4.2 Directed Graphs

- digraph API
- digraph search
- transitive closure
- topological sort
- strong components

References: Algorithms in Java, 3rd edition, Chapter 19
Directed graphs

**Digraph.** Set of vertices connected pairwise by **oriented** edges.
Data from the blogosphere. Shown is a link structure within a community of political blogs (from 2004), where red nodes indicate conservative blogs, and blue liberal. Orange links go from liberal to conservative, and purple ones from conservative to liberal. The size of each blog reflects the number of other blogs that link to it. [Reproduced from (8) with permission from the Association for Computing Machinery]
Web graph

Vertex = web page.
Edge = hyperlink.
Ecological food web graph

Vertex = species.
Edge: from producer to consumer.
**WordNet graph**

Vertex = synset.
Edge = hypernym relationship.
# Digraph applications

<table>
<thead>
<tr>
<th>graph</th>
<th>vertex</th>
<th>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>stock, currency</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 and t?

**Strong connectivity.** Are all vertices mutually reachable?

**Transitive closure.** For which vertices v and w is there a path from v to w?

**Topological sort.** Can you draw the digraph so that all edges point from left to right?

**Precedence scheduling.** Given a set of tasks with precedence constraints, how can we best complete them all?

**PageRank.** What is the importance of a web page?
digraph API
- digraph search
- topological sort
- transitive closure
- strong components
**Digraph API**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Digraph(int V)</code></td>
<td>create an empty digraph with V vertices</td>
</tr>
<tr>
<td><code>Digraph(In in)</code></td>
<td>create a digraph from input stream</td>
</tr>
<tr>
<td><code>void addEdge(int v, int w)</code></td>
<td>add an edge from v to w</td>
</tr>
<tr>
<td><code>Iterable&lt;Integer&gt; adj(int v)</code></td>
<td>return an iterator over the neighbors of v</td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>return number of vertices</td>
</tr>
<tr>
<td><code>int E()</code></td>
<td>return number of edges</td>
</tr>
<tr>
<td><code>Digraph reverse()</code></td>
<td>return reverse of this digraph (all edges reversed)</td>
</tr>
</tbody>
</table>

```java
In in = new In();
Digraph G = new Digraph(in);
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        /* process edge v→w */
```
Set of edges representation

Store a list of the edges (linked list or array).
Adjacency-matrix representation

Maintain a two-dimensional \( V \)-by-\( V \) boolean array; for each edge \( v \to w \) in the digraph: \( \text{adj}[v][w] = \text{true} \).

Note: parallel edges disallowed
Adjacency-list representation

Maintain vertex-indexed array of lists (use Bag abstraction).

same as undirected graph, but one entry for each edge
Same as graph, but only insert one copy of each edge.

```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-list representation.

- Algorithms all based on iterating over edges incident to \( v \).
- 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 edges leaving ( v )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>( E )</td>
<td>( 1 ) *</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V^2 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency list</td>
<td>( E + V )</td>
<td>( 1 ) *</td>
<td>outdegree(( v ))</td>
<td>outdegree(( v ))</td>
</tr>
<tr>
<td>adjacency set</td>
<td>( E + V )</td>
<td>log (outdegree(( v )))</td>
<td>log (outdegree(( v )))</td>
<td>outdegree(( v ))</td>
</tr>
</tbody>
</table>

- huge number of vertices, small average vertex degree
- * only if parallel edges allowed
- digraph API
- digraph search
- transitive closure
- 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.
• Happens to have edges in both directions.
• DFS is a digraph algorithm.

**DFS (to visit a vertex s)**

Mark s as visited.
Recursively visit all unmarked vertices w adjacent to s.

---

![Graph Diagram](image-url)
Depth-first search (single-source reachability)

Identical to undirected version (substitute Digraph for Graph).

