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.

![Directed graph diagram with labeled vertices and edges](image)
Road network

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

**Political blogosphere graph**

Vertex = political blog; edge = link.
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

<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 all 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
- strong components
## Digraph API

Almost identical to Graph 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 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?

A. $V$

B. $E + V$

C. $V^2$

D. $VE$

E. *I don't know.*

```java
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 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
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];
    }
}
```
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 vertex v.
Recursively visit all unmarked vertices w adjacent from v.
To visit a vertex $v$:

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

A directed graph
Depth-first search demo

To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent from \( v \).

\[
\begin{array}{c|c|c}
\text{v} & \text{marked[]} & \text{edgeTo[]} \\
0 & T & - \\
1 & T & 0 \\
2 & T & 3 \\
3 & T & 4 \\
4 & T & 5 \\
5 & T & 0 \\
6 & F & - \\
7 & F & - \\
8 & F & - \\
9 & F & - \\
10 & F & - \\
11 & F & - \\
12 & F & - \\
\end{array}
\]
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.
[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 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→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?
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
E. I don't know.
### Web crawler output

#### BFS crawl

- http://www.princeton.edu
- http://www.w3.org
- http://ogp.me
- http://giving.princeton.edu
- http://www.princetonartmuseum.org
- http://www.gopricetontigers.com
- http://library.princeton.edu
- http://helpdesk.princeton.edu
- http://tigernet.princeton.edu
- http://alumni.princeton.edu
- http://gradschool.princeton.edu
- http://vimeo.com
- http://princetonusg.com
- http://artmuseum.princeton.edu
- http://jobs.princeton.edu
- http://odoc.princeton.edu
- http://blogs.princeton.edu
- http://www.facebook.com
- http://twitter.com
- http://www.youtube.com
- http://deimos.apple.com
- http://qeprize.org
- ...

#### DFS crawl

- http://www.princeton.edu
- http://deimos.apple.com
- http://www.youtube.com
- http://www.google.com
- http://news.google.com
- http://csi.gstatic.com
- http://googlenewsblog.blogspot.com
- http://labs.google.com
- http://groups.google.com
- http://img1.blogblog.com
- http://feeds.feedburner.com
- http://buttons.googlesyndication.com
- http://fusion.google.com
- http://insiderearch.blogspot.com
- http://agooleaday.com
- http://static.googleusercontent.com
- http://searchresearch1.blogspot.com
- http://feedburner.google.com
- http://www.dot.ca.gov
- http://www.laketahoe.com
- http://ethel.tahoeguide.com
- ...

---

32
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).

How many strong components are there in this digraph?
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+)";
    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
**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.

<table>
<thead>
<tr>
<th>tasks</th>
<th>precedence constraint graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. Algorithms</td>
<td></td>
</tr>
<tr>
<td>1. Complexity Theory</td>
<td></td>
</tr>
<tr>
<td>2. Artificial Intelligence</td>
<td></td>
</tr>
<tr>
<td>3. Intro to CS</td>
<td></td>
</tr>
<tr>
<td>4. Cryptography</td>
<td></td>
</tr>
<tr>
<td>5. Scientific Computing</td>
<td></td>
</tr>
<tr>
<td>6. Advanced Programming</td>
<td></td>
</tr>
</tbody>
</table>

feasible schedule
Topological sort

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

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

Solution. DFS. What else?
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;  }
}
Topological sort in a DAG: intuition

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.
- ...
**Proposition.** Reverse DFS postorder of a DAG is a topological order.

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

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

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

- **Case 3:** \( \text{dfs}(w) \) 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.
Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

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!)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></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>
</tr>
<tr>
<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.
Depth-first search orders

**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);
}
```
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.

![Diagram of a digraph with 5 strongly-connected components]
Directed graphs: quiz 3

How many strong components are in a DAG with $V$ vertices and $E$ edges?

A. 0
B. 1
C. $V$
D. $E$
E. I don't know.
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!
Connected components vs. strongly-connected components

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

![Connected components](image)

3 connected components

**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**

![Strongly-connected components](image)

5 strongly-connected components

connected component id (easy to compute with DFS)

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

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

constant-time client connectivity query

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

constant-time client strong-connectivity query
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.

![digraph G and its strong components](image1)

![kernel DAG of G (topological order: A B C D E)](image2)
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 $G^R$.

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

---

<table>
<thead>
<tr>
<th>$v$</th>
<th>id[]</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>
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
  6 done
dfs(2)  
dfs(4)  
dfs(11)
dfs(9)  
dfs(12)  
  check 11
dfs(10)  
  dfs(9)  |  check 9
  10 done
dfs(12)  
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
  dfs(12)
  check 11
dfs(10)  
  check 9
dfs(12)
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 (2$^{nd}$ 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.reversePostorder()) {
            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];
    }
}
```
<table>
<thead>
<tr>
<th>Digraph-processing summary: algorithms of the day</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>single-source reachability in a digraph</strong></td>
</tr>
<tr>
<td><img src="image1" alt="Graph" /></td>
</tr>
<tr>
<td>DFS</td>
</tr>
<tr>
<td><strong>topological sort in a DAG</strong></td>
</tr>
<tr>
<td><img src="image2" alt="Graph" /></td>
</tr>
<tr>
<td>DFS</td>
</tr>
<tr>
<td><strong>strong components in a digraph</strong></td>
</tr>
<tr>
<td><img src="image3" alt="Graph" /></td>
</tr>
<tr>
<td>Kosaraju–Sharir DFS (twice)</td>
</tr>
</tbody>
</table>