CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 20

April 22, 2014

Graph – Reminder

- Graph – a mathematical construction that describes objects and relations between them.
- A graph consists of a set of vertices and a set of edges that connect the vertices:
- G = (V, E) where V is the set of vertices (nodes) and E is the set of edges (arcs)
- Each edge is an ordered pair (v,w) or an ordered triplet (v, w,W) where $v, w \in V$ and W is the weight.
- A directed graph (digraph) is one whose edge pair is ordered.
- Vertex w is adjacent to vertex v if and only if $(v, w) \in E$

Definitions – Reminder

Definition (Path)

A sequence of vertices $w_1 \ldots w_n$ connected by edges s.t. $\{w_i, w_{i+1}\} \in E$ for each i=1..n.

Definition (Path Length)

Number of edges on the path.

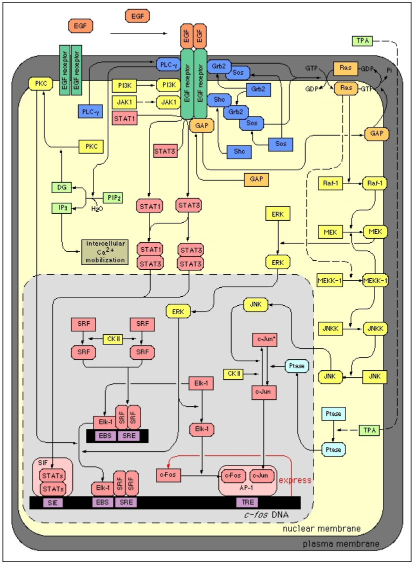
Definition (Weighted Path Length)

In a weighted graph, the sum of the costs of the edges on the path

Definition (Cycle)

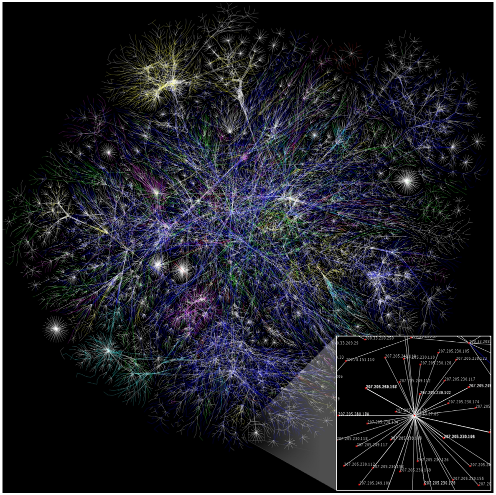
A path that begins and ends at the same vertex and contains at least one edge

Example – A Biological Network



This, Jen, Is the Internet

(A part of it, at least).



From Wikipedia

Airport Example

```
Graph<String,DefaultEdge> g = new
SimpleDirectedGraph<String,DefaultEdge>();
// provided concrete class
g.addVertex(‘‘BOS’’);
g.addVertex(‘‘LAX’’);
DefaultEdge e1 = g.addEdge(‘‘BOS’’,‘‘LAX’’);
DefaultEdge e2 = g.addEdge(‘‘LAX’’, ‘‘BOS’’);
// now e1 and e2 uniquely id these two edges
Map<DefaultEdge,Double> hours = new
HashMap<DefaultEdge,Double>();
hours.put(e1, 5.1);
hours.put(e2, 6.0);
// slower going west against winds
...
```

Airport Example

- Alternatively, create an Edge type with .equals and .hashCode based on some unique id for each edge. Then use g.addEdge("BOS","LAX", e) to put it into the graph. It can have hours as a field, so don't need hours HashMap. Looping through vertices in the graph:

```
 for (String airport:  g.vertexSet()) {
// do something with vertex named airport
}
```

- Looping through adjacent vertices to a vertex v: find hops out of LAX

```
 for (DefaultEdge e:g.edgesOf(``LAX'')) {
    if (g.getEdgeSource(e).equals(``LAX'')) {
        // filter edges
        double hrs = hours.get(e);
        // get hours for edge
        // do something with this hop
        }
}
```

Airport Example

- Alternatively, use DirectedGraph for the type of g, and that lets you use DirectedGraph's outgoingEdgesOf(V), which gives you the right set of edges.
- Hopefully you can see how to write a "hello, world" program using a Graph with this API: Set up a simple Vertex type, and a main with a new SimpleDirectedGraph, then a few addVertexs and addEdges, then a vertexSet() iteration to see that the nodes are really in the graph, and an edgeSet iteration to check the edges.
- Or look at LoadGraph.java.