```java
public class DFSearcher {
    private boolean[] marked;

    public DFSearcher(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 connected to s
- constructor marks vertices connected to s
- recursive DFS does the work
- client can ask whether any vertex is reachable from s
Depth-first-search (pathfinding) for undirected graphs [from Lecture 12]

```java
public class PathfinderDFS {
    private Integer[] edgeTo;

    public PathfinderDFS(Graph G, int s) {
        edgeTo = new Integer[G.V()];
        edgeTo[s] = s;
        dfs(G, s);
    }

    private void dfs(Graph G, int v) {
        for (int w : G.adj(v))
            if (edgeTo[w] == null) {
                edgeTo[w] = v;
                dfs(G, w);
            }
    }

    public Iterable<Integer> pathTo(int v) {
        // Stay tuned.
    }
}
```

- replace `marked[]` with instance variable for parent-link representation of DFS tree
- initialize it in the constructor with Integer, all values are initially null
- not yet visited
- set parent link
- add method for client to iterate through path
Depth-first-search (pathfinding) for undirected graphs [slightly different version]

```java
public class PathfinderDFS {
    private int s;
    private boolean[] marked;
    private int[] edgeTo;

    public PathfinderDFS(Graph G, int s) {
        edgeTo = new int[G.V()];
        marked = new boolean[G.V()];
        this.s = s;
        dfs(G, s);
    }

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

    public boolean hasPathTo(int v) {
        return marked[v];
    }

    public Iterable<Integer> pathTo(int v) {
        // Stay tuned.
    }
}
```
Depth-first-search (pathfinding) for digraphs

```java
public class PathfinderDFS {
    private int s;
    private boolean[] marked;
    private int[] edgeTo;
    
    public PathfinderDFS(Digraph G, int s) {
        edgeTo = new int[G.V()];
        marked = new boolean[G.V()];
        this.s = s;
        dfs(G, s);
    }
    
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
                {
                    edgeTo[w] = v;
                    dfs(G, w);
                }
    }
    
    public boolean hasPathTo(int v) {
        return marked[v];
    }
    
    public Iterable<Integer> pathTo(int v) {
        // Stay tuned.
    }
}
```

- **add instance variable for parent-link representation of DFS tree**
- **initialize it in the constructor**
- **remember source (for pathTo())**
- **not yet visited**
- **set parent link**
- **method for client to test whether path exists**
- **method for client to iterate through path**
DFS pathfinding trace in a digraph

- dfs(0)
- dfs(5)
- dfs(4)
- dfs(3)
- check 5
- dfs(2)
- check 0
- check 3
- 2 done
- 3 done
- check 2
- 4 done
- 5 done
- dfs(1)
- 1 done
- 0 done

<table>
<thead>
<tr>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 ...</td>
<td>0 1 2 3 4 5 ...</td>
</tr>
<tr>
<td>1 0 0 0 0 0</td>
<td>-- -- -- -- 0</td>
</tr>
<tr>
<td>1 0 0 0 0 1</td>
<td>-- -- -- 5 0</td>
</tr>
<tr>
<td>1 0 0 0 1 1</td>
<td>-- -- 4 5 0</td>
</tr>
<tr>
<td>1 0 0 1 1 1</td>
<td>-- -- 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>-- 3 4 5 0</td>
</tr>
</tbody>
</table>

Order depends on digraph representation
Depth-first-search (pathfinding iterator) [from lecture 12]

edgeTo[] is a parent-link representation of a tree rooted at $s$

```
public Iterable<Integer> pathTo(int v)
{
    Stack<Integer> path = new Stack<Integer>();
    path.push(v)
    while (v != edgeTo[v])
    {
       v = edgeTo[v];
       path.push(v);
    }
    return path;
}
```

edgeTo[v]  -  2  6  4  7  3  0  2
v      0  1  2  3  4  5  6  7

![Graph diagram]
Depth-first-search (pathfinding iterator) [slightly different version]

edgeTo[] is a parent-link representation of a tree rooted at s

```java
public Iterable<Integer> pathTo(int v)
{
    if (hasPathTo(v)) return path(s, v);
    else              return null;
}

private Stack path(int s, int v)
{
    // Find path to v from any ancestor s in tree.
    Stack<Integer> stack = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
        stack.push(x);
    stack.push(s);
    return stack;
}
```

edgeTo[v] is a parent-link representation of a tree rooted at s

0 1 2 3 4 5 6 7

edgeTo[v] - 2 6 4 7 3 0 2

0 1 2 3 4 5 6 7
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 (DFS)

