4.2 DIRECTED GRAPHS

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
4.2 Directed Graphs

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
Road networks

Vertex = intersection; edge = one-way street.
Political blogosphere links

Vertex = political blog; edge = link.

The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005
Russian troll network

Vertex = Russian troll; edge = Twitter mention.
Science clickstreams

http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803
Uber rides

Vertex = taxi pickup; edge = taxi ride.

http://blog.uber.com/2012/01/09/uberdata-san-franciscomics
## 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>
Digraph. Set of vertices connected pairwise by **directed** edges.
# Some digraph problems

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s \rightarrow t$ path</td>
<td>Is there a path from $s$ to $t$?</td>
</tr>
<tr>
<td>shortest $s \rightarrow t$ path</td>
<td>What is the shortest path from $s$ to $t$?</td>
</tr>
<tr>
<td>directed cycle</td>
<td>Is there a directed cycle in the graph?</td>
</tr>
<tr>
<td>topological sort</td>
<td>Can vertices be ordered so all edges point from earlier to later vertex?</td>
</tr>
<tr>
<td>strong connectivity</td>
<td>Is there a directed path between every pairs of vertices?</td>
</tr>
<tr>
<td>transitive closure</td>
<td>For which vertices $v$ and $w$ is there a directed path from $v$ to $w$?</td>
</tr>
<tr>
<td>PageRank</td>
<td>What is the importance of a web page?</td>
</tr>
</tbody>
</table>
4.2 Directed Graphs

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort

https://algs4.cs.princeton.edu
## Digraph API

Almost identical to Graph API.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class Digraph</code></td>
<td></td>
</tr>
<tr>
<td><code>Digraph(int V)</code></td>
<td>create an empty digraph with V vertices</td>
</tr>
<tr>
<td><code>void addEdge(int v, int w)</code></td>
<td>add a directed edge v→w</td>
</tr>
<tr>
<td><code>Iterable&lt;Integer&gt; adj(int v)</code></td>
<td>vertices adjacent from v</td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>number of vertices</td>
</tr>
<tr>
<td><code>int E()</code></td>
<td>number of edges</td>
</tr>
<tr>
<td><code>Digraph reverse()</code></td>
<td>reverse of this digraph</td>
</tr>
</tbody>
</table>

Note: algs4 version has additional useful methods.
Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.
Which is the order of growth of the running time for removing an edge $v \rightarrow w$ from a digraph using the adjacency-lists representation, where $V$ is the number of vertices and $E$ is the number of edges?

A. 1

B. $\text{outdegree}(v)$

C. $\text{indegree}(w)$

D. $\text{outdegree}(v) + \text{indegree}(w)$
Which is the order of growth of the running time of the following code fragment if the digraph uses the adjacency-lists representation, where $V$ is the number of vertices and $E$ is the number of edges?

A. $V$
B. $E + V$
C. $V^2$
D. $VE$

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

prints each edge exactly once
**Digraph representations**

**In practice.** Use adjacency-lists representation.
- Algorithms based on iterating over vertices adjacent from \(v\).
- Real-world graphs tend to be **sparse** (not **dense**).

![Flowchart: Proportional to \(V\) edges vs. \(V^2\) edges]

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

\(^\dagger\) disallows parallel edges
Adjacency-lists graph representation (review): Java implementation

```java
public class Graph {

    private final int V;
    private 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 Bag<Integer>[] adj;

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

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

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

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
Reachability

**Problem.** Find all vertices reachable from $s$ along a directed path.
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a **digraph** algorithm.

---

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

Mark vertex v.

Recursively visit all unmarked vertices w adjacent from v.
Depth-first search demo

To visit a vertex $v$:

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

Diagram:

A directed graph with vertices labeled from 0 to 12. Arrows indicate the direction of edges between vertices.
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 v is connected to s
Depth-first search (in directed graphs)

Code for directed graphs identical to undirected one.

```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 connected to s
- constructor marks vertices connected to s
- recursive DFS does the work
- client can ask whether vertex v is reachable from s
Reachability application: program control-flow analysis

Every program is a digraph.
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

**Dead-code elimination.**
Find (and remove) unreachable code.

**Infinite-loop detection.**
Determine whether exit is unreachable.
Reachability application: mark–sweep garbage collector

Every data structure is a digraph.
- Vertex = object.
- Edge = reference.