Notes on Weiss Implementation (Optional)

- Weiss, pg. 536: vertexMap is Map from String to Vertex, pg. 535, which has List<Edge> for outgoing edges.
- Edge only knows dest Vertex. So different data structure but also using general idea of adjacency lists, and no generics of its own.
- Also, Vertex has "extra" fields dist, prev, and scratch for algorithm use.
- Weiss, pg. 532: We can use the same kind of input file. See test.dat.
- The Graph table has ids and adjacency lists. In addition, in the dark gray area, it holds intermediate results for a graph algorithm.
- JGraphT graphs don't hold "extra" info for algorithms.

Notes on Weiss Implementation (Optional)

- It is easy to have the same effect by setting up another Map from vertex/vertex name to (dist, prev) objects.
- JGraphT graphs are more general-purpose than Weiss's.
- What this is saying is that many shortest-path calculations are going to need "dist" and "prev" for each Node, and these will be changed as the algorithm calculates.

Directed Acyclic Graph (DAG)

- A cycle in a digraph is a path that returns to its starting vertex.
- A directed acyclic graph is also called a DAG.
- These graphs show up in lots of applications. For example, the graph of course prerequisites.
- It is a DAG, since a cycle in prerequisites would be ridiculous.

$$\begin{tikzcd}
\mathrm{CS110} \arrow[r] \arrow[dr] & \mathrm{CS210} \arrow[r] & \mathrm{CS310} \\
 & \mathrm{CS240} \arrow[ur] &
\end{tikzcd}$$

DAGs

- A DAG induces a partial order on the nodes.
- Not all element pairs have an order, but some do, and the ones that do must be consistent.
- So CS110 $<$ CS210 $<$ CS310, and so CS110 $<$ CS310, but CS210 and CS240 have no order between them.
- Suppose a student took only one course per term in CS. Then they would be finding a sequence that satisfies the partial order requirements, for example CS110, CS210, CS240, CS310.
- Another possible sequence is CS110, CS240, CS210, CS310.
- One of these fully ordered sequences that satisfy a partial order or DAG is called a topological sort of the DAG.
- A topological sort orders the nodes such that if there is a path between two nodes u and v, u will appear before v.

Finding a Topological Sort

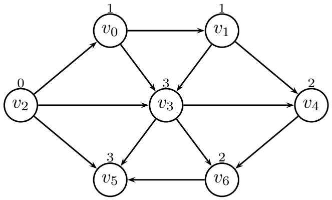
- Weiss presents a non-recursive algorithm for finding a topological sort of a DAG, checking that it really has no cycles.
- The first step of this algorithm is to determine the in-degree of all vertices in the graph.
- The in-degree of a vertex is the number of edges in the graph with this vertex as the to-vertex.
- Once we have all the in-degree numbers for the vertices, we look for a vertex with in-degree 0.
- It has no incoming edges, and so can be the vertex at the start of a topological sort, like CS110.

Finding a Topological Sort

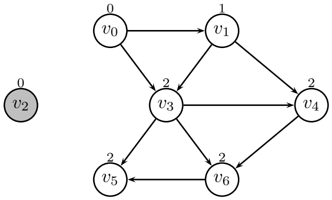
- Notice that there must be a node with in-degree 0 (why?).
- If there werent, then we could start a path anywhere, extend backwards along some in-edge from another vertex and from there to another, etc.
- Eventually we would have to start repeating vertices.
- For example, if we have managed to avoid repeating vertices and have visited all the vertices, then the last vertex still has an in-edge not yet used, and it goes to another vertex, completing a cycle.
- Thus the lack of an in-degree-0 vertex is a sure sign of a cycle and a DAG doesn't have any cycles.
- OK, we have the very first vertex, but what about the rest? Think recursively!

