Mar. 31, 2015

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

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

Imagination graph

Vertex = variable; edge = logical implication.

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

Overnight interbank loan graph

Vertex = bank; edge = overnight loan.

Combinational circuit

Vertex = logical gate; edge = wire.
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

```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);
    adj[w].add(v);
  }

  public Iterable<Integer> adj(int v)
  {  return adj[v];  }

  public int V()
  {  return V;  }

  public int E()
  {  return adj[0].size();  }

  public Digraph reverse()
  {  return new Digraph(V);  }

  public String toString()
  {  return "Digraph with V = " + V + " vertices and E = " + E();  }
}
```

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.

Adjacency-lists graph representation: 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];  }

  public int V()
  {  return V;  }

  public int E()
  {  return adj[0].size();  }

  public Graph reverse()
  {  return new Graph(V);  }

  public String toString()
  {  return "Graph with V = " + V + " vertices and E = " + E();  }
}
```
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>( V )</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

Directed Graphs

› 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

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.

---

Depth-first search

**Depth-first search in a directed graph**

To visit a vertex v:
• Mark vertex v as visited.
• Recursively visit all unmarked vertices pointing from v.
Depth-first search

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>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>-</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>

visit 5

Depth-first search

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>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</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>

visit 4

Depth-first search

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>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>5</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>

visit 3

Depth-first search

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>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</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>

visit 3
To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices pointing from \( v \).
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.
To visit a vertex $v$:

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

### Depth-first search

<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

<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

<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

<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**

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

**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];
    }
}
```

**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];
    }
}
```

**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).

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.

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).

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) from s to all other vertices n a digraph in time proportional to \( E + V \).
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 constructor?
A. Use BFS, but initialize by enqueuing all source vertices.

Breadth-first search in digraphs application: web crawler

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

```java
Queue<String> queue = new Queue<String>();
SET<String> discovered = new SET<String>();
String root = "http://www.princeton.edu"
queue.enqueue(root);
discovered.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 (!discovered.contains(w)) {
            discovered.add(w);
            queue.enqueue(w);
        }
    }
}
```

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 undiscovered, mark it as discovered and put on queue

Directed Graphs

- 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?

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

**Topological sort**

**DAG.** Directed acyclic graph.

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

**Solution.** DFS. What else?

**Topological sort algorithm**

- Run depth-first search.
- Return vertices in reverse postorder.
Topological sort algorithm

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

![Graph with vertices 1, 4, 2, 3, 5, 6 and edges](image)

- Visit vertex 1.
- Postorder.

- Visit vertex 4.
- Postorder.

- Visit vertex 1.
- Postorder.
Topological sort algorithm

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

Topological sort algorithm

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

Topological sort algorithm

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

Topological sort algorithm

• Run depth-first search.
• Return vertices in reverse postorder.
Topological sort algorithm

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

### Step 1

1. Visit vertex 0.
2. Postorder: 0.

### Step 2

1. Visit vertex 5.
2. Postorder: 0, 5.

### Step 3

2. Postorder: 0, 5, 6.

### Step 4

1. Visit vertex 3.
2. Postorder: 0, 5, 6, 3.

### Step 5

1. Visit vertex 2.
2. Visit vertex 1.
3. Postorder: 0, 5, 6, 3, 2, 1.

### Step 6

2. Postorder: 0, 5, 6, 3, 2, 1, 4.

5 done
Topological sort algorithm

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

Topological sort algorithm

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

Topological sort algorithm

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

Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.
Topological sort algorithm

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

![Diagram of graph with vertices and arrows indicating depth-first search]

postorder

4 1 2 5 0

visit 6

6 done
Topological sort algorithm

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

Topological sort algorithm

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

Topological sort algorithm

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

Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.
Topological sort algorithm

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

```
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;
    }
}
```

Depth-first search order

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

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, $w$ was done before $v$.

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

- Case 3: $\text{dfs}(w)$ 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.

Ex:  
```java
dfs(0)  
dfs(1)  
1 done  
dfs(4)  
4 done  
dfs(2)  
2 done  
dfs(5)  
5 done  
0 done  
check 1  
check 2  
dfs(3)  
check 2  
check 4  
check 5  
dfs(6)  
6 done  
3 done  
```

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

Directed cycle detection

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

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

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

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

- Case 3: $\text{dfs}(w)$ 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.

Ex:  
```java
dfs(0)  
dfs(5)  
dfs(4)  
dfs(3)  
check 5  ```

```
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”

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?

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

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

public class B extends C {
    ...
}

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

Microsoft Excel does cycle detection (and has a circular reference toolbar!)

Directed cycle detection applications

- Causalities.
- Email loops.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Precedence scheduling.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.
**Directed Graphs**

- 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.

**Examples of strongly-connected digraphs**

**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 $v$

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

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

- 3 connected components
- 3 strongly-connected components
- 5 connected components
- 5 strongly-connected components

```
connected component id (easy to compute with DFS)
```

```
strongly-connected component id (how to compute?)
```

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

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

```
constant-time client connectivity query
```

```
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.

http://www.nasingroes.district96.k12.il.us/Wetlands/Salamander/SaGraphics/salfoodweb.gif

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’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.