DFS enables direct solution of simple digraph problems.
✓• Reachability.
  • Cycle detection.
  • Topological sort.
  • Transitive closure.

Basis for solving difficult digraph problems.
• Directed Euler path.
• Strong connected components.
Breadth-first search in digraphs

Every undirected graph is a digraph.
- Happens to have edges in both directions.
- BFS is a **digraph** algorithm.

**BFS (from source vertex s)**

Put s onto a FIFO queue.
Repeat until the queue is empty:
- remove the least recently added vertex v
- add each of v’s unvisited neighbors to the queue and mark them as visited.

**Property.** Visits vertices in increasing distance from s.
Digraph BFS application: web crawler

**Goal.** Crawl web, starting from some root web page, say www.princeton.edu.

**Solution.** BFS with implicit graph.

**BFS.**
- Start at some root web page.
- 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?
Web crawler: BFS-based Java implementation

```java
Queue<String> q = new Queue<String>();
SET<String> visited = new SET<String>();

String s = "http://www.princeton.edu";
q.enqueue(s);
visited.add(s);

while (!q.isEmpty())
{
    String v = q.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 (!visited.contains(w))
        {
            visited.add(w);
            q.enqueue(w);
        }
    }
}
```

- queue of websites to crawl
- set of visited websites
- start crawling from website s
- read in raw html for next website in queue
- use regular expression to find all URLs in website of form http://xxx.yyy.zzz
- if unvisited, mark as visited and put on queue
- digraph API
- digraph search
- transitive closure
- topological sort
- strong components
**Graph-processing challenge (revisited)**

**Problem.** Is there an undirected path between $v$ and $w$?

**Goals.** Linear preprocessing time, constant query time.

**How difficult?**

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.
**Problem.** Is there a directed path from \( v \) to \( w \)?

**Goals.** Linear preprocessing time, constant query time.

**How difficult?**

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- 
  ✓ Impossible.

\( \text{can't do better than } V^2 \) (reduction from boolean matrix multiplication)
Transitive closure

**Def.** The transitive closure of a digraph $G$ is another digraph with a directed edge from $v$ to $w$ if there is a directed path from $v$ to $w$ in $G$.

*digraph $G$*

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*transitive closure $TC(G)$*

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*digraph $G$ is usually sparse*

*TC($G$) is usually dense*
Digraph-processing challenge 1 (revised)

**Problem.** Is there a directed path from v to w?

**Goals.** \( \sim V^2 \) preprocessing time, constant query time.

**How difficult?**

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- **No one knows.** open research problem
- Impossible.
**Problem.** Is there a directed path from v to w?

**Goals.** ~ V E preprocessing time, ~ V² space, constant query time.

**How difficult?**
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Use DFS once for each vertex to compute rows of transitive closure.

Useful edges:

\[
\begin{align*}
0 \to 1 & \quad 0 \to 6 \\
0 \to 2 & \quad 3 \to 4 \\
3 \to 2 & \quad 5 \to 4 \\
5 \to 0 & \quad 3 \to 5 \\
2 \to 1 & \quad 6 \to 4 \\
3 \to 1 &
\end{align*}
\]
Use an array of `DFSearcher` objects, one for each row of transitive closure.

```java
public class TransitiveClosure {
    private DFSearcher[] tc;

    public TransitiveClosure(Digraph G) {
        tc = new DFSearcher[G.V()];
        for (int v = 0; v < G.V(); v++)
            tc[v] = new DFSearcher(G, v);
    }

    public boolean reachable(int v, int w) {
        return tc[v].visited(w);
    }
}
```

Similar approach (array of `PathFinderDFS` objects) can provide paths.

