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 (job x has to be done before job y), in which order should we schedule the tasks?

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

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

**tasks**

**precedence constraint graph:** need to take prerequisite before course

**feasible schedule**
**Topological sort**

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

**Topological sort.** Redraw DAG so all edges point upwards (or downwards or rightwards or...)
In a topological sort order v \(\rightarrow\)w implies v before w in the topological sort order.
The topological sort order is 3, 6, 0, 5, 2, 1, 4 here

<table>
<thead>
<tr>
<th>Directed edges</th>
<th>DAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0→5</td>
<td>0→2</td>
</tr>
<tr>
<td>0→1</td>
<td>3→6</td>
</tr>
<tr>
<td>3→5</td>
<td>3→4</td>
</tr>
<tr>
<td>5→2</td>
<td>6→4</td>
</tr>
<tr>
<td>6→0</td>
<td>3→2</td>
</tr>
<tr>
<td>1→4</td>
<td></td>
</tr>
</tbody>
</table>

**Solution.** DFS. What else?
Topological sort algorithm

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

Dfs(0): 1, 4, 4 done, back to 1, 1 done, back to 0, 2, 2 done, back to 0, 5, 5 done, back to 0, 0 done
Preorder: 0, 1, 4, 2, 5 (order visited)
Postorder: 4, 1, 2, 5, 0 (order done)

DFS(3): 3, 6, 6 done, back to 3, 3 done
Preorder: 3, 6
Postorder: 6, 3

Full preorder: 0, 1, 4, 2, 5, 3, 6
Full postorder: 4, 1, 2, 5, 0, 6, 3
Reverse postorder: 3, 6, 0, 5, 2, 1, 4 = top sort

A demo (.MOV file) is available for this at Princeton.edu

A directed acyclic graph
Topological sort algorithm

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

*Online demo* from USF (U. San Francisco)

Here are all the edges going in one direction.

**postorder**

4 1 2 5 0 6 3

**topological order**

3 6 0 5 2 1 4

done
A familiar example: prerequisites

Topological sort: cs110, cs210, cs240, cs310
Another one: cs110, cs240, cs210, cs310

Using dfs from cs110:
- cs210, cs310, done with 310, back to 210, done with 210, back to 110, 240, (see 310 marked), done with 240, back to 110, done with 110.
- Postorder: 310, 210, 240, 110
- Reverse postorder: 110, 240, 210, 310, a top sort

Using dfs from cs210 (to show you can start from anywhere)
- 310, done with 310, back to 210, done with 210
- Some vertices are still unmarked, so find one, say cs240
- 240, done with 240
- Some vertices are still unmarked, so find one, say cs110
- Done with 110 (both adjacent vertices are already marked)
- Postorder: 310, 210, 240, 110
- Reverse Postorder: 110, 240, 210, 310, a top sort
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;
    }
}
Prerequisites example execution trace

Constructor: Initialize empty stack, marked array
Do dfs from vertex 0, say cs110, unmarked
Mark it: marked[0] = true
adj list for it: vertex 1 and 2, unmarked
  Call dfs on vertex 1 (cs210)
    mark it: marked[1] = true;
    call dfs on its one adj vertex 3
      mark it: mark[3] = true
done with 3: push 3 on stack
done with 1: push 1 on stack
call dfs on vertex 2 (cs240), unmarked
mark it: marked[2] = true
find no unmarked adj vertices
done with 2: push 2 on stack
done with 0: push 0 on stack
Return to constructor: now stack = (0, 2, 1, 3)

In method reversePostorder(), return stack to caller as Iterable, i.e. not a stack, just a sequence of ints
Topological sort in a DAG: correctness proof

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

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

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

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

- **Case 3:** $dfs(w)$ 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.

(all vertices pointing from $3$ are done before $3$ is done, so they appear after $3$ in topological order)
Consider any edge v→w. When dfs(v) is called, we showed that dfs(w) has already been called, so w is marked already, or dfs(w) has not yet been called, but will be before dfs(v) finishes, so w will be marked before v is marked, which happens at the very end of dfs(v).

