4.2 DIRECTED GRAPHS

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

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

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

Vertex = intersection; edge = one-way street.
Taxi flow patterns (Uber)

http://blog.uber.com/2012/01/09/uberdata-san-franciscomics/

Uber cab service

- Left Digraph: Color is the source neighborhood (no arrows).
- Right Plot: Digraph analysis shows financial districts have similar demand.
Reverse engineering criminal organizations (LogAnalysis)

“The analysis of reports supplied by mobile phone service providers makes it possible to reconstruct the network of relationships among individuals, such as in the context of criminal organizations. It is possible, in other terms, to unveil the existence of criminal networks, sometimes called rings, identifying actors within the network together with their roles” — Cantanese et. al

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMEI</td>
<td>IMEI code MS</td>
</tr>
<tr>
<td>called</td>
<td>called user</td>
</tr>
<tr>
<td>calling</td>
<td>calling user</td>
</tr>
<tr>
<td>date/time start</td>
<td>date/time start calling (GMT)</td>
</tr>
<tr>
<td>date/time end</td>
<td>date/time end calling (GMT)</td>
</tr>
<tr>
<td>type</td>
<td>sms, mms, voice, data etc.</td>
</tr>
<tr>
<td>IMSI</td>
<td>calling or called SIM card</td>
</tr>
<tr>
<td>CGI</td>
<td>Lat. long. BTS company</td>
</tr>
</tbody>
</table>

Table 1 An example of the structure of a log file.
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
Digraph API

```java
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
```

```java
In in = new In(args[0]);
Digraph G = new Digraph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

read digraph from input stream
print out each edge (once)
**Digraph API**

In in = new In(args[0]);
Digraph G = new Digraph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);

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

read digraph from input stream
print out each edge (once)
Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.
Do you slumber?

Suppose we are given an arbitrary Digraph $G$ and a path of length $V$ given by int[] $P$.

Q: What is the worst case run time to check validity of a path $P$ for a general graph with $E$ edges and $V$ vertices?

A. $E$ [41138]  
B. $V$ [41142]  
C. $EV$ [41146]  
D. $E+V$ [41182]
Do you slumber?

Suppose we are given an arbitrary Digraph $G$ and a path of length $V$ given by $\text{int}[] \ P$.

Q: What is the worst case run time to check validity of a path $P$ for a general graph with $V$ vertices?

A. $1$  
B. $V$  
C. $V^2$
Adjacency-lists graph representation (review): Java implementation

```java
class Graph {
    private final int V;
    private final Bag<Integer>[] adj;

