) (or, equivalently, a path from \( t \) to \( s \) in \( G^R \)).
- Reverse postorder construction implies that \( t \) is done before \( s \) in \( \text{dfs} \) of \( G^R \).
- The only possibility for \( \text{dfs} \) in \( G^R \) implies there is a path from \( s \) to \( t \) in \( G^R \) (or, equivalently, a path from \( t \) to \( s \) in \( G \)).
Connected components in an undirected graph (with DFS)

```java
public class CC {
    private boolean marked[];
    private int[] id;
    private int count;

    public CC(Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];

        for (int v = 0; v < G.V(); v++)
        {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

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

    public boolean connected(int v, int w) {
        return id[v] == id[w];
    }
}
```
public class KosarajuSCC
{
    private boolean marked[];
    private int[] id;
    private int count;

    public KosarajuSCC(Digraph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
        for (int v : dfs.reversePost())
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

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

    public boolean stronglyConnected(int v, int w)
    {  return id[v] == id[w];  }
}
Digraph-processing summary: algorithms of the day

<table>
<thead>
<tr>
<th>single-source reachability</th>
<th>DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>topological sort (DAG)</td>
<td>DFS</td>
</tr>
<tr>
<td>strong components</td>
<td>Kosaraju DFS (twice)</td>
</tr>
</tbody>
</table>