How to compute?

digraph G and its strong components
kernel DAG of G (in reverse topological order)
Kosaraju's Algorithm

- DFS in reverse graph
- DFS in original graph

Kosaraju-Sharir

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

Kosaraju-Sharir

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

Kosaraju-Sharir

reverse digraph $G^R$

Kosaraju-Sharir

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

visit 0

marked[v]

<table>
<thead>
<tr>
<th>v</th>
<th>marked[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
</tr>
</tbody>
</table>
Phase 1. Compute reverse postorder in $G^R$. 

Kosaraju-Sharir

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

Kosaraju-Sharir

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

Kosaraju-Sharir

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

Kosaraju-Sharir
Phase 1. Compute reverse postorder in $G^R$.

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
T & F & F & F & F & F & T & T & F & F & F & F & F \\
\end{array}
\]

visit 7

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & T & T & T & F & F & F \\
\end{array}
\]

visit 0

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
7 & 8 & 0 & 1 & 2 & 3 & 4 & F & F & F & F & F & F \\
\end{array}
\]

visit 0

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
6 & 7 & 8 & 6 & 7 & 8 & 0 & 1 & 2 & 3 & 4 & F & F \\
\end{array}
\]

visit 0
Phase 1. Compute reverse postorder in $G^R$.

visit 2

visit 4

visit 11

visit 9
Phase 1. Compute reverse postorder in $G^R$.

**Visit 12**

Marked vertices:

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

**Visit 10**

Marked vertices:

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

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

12 done

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

visit 9

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

9 done

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

T F T F T F T T T T T
Phase 1. Compute reverse postorder in $G^R$.

1. Visit 4

2. 11 done

3. Visit 4

4. Visit 5
Phase 1. Compute reverse postorder in $G^R$.

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

visit 3

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

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

visit 3

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

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

visit 5

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

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

visit 5
Phase 1. Compute reverse postorder in $G^R$.

### Kosaraju-Sharir

#### Phase I

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

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

5 done

### Kosaraju-Sharir

#### Phase 1

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

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

4 done

### Kosaraju-Sharir

#### Phase 1

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

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

2 done

### Kosaraju-Sharir

#### Phase 1

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

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

2 done
Kosaraju-Sharir

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

$$\begin{array}{c|c}
 v & \text{marked}[v] \\
\hline
 0 & T \\
 1 & F \\
 2 & T \\
 3 & T \\
 4 & T \\
 5 & T \\
 6 & T \\
 7 & T \\
 8 & T \\
 9 & T \\
 10 & T \\
 11 & T \\
 12 & T \\
\end{array}$$

$0$ done

Kosaraju-Sharir

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

$$\begin{array}{c|c}
 v & \text{marked}[v] \\
\hline
 0 & T \\
 1 & T \\
 2 & T \\
 3 & T \\
 4 & T \\
 5 & T \\
 6 & T \\
 7 & T \\
 8 & T \\
 9 & T \\
 10 & T \\
 11 & T \\
 12 & T \\
\end{array}$$

visit $1$

Kosaraju-Sharir

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

$$\begin{array}{c|c}
 v & \text{marked}[v] \\
\hline
 0 & T \\
 1 & T \\
 2 & T \\
 3 & T \\
 4 & T \\
 5 & T \\
 6 & T \\
 7 & T \\
 8 & T \\
 9 & T \\
 10 & T \\
 11 & T \\
 12 & T \\
\end{array}$$

check $2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12$

Kosaraju-Sharir

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

$$\begin{array}{c|c}
 v & \text{marked}[v] \\
\hline
 0 & T \\
 1 & T \\
 2 & T \\
 3 & T \\
 4 & T \\
 5 & T \\
 6 & T \\
 7 & T \\
 8 & T \\
 9 & T \\
 10 & T \\
 11 & T \\
 12 & T \\
\end{array}$$

