Last Time

- Basic graph terminology
- Storing a graph in memory
- Depth-first search and paths
- Breadth-first search
Directed Graphs

• Instead of defining a graph as vertices and edges, now we use vertices and **directed edges**
  – Digraph
  – Outdegree
  – Indegree
  – Head
  – Tail

  – Directed path
  – Directed cycle
  – Directed acyclic graph (DAG)
  – Reachable
Finding Paths

• What changes with DFS?

```java
private void dfs(Digraph G, int v) {
    marked[v] = true;
    count++;
    for (int w : G.adj(v)) {
        if (!marked(w)) {
            dfs(G, w);
        } //if
    } //for
} //dfs
```
So Why Directed Graphs?

- Directly accessible objects
- Objects available for collection
- Potentially accessible objects

Objects may be collected in a directed manner depending on their accessibility.
Finding Paths

• What changes with BFS?

```java
private void bfs(Digraph G, int s) {
    Queue<Integer> queue = new Queue<Integer>();
    marked[s] = true;
    queue.enqueue(s);
    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        for (int w : G.adj(V)) {
            if (!marked(w)) {
                edgeTo[w] = v; // save last edge on path
                marked[w] = true; // mark it because path known
                queue.enqueue(w); // add it to the queue
            }
        }
    }
}
```
So What Can We Do?

- Finding directed cycles
- Topological sort
- Strong connectedness
- Transitive closure
Finding a Directed Cycle

private Stack<Integer> cycle;
private boolean[] onStack = new boolean[G.V()];

private void dfs(Digraph G, int v) {
    onStack[v] = true;
    marked[v] = true;
    for (int w: G.adj(v)) {
        if (this.hasCycle()) { return; }
        else if (!marked(w)) {
            edgeTo[w] = v; dfs(G, w);
        } else if (onStack[w]) {
            cycle = new Stack<Integer>;
            for (int x = v; x != w; x = edgeTo[x])
                cycle.push(x);
            cycle.push(w);
            cycle.push(b);
        }  //if-else
    }  //for
    onStack[v] = false;
}  //dfs

public boolean hasCycle() {
    return (cycle != null);
}  //hasCycle
Finding a Directed Cycle – Visual
Topological Order

• Can a graph have a topological order if it includes a cycle?
  – First, let’s extend DFS to store nodes in visited sequence.
  – Second, let’s examine which order is appropriate for finding a topological order.
Topological Order

```java
private Queue<Integer> pre = new Queue<Integer>();
private Queue<Integer> post = new Queue<Integer>();
private Stack<Integer> reversePost = new Stack<Integer>();

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

public Iterable<Integer> preorder() { return pre; }
public Iterable<Integer> postorder() { return post; }
public Iterable<Integer> reversePostorder() { return reversePost; }
```
Topological Order

Directed Graphs
Topological Order

• **Theorem:** Reverse postorder in a DAG is a topological sort.

• **Proof:**
  – Consider any edge $v \rightarrow w$. One of the following three cases must hold when $\text{dfs}(v)$ is called:
    • $\text{dfs}(w)$ has already been called and has returned ($w$ is marked).
    • $\text{dfs}(w)$ has not yet been called ($w$ is unmarked), so $v \rightarrow w$ will cause $\text{dfs}(w)$ to be called (and return) when $\text{dfs}(v)$ is called.
    • $\text{dfs}(w)$ has been called and has not yet returned when $\text{dfs}(v)$ is called. This case is impossible in a DAG, because the recursive call chain implies a path from $w$ to $v$ and $v \rightarrow w$ would complete a directed cycle.
  – In the two possible cases, $\text{dfs}(w)$ is done before $\text{dfs}(v)$, so $w$ appears before $v$ in postorder and after $v$ in reverse postorder. Thus, each edge $v \rightarrow w$ points from a vertex earlier in the order to a vertex later in the order, as desired.
Topological Order

Private Iterable<Integer> order;

Public Topological(Digraph G) {
    DirectedCycle cyclefinder = new DirectedCycle(G);
    if !(cyclefinder.hasMoreCycle()) {
        DepthFirstOrder dfs = new DepthFirstOrder(G);
        order = dfs.reversePostorder;
    } //if
} //Topological (constructor)
Topological Order – Analysis

• Theorem: With DFS, we can topologically sort a DAG in time proportional to $V + E$.

• Proof: Immediate from the code.
  – Depth-first search ensures that the graph has no cycles ($V + E$).
  – Depth-first search handles the reverse postorder ordering ($V + E$).■
Strong Connectedness

• Two vertices $v$ and $w$ are **strongly connected** if they are mutually reachable.
  – A directed path exists from $v$ to $w$.
  – A directed path exists from $w$ to $v$.

• A digraph is strongly connected if all its vertices are strongly connected to one another.
Strong Connectedness

- We can also use the idea of strong connectedness to partition a graph into strongly connected components (or strong components) – subsets of the graph that are internally strongly connected.
Kosaraju-Sharir Algorithm

• Start by building a condensation digraph – shrink all connected components into a single node for each.
  – The result must be a DAG, because a directed cycle would imply a larger strongly connected component.
Kosaraju-Sharir Algorithm

• Since the condensation digraph is a DAG, we can place its vertices in a topological order.

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

• We want to start at one of the nodes in the sink, and see which other nodes we can visit (and check for cycles).

• Because the graph that currently exists points from source to sink, we want to reverse all of the edges so that they point from sink to source.
Kosaraju-Sharir Algorithm

```java
public KosarajuSharirCC(Digraph G) {
    marked = new boolean[G.V()];
    id = new int[G.V()];
    DepthFirstOrder order = new DepthFirstOrder(G.reverse());
    for (int s : order.reversePostorder) {
        if (!marked(s)) {
            dfs(G, s);
            count++;
        } //if
    } //for
} //KosarajuSharirCC

private void dfs(Digraph G, int v) {
    ...
    id[v] = count;
    ...
} //dfs
```
Kosaraju-Sharir Algorithm – Visual
Transitive Closure

- The **transitive closure** of a digraph $G$ is another digraph with the same set of vertices, but with an edge from $v$ to $w$ in the transitive closure if and only if $w$ is reachable from $v$ in $G$. 
private DirectedDFS[] all;

TransitiveClosure(Digraph G) {
    all = new DirectedDFS[G.V()];
    for (int v = 0; v < G.V(); v++) {
        all[v] = new DirectedDFS(G, v);
    } //for
} //TransitiveClosure (constructor)

boolean reachable(int v, int w) {
    return all[v].marked(w);
} //reachable
Any Questions?