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.

![Diagram of directed graph with vertex and path annotations]
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.
WordNet graph

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>
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
4.2 Directed Graphs

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

```
public class Digraph

    Digraph(int V)  // create an empty digraph with V vertices
    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
```

```
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 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 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);

read digraph from input stream
print out each edge (once)
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 = 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];
    }
}
```
---

- **Adjacency lists**
- **Create empty digraph with V vertices**
- **Add edge v→w**
- **Iterator for vertices pointing from 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
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 $v$. 

<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.

```
tinyDG2.txt
V
6
8
5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2
```

Graph $G$
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\( \rightarrow \)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?
A. Use BFS, but initialize by enqueuing all source vertices.
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).

Q. Why not use DFS?
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";
quueues.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);
      }
   }
}
```

- 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 it and put on the queue
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
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

---

tasks

precedence constraint graph

feasible schedule
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→2  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.

a directed acyclic graph
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.

All vertices pointing from 3 are done before 3 is done, so they appear after 3 in topological order.
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?

![Table of courses and prerequisites](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 class 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>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></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.
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
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

A digraph and its strong components

A graph and its connected components

Connected component id (easy to compute with DFS)

<table>
<thead>
<tr>
<th>v</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

`cc[]`:

<table>
<thead>
<tr>
<th>v</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</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>v</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

`scc[]`:

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.
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 $G^R$.

$$
\begin{align*}
1 & \quad 0 & \quad 2 & \quad 4 & \quad 5 & \quad 3 & \quad 11 & \quad 9 & \quad 12 & \quad 10 & \quad 6 & \quad 7 & \quad 8
\end{align*}
$$

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

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

dfs(0)
  dfs(6)
    dfs(8)
      check 6
      8 done
dfs(7)
    7 done
dfs(4)
  dfs(2)
    dfs(11)
      dfs(12)
        check 11
dfs(10)
      check 9
dfs(9)
    10 done
dfs(3)
check 12
check 7
check 6
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.
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!
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];
    }
}
```
Single-source reachability in a digraph

Topological sort in a DAG

Strong components in a digraph

Kosaraju-Sharir DFS (twice)
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