4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
- strong components
4.2 Directed Graphs

- introduction
- 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 x5 is true, then x0 is true
Combinational circuit

Vertex = logical gate; edge = wire.
WordNet graph

Vertex = synset; edge = hypernym relationship.

http://wordnet.princeton.edu
## 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 a 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?
4.2 Directed Graphs

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

```java
public class Digraph {
    public Digraph(int V) { create an empty digraph with V vertices }
    public Digraph(In in) { create a digraph from input stream }
    void addEdge(int v, int w) { add a directed edge v→w }
    Iterable<Integer> adj(int v) { vertices pointing from v }
    int V() { number of vertices }
    int E() { number of edges }
    Digraph reverse() { reverse of this digraph }
    String toString() { string representation }
}
```

```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);
```

- `In in = new In(args[0]);` reads digraph from input stream
- `Digraph G = new Digraph(in);` creates a digraph from input stream
- `for (int v = 0; v < G.V(); v++)` iterates over all vertices
- `for (int w : G.adj(v))` iterates over all adjacent vertices
- `StdOut.println(v + "->" + w);` prints each edge (once)
Digraph API

```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);
```

% 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

In the diagram, V represents the vertices, and E represents the edges.

- `tinyDG.txt` is the input file containing the digraph.
- The program reads the digraph from the input stream.
- It prints out each edge once.
- The diagram illustrates the vertices and edges of the digraph according to the input file.
Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.
Adjacency-lists graph representation (review): 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 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];
    }
}
```
### Digraph representations

**In practice.** Use adjacency-lists representation.
- Algorithms based on iterating over vertices pointing 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 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>$V^2$</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
4.2 Directed Graphs

- introduction
- 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.
Depth-first search demo

To visit a vertex $v$:

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

![Directed Graph](image)
Depth-first search demo

To visit a vertex $v$:

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

![Graph with vertices and edges demonstrating depth-first search](image)

<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>
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
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.
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) 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 pointing from \( v \) and mark them.

**Graph G**

```
tinyDG2.txt
V 6
8 5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2
```
Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices pointing 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>
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. \( S = \{ 1, 7, 10 \} \).

- Shortest path to 4 is 7 → 6 → 4.
- Shortest path to 5 is 7 → 6 → 0 → 5.
- Shortest path to 12 is 10 → 12.
- ...

Q. How to implement multi-source shortest paths algorithm?
A. Use BFS, but initialize by enqueuing all source vertices.
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 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).

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

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+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);
    while (matcher.find())
    {
        String w = matcher.group();
        if (!marked.contains(w))
        {
            marked.add(w);
            queue.enqueue(w);
        }
    }
}
4.2 Directed Graphs

- Introduction
- 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.

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

**DAG.** Directed acyclic graph.

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

```
0→5  0→2
0→1  3→6
3→5  3→4
5→4  6→4
6→0  3→2
1→4
```

directed edges

**Solution.** DFS. What else?
Topological sort demo

• Run depth-first search.
• Return vertices in reverse postorder.
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
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”
Topological sort in a DAG: correctness proof

**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, $w$ was done before $v$.
- **Case 2:** $dfs(w)$ has not yet been called. $dfs(w)$ will get called directly or indirectly by $dfs(v)$ and will finish before $dfs(v)$. Thus, $w$ will be done before $v$.
- **Case 3:** $dfs(w)$ has already been called, but has not yet 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.

![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?

![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!)
4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
- strong components
**Strongly-connected components**

**Def.** Vertices $v$ and $w$ are **strongly connected** if there is both 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 both 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)

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

strongly-connected component id (how to compute?)

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

http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

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-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-Sharir 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.

Diagram:
- Digraph $G$ and its strong components.
- Kernel DAG of $G$ (in reverse topological order).

First vertex is a sink (has no edges pointing from it).
How to compute?
Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in $G^R$.
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$. 

digraph $G$
Kosaraju-Sharir algorithm demo

**Phase 1.** Compute reverse postorder in $G^R$.

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

reverse digraph $G^R$
Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$.

\begin{itemize}
\item 1 0 2 4 5 3 11 9 12 10 6 7 8
\end{itemize}

done
Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on $G^R$ to compute reverse postorder.
- Phase 2: run DFS on $G$, considering vertices in order given by first DFS.

DFS in reverse digraph $G^R$

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

reverse postorder for use in second dfs()
1 0 2 4 5 3 11 9 12 10 6 7 8
Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on $G^R$ to compute reverse postorder.
- Phase 2: run DFS on $G$, considering vertices in order given by first DFS.
Kosaraju-Sharir algorithm

**Proposition.** Kosaraju-Sharir algorithm computes the strong components of a digraph in time proportional to $E + V$.

**Pf.**
- Running time: bottleneck is running DFS twice (and computing $G^R$).
- Correctness: tricky, see textbook (2nd printing).
- Implementation: easy!
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);
    }

    public boolean connected(int v, int w) {
        return id[v] == id[w];
    }
}
```
Strong components in a digraph (with two DFSs)

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

    public KosarajuSharirSCC(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

- **single-source reachability in a digraph**
  - DFS

- **topological sort in a DAG**
  - DFS

- **strong components in a digraph**
  - Kosaraju-Sharir DFS (twice)