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

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

**Kosaraju-Sharir**

**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.

**Kosaraju-Sharir**

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

**Kosaraju's algorithm**

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
dfs(0)
dfs(6)
dfs(8)
done

check 6
dfs(7)
done
dfs(2)
dfs(4)
dfs(11)
dfs(9)
dfs(12)
done

check 11
dfs(10)
done
dfs(7)
done

check 6
dfs(5)
done
dfs(3)
done

check 4
done
check 2
done
check 0
done
```

DFS in original digraph $G$

```
v   scc(v)
0   -
1   -
2   -
3   -
4   -
5   -
6   -
7   -
8   -
9   -
10  -
11  -
12  -
```

**Kosaraju's algorithm**

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
dfs(0)
dfs(6)
dfs(8)
done

check 6
dfs(7)
done
dfs(2)
dfs(4)
dfs(11)
dfs(9)
dfs(12)
done

check 11
dfs(10)
done
dfs(7)
done

check 6
dfs(5)
done
dfs(3)
done

check 4
done
check 2
done
check 0
done
```

**Kosaraju-Sharir**

DFS in original digraph $G$

```
v   scc(v)
0   -
1   -
2   -
3   -
4   -
5   -
6   -
7   -
8   -
9   -
10  -
11  -
12  -
```
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$. 

Visit 1: 

$$
\begin{array}{c|c}
 v & scc(v) \\
\hline
 0 & - \\
 1 & 0 \\
 2 & - \\
 3 & - \\
 4 & - \\
 5 & - \\
 6 & - \\
 7 & - \\
 8 & - \\
 9 & - \\
 10 & - \\
 11 & - \\
 12 & - \\
\end{array}
$$

Visit 0: 

$$
\begin{array}{c|c}
 v & scc(v) \\
\hline
 0 & 1 \\
 1 & 0 \\
 2 & - \\
 3 & - \\
 4 & - \\
 5 & - \\
 6 & - \\
 7 & - \\
 8 & - \\
 9 & - \\
 10 & - \\
 11 & - \\
 12 & - \\
\end{array}
$$

Strong component: 1

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

1 done
Kosaraju-Sharir

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

Kosaraju-Sharir

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

Kosaraju-Sharir

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

Kosaraju-Sharir

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

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

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

visit 2

Kosaraju-Sharir

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

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

visit 2

Kosaraju-Sharir

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

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

2 done

Kosaraju-Sharir

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

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

Kosaraju-Sharir

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

4 done

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

Kosaraju-Sharir

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

5 done

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

Kosaraju-Sharir

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

visit 0

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$. 

Kosaraju-Sharir

0 done

Kosaraju-Sharir

check 2

Kosaraju-Sharir

strong component: 0 2 3 4 5

Kosaraju-Sharir

check 4
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$.  

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

Kosaraju-Sharir

Kosaraju-Sharir

Kosaraju-Sharir

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

<table>
<thead>
<tr>
<th>$v$</th>
<th>$scc[v]$</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>

visit 12
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$. 

v | scc[v]
---|---
0 | 1
1 | 0
2 | 1
3 | 1
4 | 1
5 | 1
6 | -
7 | -
8 | -
9 | 2
10 | 2
11 | 2
12 | 2

10 done

v | scc[v]
---|---
0 | 1
1 | 0
2 | 1
3 | 1
4 | 1
5 | 1
6 | -
7 | -
8 | -
9 | 2
10 | 2
11 | 2
12 | 2

9 done

Kosaraju-Sharir

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

v | scc[v]
---|---
0 | 1
1 | 0
2 | 1
3 | 1
4 | 1
5 | 1
6 | -
7 | -
8 | -
9 | 2
10 | 2
11 | 2
12 | 2

12 done

v | scc[v]
---|---
0 | 1
1 | 0
2 | 1
3 | 1
4 | 1
5 | 1
6 | -
7 | -
8 | -
9 | 2
10 | 2
11 | 2
12 | 2

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

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

Kosaraju-Sharir

strong component: 9 10 11 12

check 9

Kosaraju-Sharir

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

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

Kosaraju-Sharir

check 10

Kosaraju-Sharir

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

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

Kosaraju-Sharir

check 12

Kosaraju-Sharir

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

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

Kosaraju-Sharir

check 12
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$. 

<table>
<thead>
<tr>
<th>vertex</th>
<th>$\text{scc}(v)$</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>-</td>
</tr>
<tr>
<td>8</td>
<td>-</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>

visit 6
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$. 

**Kosaraju-Sharir**


done

6 done

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

<table>
<thead>
<tr>
<th>v</th>
<th>scc[v]</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>

visit 7

Kosaraju-Sharir

strong component: 7
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$.

Kosaraju-Sharir

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. (!!)
Strong components in a digraph (with two DFSs)

```java
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>