4.2 Directed Graphs

**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.

**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

\[
\begin{align*}
0 &\rightarrow 1 \\
0 &\rightarrow 4 \\
0 &\rightarrow 6 \\
3 &\rightarrow 5 \\
5 &\rightarrow 4 \\
6 &\rightarrow 0 \\
1 &\rightarrow 4 \\
1 &\rightarrow 6
\end{align*}
\]

**Solution.** DFS. What else?

**Topological sort algorithm**

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

**Postorder**

\[4125063\]

**Topological order**

\[3605214\]

Here are all the edges going in one direction.

A familiar example: prerequisites.

**Topological sort**

Here are all the edges going in one direction.
Depth-first search order

```java
public class DepthFirstOrder {
  private boolean[] marked;
  private Stack<Integer> reversePostorder;

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

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

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

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

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

- **Case 3:** \( dfs(v) \) 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 \( v \) would complete a cycle.

Directed cycle detection

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

**Proof.**
- 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?

**Remark.** A directed cycle implies scheduling problem is infeasible.
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

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

Directed cycle detection application: spreadsheet recalculation

Depth-first search orders

**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**

**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:
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w is strongly connected to z, then v is strongly connected to z.

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

Connected components vs. strongly-connected components

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

**v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v.**

**Connected components.**

3 connected components

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

**Strongly-connected components.**

5 strongly-connected components

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

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 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 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: Alg4++.
- 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'.

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

Concept:
- Compute topological order (reverse postorder) in G.
- Run DFS on G, considering vertices in this topological order.

digraph G and its strong components kernel DAG of G (topological order: A B C D E)
Kosaraju-Sharir algorithm

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

[Diagram of a graph $G$ showing a reverse postorder]

Online demo from USF (does DFS, then DFS on $G'$ in reordered vertex order)

Kosaraju-Sharir algorithm

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

[Diagram of a graph $G'$ showing a reverse postorder]

Note that this graph has lots of cycles, so must have non-trivial strong components.

Kosaraju-Sharir algorithm

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

[Diagram of a graph $G'$ showing a reverse postorder and vertex relabeling]

Here the vertices are relabeled for the second dfs:

Kosaraju-Sharir algorithm

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

[Table showing index translation]

Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

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

Simple (but mysterious) algorithm for computing strong components.
- Phase 1: run DFS on \( G^\text{R} \) to compute reverse postorder.
- Phase 2: run DFS on \( G \), considering vertices in order given by first DFS.

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.

Reversed postorder
4 1 2 5 0 3 6 7
New indexes for Phase 2: 0 1 2 3 4 5 6
Every edge goes forward in new-index values.

Connected components in an undirected graph (with DFS)

```
public class CC
{
private boolean marked[];
private int count;
public CC(Graph G)
{
marked = new boolean[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;
for (int w : G.adj(v))
{
if (!marked[w])
dfs(G, w);
}
publish boolean connected(int v, int w)
{
return id[v] == id[w];
}
}
}
```

Strong components in a digraph (with two DFSs)

```
public class KosarajuSharirSCC
{
private boolean marked[];
private int count;
public KosarajuSharirSCC(Graph G)
{
marked = new boolean[G.V()];
for (int v = 0; v < G.V(); v++)
{
if (!marked[v])
dfs(G, v);
}
reversePostorder();
}
private void dfs(Graph G, int v)
{
marked[v] = true;
for (int w : G.adj(v))
{
if (!marked[w])
dfs(G, w);
}
public boolean connected(int v, int w)
{
return id[v] == id[w];
}
}
```

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 (2+p printing).
- Implementation: easy!
<table>
<thead>
<tr>
<th>Digraph-processing summary: algorithms of the day</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>single-source reachability in a digraph</strong></td>
</tr>
<tr>
<td><img src="image1.png" alt="Image" /></td>
</tr>
<tr>
<td><strong>topological sort in a DAG</strong></td>
</tr>
<tr>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td><strong>strong components in a digraph</strong></td>
</tr>
<tr>
<td><img src="image3.png" alt="Image" /></td>
</tr>
<tr>
<td><strong>DFS</strong></td>
</tr>
<tr>
<td><strong>DFS</strong></td>
</tr>
<tr>
<td><strong>Kosaraju-Sharir DFS (twice)</strong></td>
</tr>
</tbody>
</table>