In either case w will be pushed on the stack before v, i.e., appears after v in the topological order, as required.
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: precedence scheduling

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

![Directed cycle detection application: precedence scheduling](http://xkcd.com/754)

**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 {
    ...
}
```

```
% 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!)
Observation. DFS visits each vertex exactly once. The order in which it does so can be important.

Orderings.

- Preorder: order in which dfs() is called.
- Postorder: order in which dfs() returns.
- Reverse postorder: reverse order in which dfs() returns.

```java
private void dfs(Graph G, int v) {
    marked[v] = true;
    preorder.enqueue(v);
    for (int w : G.adj(v))
        if (!marked[w]) dfs(G, w);
    postorder.enqueue(v);
    reversePostorder.push(v);
}
```
4.2 Directed Graphs

- Introduction
- Digraph API
- Digraph Search
- Topological Sort
- Strong Components
Strongly-connected components

**Def.** Vertices \( v \) and \( w \) are **strongly connected** if there is both a directed path from \( v \) to \( w \) and a directed path from \( w \) to \( v \).

**Key property.** Strong connectivity is an equivalence relation:
- \( v \) is strongly connected to \( v \).
- If \( v \) is strongly connected to \( w \), then \( w \) is strongly connected to \( v \).
- If \( v \) is strongly connected to \( w \) and \( w \) to \( x \), then \( v \) is strongly connected to \( x \).

**Def.** A **strong component** is a maximal subset of strongly-connected vertices.

![Diagram of a directed graph showing five strongly-connected components](image-url)
Connected components vs. strongly-connected components

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

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

connected component id (easy to compute with DFS)

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

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

constant-time client connectivity query

**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**

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

strongly-connected component id (how to compute?)

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

constant-time client strong-connectivity query
Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.

http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

Strong component. Subset of species with common energy flow.
Strong component application: software modules

Software module dependency graph.
- Vertex = software module.
- Edge: from module to dependency.

Firefox

Strong component. Subset of mutually interacting modules.

Approach 1. Package strong components together.

Approach 2. Use to improve design!

Note: I could only find one little cycle in the IE graph, on right-hand edge near the middle vertically, so only one strong component has multiple vertices here. Let me know if you can find another.
Our little software system: pa1

• The Xref object depends on the interface JavaTokenizer for type of its variable
• Xref’s main depends on WTokenizer for “new WTokenizer” and Xref’s object
• WTokenizer depends on JavaTokenizer because of “implements JavaTokenizer”
• RegexTokenizer depends on JavaTokenizer because of “implements JavaTokenizer”

• We see no cycles, no mutual dependencies, so each module is its own strong component.
• (main is not in an oval because it’s not an object type)
• That’s a good thing to see: use of the interface here does not cause cycles
Our little software system: pa3

- The Xref object depends on the interface JavaTokenizer for type of its variable
- Xref’s main depends on WTokenizer for “new WTokenizer” and Xref’s object
- WTokenizer depends on JavaTokenizer because of “implements JavaTokenizer”
- RegexTokenizer depends on JavaTokenizer because of “implements JavaTokenizer”

- We see no cycles, no mutual dependencies, so each module is its own strong component.
- That’s a good thing to see: use of the interface here does not cause cycles
Strong components algorithms: brief history

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

1972: linear-time DFS algorithm (Tarjan).
- Classic algorithm.
- Level of difficulty: Algs4++.
- Demonstrated broad applicability and importance of DFS.

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

1990s: more easy linear-time algorithms.
- Gabow: fixed old OR algorithm.
- Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.
Kosaraju-Sharir algorithm: intuition

Reverse graph. Strong components in $G$ are same as in $G^R$.

Kernel DAG. Contract each strong component into a single vertex.

Idea.
- Compute topological order (reverse postorder) in $G^R$.
- Run DFS on $G$, considering vertices in this topological order.

how to compute? DFS of course

first vertex is a sink (has no edges pointing from it)
Kosaraju-Sharir algorithm

Phase 1. Compute reverse postorder in $G^R$.
Phase 2. Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$.

Online demo from USF (does DFS, then DFS on $G^R$ in reordered vertex order)
Kosaraju-Sharir algorithm

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

**Phase 2.** Run DFS in $G$, visiting unmarked vertices in reverse postorder of $G^R$. Note that this graph has lots of cycles, so must have non-trivial strong components.