**Warning:** Not for use with huge graphs (~$V^2$ space)
› digraph API
› digraph search
› transitive closure
› **DAGs**
› strong components
Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

Graph model.

- Create a vertex \( v \) for each task.
- Create an edge \( v \rightarrow w \) if task \( v \) must precede task \( w \).

Digraph application: scheduling
Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

No solution iff digraph has a directed cycle

First problem. Make sure digraph has no cycles.
Digraph-processing challenge 2a

Problem. Check that a digraph is a DAG.

Goal. Linear time.

How difficult?

• Any COS 126 student could do it.
✓ • Need to be a typical diligent COS 226 student.
• Hire an expert.
• Intractable.
• No one knows.
• Impossible.

use DFS

0 → 1
0 → 6
0 → 2
0 → 5
2 → 3
4 → 9
6 → 4
6 → 9
7 → 6
8 → 7
9 → 10
9 → 11
9 → 12
11 → 12
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.
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
Cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)
Cycle detection application: symbolic links

The Linux file system does not do cycle detection.

```bash
% ln -s a.txt b.txt
% ln -s b.txt c.txt
% ln -s c.txt a.txt

% more a.txt
a.txt: Too many levels of symbolic links
```
Finding a cycle in a digraph: Java implementation

```java
public class DigraphCycleFinder {
    private boolean[] marked;
    private int[] edgeTo;
    private Stack<Integer> cycle;
    private boolean[] onStack;

    public DigraphCycleFinder(Digraph G) {
        onStack = new boolean[G.V()];
        edgeTo = new int[G.V()];
        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) { // See next slide
        onStack[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
        onStack[v] = false;
        if (!cycle.isEmpty()) cycle.push(v);
        edgeTo[v] = null;
    }

    public boolean isDAG() { return cycle == null; }
    public Iterable<Integer> cycle() { return cycle; }
}
```
Finding a cycle in a digraph: Java implementation (continued)

```java
private void dfs(Digraph G, int v)
{
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v))
        if (cycle != null) return;
    else if (!marked[w])
        { edgeTo[w] = v; dfs(G, w); }
    else if (onStack[w])
        cycle = path(w, v);
    onStack[v] = false;
}

private Stack path(int s, int v)
// Same as for PathfinderDFS.
```

- done if cycle found
- w must be ancestor of v in edgeTo[]
Finding a cycle in a digraph

```
dfs(0)
dfs(5)
dfs(4)
dfs(3)
check 5
```

<table>
<thead>
<tr>
<th>marked[]</th>
<th>edgeTo[]</th>
<th>onStack[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 ...</td>
<td>0 1 2 3 4 5 ...</td>
<td>0 1 2 3 4 5 ...</td>
</tr>
<tr>
<td>1 0 0 0 0 0</td>
<td>- - - - - 0</td>
<td>1 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 0 1</td>
<td>- - - 5 0</td>
<td>1 0 0 0 0 1</td>
</tr>
<tr>
<td>1 0 0 0 1 1</td>
<td>- - - 4 5 0</td>
<td>1 0 0 0 1 1</td>
</tr>
<tr>
<td>1 0 0 1 1 1</td>
<td>- - - 4 5 0</td>
<td>1 0 0 1 1 1</td>
</tr>
</tbody>
</table>

Diagram:
- Graph with nodes 0 to 12 and edges indicating a cycle.
- Depth-first search (dfs) order: 0, 5, 4, 3.
- Marked, edgeTo, and onStack arrays showing the state of the graph.
- Red arrow indicates the cycle.
Topological sort

**DAG.** Directed acyclic graph.

Topological sort. Redraw DAG so all edges point left to right.