    public Graph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
Adjacency-lists digraph representation: Java implementation

```java
public class Digraph {
    private final int V;
    private final Bag<Integer>[] adj;

    public Digraph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
    }

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

**In practice.** Use adjacency-lists representation.
- Algorithms based on iterating over vertices pointing from $v$.
- Real-world digraphs tend to be sparse.

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from $v$ to $w$</th>
<th>edge from $v$ to $w$?</th>
<th>iterate over vertices pointing from $v$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>$E$</td>
<td>1</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>$1^\dagger$</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>

$^\dagger$ disallows parallel edges
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.

Difficulty level.
  • Exactly the same problem for computers.
  • Harder for humans than undirected graphs.
    - Edge interpretation is context dependent!
The man-machine

Difficulty level.

- Exactly the same problem for computers.
- Harder for humans than undirected graphs.
  - Edge interpretation is context dependent!
Depth-first search demo

To visit a vertex \( v \):

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

A directed graph

To visit a vertex \( v \):
To visit a vertex $v$:

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

### Depth-first search demo

![Graph](image)

- **Marked vertices:**
  - $0$, $1$, $2$, $3$, $4$, $5$, $6$, $7$, $8$, $9$, $10$, $11$, $12$

- **Visit Order:**
  - $0$, $2$, $3$, $1$, $4$, $5$, $6$, $7$, $8$, $9$, $10$, $11$, $12$

- **Edge-to-Vehicle:**
  - $v$ | marked | 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     | -     |
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.
- Cow.java:5: unreachable statement

Infinite-loop detection.
Determine whether exit is unreachable.
- Trivial?
- Doable by student?
- Doable by expert?
- Intractable?
- Unknown?
- Impossible?
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.

Proposition. BFS computes shortest paths (fewest number of edges) from $s$ to all other vertices in a digraph in time proportional to $E + V$. 

**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.
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";
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
BFS Webcrawler Output

http://www.princeton.edu
http://www.w3.org
http://ogp.me
http://giving.princeton.edu
http://www.princetonartmuseum.org
http://www.goprincetontigers.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://princeton.usg.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
http://en.wikipedia.org
...

DFS Webcrawler Output

http://www.princeton.edu
http://deimos.apple.com [dead end]
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://insidesearch.blogspot.com
http://agoogleday.com

http://static.googleusercontent.com
http://searchresearch1.blogspot.com
http://feedburner.google.com
http://www.dot.ca.gov
http://www.getacross80.com
http://www.TahoeRoads.com
http://www.LakeTahoeTransit.com
http://www.laketahoe.com
http://ethel.tahoeguide.com
...

4.2 Directed Graphs

- introduction
- 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.

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

DAG. Directed acyclic graph.

Topological sort. Redraw DAG so all edges point upwards.

Solution. DFS. What else?

directed edges

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

DAG

topological order
Topological sort demo

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

A directed acyclic graph

0 → 5
0 → 2
0 → 1
3 → 6
3 → 5
3 → 4
5 → 4
6 → 4
6 → 0
3 → 2
1 → 4
Topological sort intuitive proof

- Run depth-first search.
- Return vertices in reverse postorder.
- Why does it work?
  - Last item in postorder has indegree 0. Good starting point.
  - Second to last can only be pointed to by last item. Good follow-up.
  - ...

See book / online slides for foolproof full proof.
Topological sort demo

Q: Is the reverse postorder the only valid topological order for this graph?
A. No [452392]
B. Yes [452393]
Topological sort demo

postorder
4 1 2 5 0 6 3

topological order
3 6 0 5 2 1 4

Q: Is the reverse postorder the only valid topological order for this graph?
A. No [452392]

Example: Could move 1 down one step. 0 → 1 still points up.
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”
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: 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!)

- In cells A1, B1, and C1, the formulas are set to:
  - A1: =B1 + 1
  - B1: =C1 + 1
  - C1: =A1 + 1

When these formulas are applied, Microsoft Excel cannot calculate the formula due to a circular reference. The error message indicates that the cell references in the formula refer to the formula's result, creating a circular reference. The message suggests 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
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 $v$. Every node is strongly connected to itself.

**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 graph with labeled vertices and arrows showing the strong components shaded.]
Examples of strongly-connected digraphs: 1 strong component
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 \( v \). Every node is strongly connected to itself.

Q: How many strong components does a DAG on \( V \) vertices and \( E \) edges have?

A. 0  \([452453]\)  
B. 1  \([452459]\)  
C. \( E \)  \([452460]\)  
D. \( V \)  \([452461]\)
Connected components vs. strongly-connected components

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

![Connected Components Diagram](image1)

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 Diagram](image2)

5 strongly-connected components