Because this graph has a cycle the reverse postorder will not be a topological sort, since it doesn’t have a top sort order!
Kosaraju-Sharir algorithm

**Phase 1. Compute reverse postorder in \( G^R \).**

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

(S&W page 589: using adj lists on page 569:
1 0 2 4 5 3 11 9 12 10 6 7 8)

DFS: doing adj list lowest id first, unlike S&W, page 589:
From 0: 2, 3, 4, 5, done with 5, back to 4, 6, 7, done with 7, back to 6, 8, done with 8, back to 6, done with 6, back to 4, 11, 9, 12, 10, done with 10, done with 12, back to 9, done with 9, back to 11, done with 11, back to 4, done with 4, back to 3, done with 3, back to 2, done with 2, back to 0, done with 0

From 1: done with 1
Postorder: 5, 7, 8, 6, 10, 12, 9, 11, 4, 3, 2, 0, 1
Reverse PO: 1, 0, 2, 3, 4, 11, 9, 12, 10, 6, 8, 7, 5

reverse digraph \( G^R \)
Kosaraju-Sharir algorithm algorithm

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

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

Here the vertices are relabeled for the second dfs:

Id 0: Dfs: from 0: done with 0
Id 1: From 1: 12, 4, 3, 2, done with 2, back to 3, done with 3, back to 4, done with 4, back to 12, done with 12, back to 1, done with 1
Id 2: From 5: 7, 6, 8, done with 8, back to 6, done with 6, back to 7, done with 7, back to 5, done with 5
Id 3: From 9: 10, done with 10, back to 9, done with 9
Id 4: From 11: done with 11

Translate results back to old indexes:
Id 0: new 0, old 1
Id 1: new 1, 12, 4, 3, 2
old: 0, 5, 4, 3, 2
Id 2: new 5, 7, 6, 8
old 11, 12, 9, 10
Id 3: new 9, 10
old 6, 8
Id 4: new 11
old 7

These groups are the strong components

Renumbered reverse digraph $G^R$
Kosaraju-Sharir algorithm algorithm

Phase 2. Translate back from new indexes to old original indexes

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

```
<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>
```

Result: five strong components: #0 has 1, #1 has 0,2,3,4,5, etc., by id[v] values
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.

Black arrows: explored in DFS to establish reverse postorder order

Red arrows: back against the reverse postorder order: wouldn’t exist if the graph was acyclic, are parts of cycles
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.

Mark of first vertex from each strong component: This vertex was last explored of its group in original DFS, at which point the DFS went on to the vertices of only-earlier components.
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!
Try it out on an acyclic graph (the one used earlier for top sort)

- Phase 1: run depth-first search (previously done).
- Return vertices in reverse postorder, in this case a proper topological order.

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

Reverse postorder
```
3 6 0 5 2 1 4
```

New indexes for Phase 2:
```
0 1 2 3 4 5 6
```
Every edge goes forward in new-index values.
Try it out on an acyclic graph (the one used earlier for top sort)

- Reverse directions, renumber, do DFS

**Reverse postorder of G**

3 6 0 5 2 1 4

New indexes:
0 1 2 3 4 5 6

DFS:
Id 0: from 0: done with 0
Id 1: from 1: done with 1
Id 2: from 2: done with 2
Id 3: from 3: done with 3
Id 4: from 4: done with 4
Id 5: from 5: done with 5
Id 6: form 6: done with 6

Pretty boring!
In an acyclic graph, every vertex is in its own strong component, so we have no need of the terminology.
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);
                count++;
            }
        }
    }

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

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

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

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

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

<table>
<thead>
<tr>
<th><strong>single-source reachability in a digraph</strong></th>
<th><img src="image1.png" alt="Diagram" /></th>
<th><strong>DFS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>topological sort in a DAG</strong></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><strong>DFS</strong></td>
</tr>
<tr>
<td><strong>strong components in a digraph</strong></td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><strong>Kosaraju-Sharir DFS (twice)</strong></td>
</tr>
</tbody>
</table>