**Application.** Scheduling.

**Solution.** DFS (what else!).
Reverse DFS postorder in a digraph: Java implementation

```java
public class PostorderDFS {
    private boolean[] marked;
    private Stack<Integer> order;

    public PostorderDFS(Digraph G) {
        marked = new boolean[G.V()];
        order = new Stack<Integer>();
        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);
        order.push(v);
    }

    public Iterable<Integer> reverse() {
        return order;
    }
}
```
returns all vertices in "reverse DFS postorder"
Reverse DFS postorder in a DAG

marked[]  order

dfs(0): 1 0 0 0 0 0 0   -
dfs(1): 1 1 0 0 0 0 0   -
dfs(4): 1 1 0 0 1 0 0   -
4 done: 1 1 0 0 1 0 0   4
1 done: 1 1 0 0 1 0 0   4 1
dfs(2): 1 1 1 0 1 0 0   4 1
dfs(5): 1 1 1 0 1 1 0   4 1 2
2 done: 1 1 1 0 1 0 0   4 1 2
5 done: 1 1 1 0 1 1 0   4 1 2 5
0 done: 1 1 1 0 1 1 0   4 1 2 5 0
check 1: 1 1 1 0 1 1 0   4 1 2 5 0
check 2: 1 1 1 0 1 1 0   4 1 2 5 0
dfs(3): 1 1 1 1 1 0   4 1 2 5 0
check 2: 1 1 1 1 1 0   4 1 2 5 0
check 4: 1 1 1 1 1 0   4 1 2 5 0
check 5: 1 1 1 1 1 0   4 1 2 5 0
dfs(6): 1 1 1 1 1 1   4 1 2 5 0 6
6 done: 1 1 1 1 1 1   4 1 2 5 0 6
3 done: 1 1 1 1 1 1   4 1 2 5 0 6 3
check 4: 1 1 1 1 1 0   4 1 2 5 0 6 3
check 5: 1 1 1 1 1 0   4 1 2 5 0 6 3
check 6: 1 1 1 1 1 0   4 1 2 5 0 6 3

reverse DFS postorder → 3 6 0 5 2 1 4
Reverse DFS postorder in a DAG: an amazing fact

Reverse DFS postorder of a DAG is a topological order!

marked[] | order
----------|---------

dfs(0): 1 0 0 0 0 0 0 -
dfs(1): 1 1 0 0 0 0 0 -
dfs(4): 1 1 0 0 1 0 0 -
  4 done: 1 1 0 0 1 0 0 4
  1 done: 1 1 0 0 1 0 0 4 1
dfs(2): 1 1 1 0 1 0 0 4 1
dfs(5): 1 1 1 0 1 1 0 4 1 2
  check 2: 1 1 1 0 1 1 0 4 1 2
  5 done: 1 1 1 0 1 1 0 4 1 2 5
  0 done: 1 1 1 0 1 1 0 4 1 2 5 0
check 1: 1 1 1 0 1 1 0 4 1 2 5 0
ccheck 2: 1 1 1 0 1 1 0 4 1 2 5 0
dfs(3): 1 1 1 1 1 1 0 4 1 2 5 0
  check 2: 1 1 1 1 1 1 0 4 1 2 5 0
  check 4: 1 1 1 1 1 1 0 4 1 2 5 0
  check 5: 1 1 1 1 1 1 0 4 1 2 5 0
dfs(6): 1 1 1 1 1 1 1 4 1 2 5 0 6
  6 done: 1 1 1 1 1 1 1 4 1 2 5 0 6
  3 done: 1 1 1 1 1 1 1 4 1 2 5 0 6 3
check 4: 1 1 1 1 1 1 0 4 1 2 5 0 6 3
check 5: 1 1 1 1 1 1 0 4 1 2 5 0 6 3
check 6: 1 1 1 1 1 1 0 4 1 2 5 0 6 3

reverse DFS postorder: 3 6 0 5 2 1 4
Topological sort in a DAG: correctness proof

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, \( w \) was done before \( v \).