connected component id (easy to compute with DFS)

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

strongly-connected component id (how to compute?)

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

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

constant-time client connectivity query

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

constant-time client strong-connectivity query
Strongly connected components

Analysis of Yahoo Answers

- Edge is from asker to answerer.
- “A large SCC indicates the presence of a community where many users interact, directly or indirectly.”

<table>
<thead>
<tr>
<th>Category</th>
<th>Nodes</th>
<th>Edges</th>
<th>Avg. deg.</th>
<th>Mutual edges</th>
<th>SCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrestling</td>
<td>9,959</td>
<td>56,859</td>
<td>7.02</td>
<td>1,898</td>
<td>13.5%</td>
</tr>
<tr>
<td>Program.</td>
<td>12,538</td>
<td>18,311</td>
<td>1.48</td>
<td>0</td>
<td>0.01%</td>
</tr>
<tr>
<td>Marriage</td>
<td>45,090</td>
<td>164,887</td>
<td>3.37</td>
<td>179</td>
<td>4.73%</td>
</tr>
</tbody>
</table>

Strongly connected components

Understanding biological control systems

- *Bacillus subtilis* spore formation control network.
- SCC constitutes a functional module.
Strong components algorithms: brief history

1960s: Core OR problem.
- Widely studied; some practical algorithms.
- Complexity not understood.

1972: linear-time one-pass 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: easier one-pass linear-time algorithms.
- Gabow: fixed old OR algorithm.
- Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.
Intuitive solution to finding strongly connected components.
Intuitive solution to finding strongly connected components.

Example

Run DFS(1), get the SCC: \{1\}.
Run DFS(0), get \{0, 1, 2, 3, 4, 5\} - not an SCC.
Run DFS(1), then DFS(0), get SCC \{1\} and SCC \{0, 2, 3, 4, 5\}.
Intuitive solution to finding strongly connected components.

Q: Which DFS call should come next?
A. DFS(7) [397963]
B. DFS(6) or DFS(8) [398061]
C. DFS(9), DFS(10), DFS(11), or DFS(12) [398062]
Intuitive solution to finding strongly connected components.

Example

Run DFS(1), get the SCC: \{1\}.
Run DFS(0), get \{0, 1, 2, 3, 4, 5\} - not an SCC.
Run DFS(1), then DFS(0), get SCC \{1\} and SCC \{0, 2, 3, 4, 5\}.

Punchline. A Magic Sequence of DFS calls yields SCC (MSDFSSCC)
Intuitive solution to finding strongly connected components.

**DFS.** Calling DFS wantonly is a problem. Never want to leave your SCC.

**Starting SCCs.** There’s always some set of SCCs with outdegree 0, e.g. \{1\}. Calling DFS on any node in these SCCs finds the SCC.

**DFS Order.** After calling DFS on all starting SCCs, there’s at least one SCC that only points at the starting SCCs.

digraph G and its strong components

Treat SCCs as one big node. Kernel DAG. Arrows only connect SCCs. Graph is acyclic.
Kosaraju-Sharir algorithm: intuitive example

**Kernel DAG.** Topological sort of kernelDAG(G) is A, B, C, D, E.

**MSDFSSCC.** Call DFS on element from E, D, C, B, A. Valid MSDFSSCC. For example, DFS(1), DFS(2), DFS(9), DFS(6), DFS(7).

**Summary.**
- The MSDFSSCC is given by reverse of the topological sort of kernelDAG(G).
Kosaraju-Sharir algorithm: intuition (general)

Kernel DAG. MSDFSSCC is given by reverse of topological sort of kernelDAG(G).

Reverse Graph Lemma. Reverse of topological sort of kernelDAG(G) is given by reverse postorder of $G^R$ (see book), where $G^R$ is $G$ with all arrows flipped around.

Punchline.
- MSDFSSCC: The reverse postorder of $G^R$. 

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

![kernel DAG of G (in reverse topological order)](image2)

first vertex is a sink (has no edges pointing from it)
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

---

**Graph Representation**

![Graph Diagram]

<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: intuition

The Kosaraju-Sharir algorithm is used to determine the strongly connected components (SCCs) of a directed graph (digraph). The intuition behind the algorithm involves finding the kernel DAG of the graph, which is the digraph in reverse topological order. The first vertex in the kernel DAG is a sink (has no edges pointing from it).

The algorithm consists of two main steps:
1. **First DFS (Depth-First Search):** Perform a DFS on the original digraph to compute a reverse topological ordering.
2. **Second DFS (Reverse):** Perform a DFS on the reverse digraph (where the direction of the edges is reversed) using the reverse topological order from step 1 to find the SCCs.

The kernel DAG of the graph is shown in the figure, with vertices labeled from 0 to 12, and edges indicating the reverse topological order. The first vertex in the kernel DAG is a sink.
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.
Simple (but mysterious) algorithm for computing strong components.

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

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

**Pf.**
- Running time: bottleneck is running DFS twice (and computing $G^R$).
- Correctness: tricky, see textbook (2nd printing).
- Implementation: easy!
Connected components in an undirected graph (with DFS)

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

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

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

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

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

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

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

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

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

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

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

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