**Roots.** Objects known to be directly accessible by program (e.g., stack).

**Reachable objects.** Objects indirectly accessible by program (starting at a root and following a chain of pointers).
Reachability application: mark–sweep garbage collector

Mark–sweep algorithm.
- 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.

Vol. 1, No. 2, June 1972

DEEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS®
ROBERT TARJAN†

Abstract. The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$, for some constants $k_1, k_2, k_3$, where $V$ is the number of vertices and $E$ is the number of edges of the graph being examined.
4.2 Directed Graphs

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort

https://algs4.cs.princeton.edu
Shortest directed paths

**Problem.** Find directed path from $s$ to each vertex that uses fewest edges.
Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

**BFS (from source vertex s)**

1. Put s on a queue, and mark s as visited.
2. Repeat until the queue is empty:
   - dequeue vertex v
   - enqueue all unmarked vertices adjacent from v, and mark them.

**Proposition.** BFS computes directed path with fewest edges from s to each vertex in time proportional to $E + V$. 
Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent from $v$ and mark them.
Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex \( v \) from queue.
- Add to queue all unmarked vertices adjacent from \( v \) and mark them.

all done
**MULTIPLE-SOURCE SHORTEST PATHS**

Given a digraph and a set of source vertices, find shortest path from any vertex in the set to every other vertex.

**Ex.** $S = \{1, 7, 10\}$.
- Shortest path to 4 is $7 \rightarrow 6 \rightarrow 4$.
- Shortest path to 5 is $7 \rightarrow 6 \rightarrow 0 \rightarrow 5$.
- Shortest path to 12 is $10 \rightarrow 12$.

![Graph Diagram](tinyDG.txt)

implemented in BreadthFirstDirectedPaths.java

**Q.** How to implement multi-source shortest paths algorithm?

**A.**
1. Add each element of $S$ to a queue, and mark each as visited
2. Run BFS as usual.
Suppose that you want to design a web crawler. Which graph search algorithm should you use?

A. depth-first search
B. breadth-first search
C. either A or B
D. neither A nor B

Exercise. how will you account for the fact that the list of vertices is not known in advance (and is potentially infinite)?
# Web crawler output

## BFS crawl

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

...  

## DFS crawl

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

...
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 marked websites.
- Dequeue the next website and enqueue any unmarked websites to which it links.

Note. Real-life web crawlers use more sophisticated algorithms.
4.2 Directed Graphs

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
Combinational circuit

Vertex = logical gate; edge = wire.
WordNet digraph

Vertex = synset; edge = hypernym relationship.

http://wordnet.princeton.edu
**Precedence scheduling**

**Goal.** Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

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

**DAG.** Directed acyclic graph.

Not a sort by usual definition — not a total order of vertices

**Topological “sort”**. Order the vertices of a DAG so that all edges point from an earlier vertex to a later vertex.

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

directed edges
```

```
DAG
```

```
topological order
```

- 0
- 1
- 2
- 3
- 4
- 5
- 6
Topological sort demo

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

Recall: In DFS postorder, visit vertex after recursive call.

A directed acyclic graph
Topological sort demo

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

postorder
4 1 2 5 0 6 3

topological order
3 6 0 5 2 1 4
done
public class DepthFirstOrder
{
    private boolean[] marked;
    private Stack<Integer> reversePostorder;

    public DepthFirstOrder(Digraph G)
    {
        reversePostorder = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

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

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

returns all vertices in “reverse DFS postorder”
Why does topological sort algorithm work?

- First vertex in postorder has outdegree 0.
- Second-to-last vertex in postorder can only point to last vertex.
- ...
**Proposition.** Reverse DFS postorder of a DAG is a topological order.

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

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

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

- **Case 3:** \( \text{dfs}(w) \) has already been called, but has not yet returned.
  - function-call stack contains path from \( w \) to \( v \)
  - edge \( v \rightarrow w \) would complete a directed cycle
  - contradiction (it’s a DAG)
Directed cycle detection

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

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.

![Diagram of a digraph with a directed cycle](image)

**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
default class A extends B { } ^
1 error
```
Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection.
## Digraph-processing summary: algorithms of the day

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-source reachability</td>
<td>in a digraph</td>
<td>DFS/BFS</td>
</tr>
<tr>
<td>Shortest path</td>
<td>in a digraph</td>
<td>BFS</td>
</tr>
<tr>
<td>Topological sort</td>
<td>in a DAG</td>
<td>DFS</td>
</tr>
</tbody>
</table>