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

- **Case 3:** \( \text{dfs}(G, w) \) has already been called, but has not returned. Can’t happen in a DAG. \( v \rightarrow w \) makes a cycle.

Ex:
› digraph API
› digraph search
› transitive closure
› 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 \( v \).

(Equivalent) Vertices \( v \) and \( w \) are strongly connected if there is a directed cycle containing \( v \) and \( w \).

Def. A strong component is a maximal subset of strongly connected vertices.
Examples of strongly connected digraphs
Digraph-processing challenge 3

Problem. Are \( v \) and \( w \) strongly connected?

Goal. Linear preprocessing time, constant query time.

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert (or a COS 423 student).
- Intractable.
- No one knows.
- Impossible.

use DFS twice to find strong components (stay tuned)

4 strong components
Ecological food web graph

Vertex = species.
Edge: from producer to consumer.

Strong component. Subset of species with common energy flow.
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: CS226++. 
• Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju).
• Forgot notes for teaching algorithms class; developed alg in order to teach it!
• Later found in Russian scientific literature (1972).

1990s: more easy linear-time algorithms (Gabow, Mehlhorn).
• Gabow: fixed old OR algorithm.
• Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.
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. (!!)
Proposition: Kosaraju’s algorithm computes strong components

Lemma. Every vertex $s$ visited by $\text{dfs}(r)$ is strongly connected to $r$.

Proof of Lemma:

- Path from $r$ to $s$ exists ($\text{dfs()}$ follows edges).
  Thus, path from $s$ to $r$ exists in $G^R$.
- The only possibility for the $\text{dfs()}$ in $G^R$ implies path from $r$ to $s$ exists in $G^R$.
- Thus, $s$ and $r$ are strongly connected in $G^R$.
- Thus, $s$ and $r$ are strongly connected in $G$.

Proof of Proposition:

All vertices visited by $\text{dfs}(r)$ are strongly connected because they are all strongly connected to $r$. 
public class CCfinder
{
    private boolean[] marked;
    private Bag<Integer>[] connectedTo;
    private Bag<Integer> representatives;

    public CCfinder(Graph G)
    {
        marked = new boolean[G.V()];
        representatives = new Bag<Integer>();
        connectedTo = (Bag<Integer>[]) new Bag[G.V()];

        for (int s = 0; s < G.V(); s++)
            if (!marked[s])
            {
                Bag<Integer> bag = new Bag<Integer>();
                representatives.add(s);
                dfs(G, s, bag);
            }
    }

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

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

    public Iterable<Integer> representatives()
    {  return representatives;  }

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

Finding connected components in an undirected graph with DFS (from Section 4.1)
Finding strongly connected components in a digraph with DFS (Kosaraju)

```java
public class KosarajuSCC
{
    private boolean[] marked;
    private Bag<Integer>[] connectedTo;
    private Bag<Integer> representatives;

    public KosarajuSCC(Digraph G)
    {
        marked = new boolean[G.V()];
        representatives = new Bag<Integer>();
        connectedTo = (Bag<Integer>[]) new Bag[G.V()];
        PostorderDFS postorder = new PostorderDFS(G.reverse());
        for (int s : postorder.reverse())
            if (!marked[s])
                { Bag<Integer> bag = new Bag<Integer>();
                  representatives.add(s);
                  dfs(G, s, bag);
                }
    }

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

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

    public Iterable<Integer> representatives()
    { return representatives; }

    public Iterable<Integer> stronglyConnectedTo(int v)
    { return connectedTo[v]; }
}
```
## Digraph-processing summary: algorithms of the day

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-source reachability</td>
<td>Depth-first search (DFS)</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>Depth-first search (DFS) (from each vertex)</td>
</tr>
<tr>
<td>Topological sort (DAG)</td>
<td>Depth-first search (DFS)</td>
</tr>
<tr>
<td>Strong components</td>
<td>Kosaraju's algorithm (DFS twice)</td>
</tr>
</tbody>
</table>