Topological Sort Example

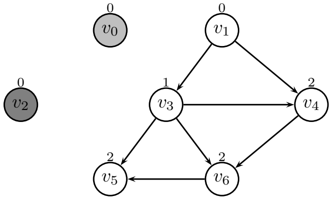
1



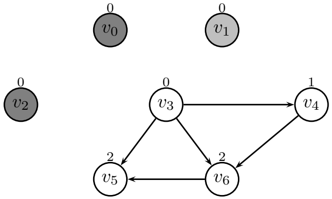
2



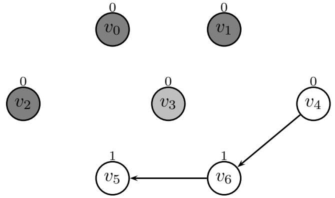
3

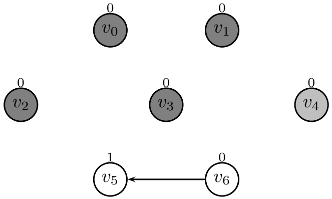


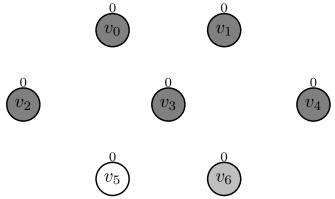
4

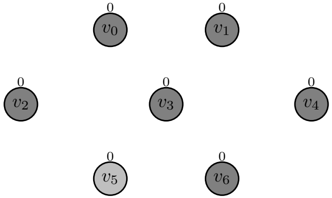


Topological Sort Example

5


6


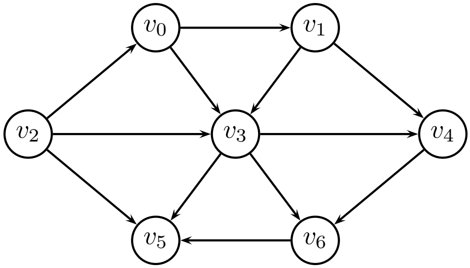
7


8


The topological order is:
$V_2, V_0, V_1, V_3, V_4, V_6, V_5$

Topological Sorting Using an Array of In-degree Values

V_0	V_1	V_2	V_3	V_4	V_5	V_6	out
1	1	0	3	2	3	2	V2
0	1	0	2	2	2	2	V0
0	0	0	1	2	2	2	V1
0	0	0	0	1	2	2	V3
0	0	0	0	0	1	1	V4
0	0	0	0	0	1	0	V6
0	0	0	0	0	0	0	V5


$$\begin{tikzcd}
 & v_0 \arrow[r] \arrow[dr] & v_1 \arrow[dl] \arrow[dr] & \\
v_2 \arrow[ur] \arrow[rr] \arrow[dr] & & v_3 \arrow[r] \arrow[dl] \arrow[d] & v_4 \arrow[dl] \\
 & v_5 & v_6 \arrow[l] &
\end{tikzcd}$$

Calculating in-Degrees For All Vertices

- The first step of topological sort algorithm, as a Graph application example.
- Each vertex has V-value to identify it.
- For each V value, we have a count of in-edges, an integer. A Map of V to Integer will do the trick.
- With JGraphT, all we really have to do is use the type DirectedGraph, and bingo, it has an inDegreeOf(V) method to give these numbers.

Pseudocode for Topological Sort

```
Create a queue q and enqueue all vertices of
in-degree 0.
Create an empty list t for top-sorted vertices.
Loop while q is not empty
    Dequeue from q a vertex v and append it to list t
    Loop over vertices w adjacent to v
    Decrement w's indegree
    (i.e.  decrement value in Map m)
    If w's indegree is 0, enqueue w in q
Return top-sorted list t.
```

What Happens If There is a Cycle?

- The presence of a loop means in-degree contributions of one for all members of the cycle, so the cycle protects the whole group from being put on the queue.
- For example, consider $A \rightarrow B \rightarrow C \rightarrow A$ and see in-degree $=$ 1 for all nodes.
- Each pass of the loop dequeues an element from the queue, so all that can happen is that the queue goes empty with the cycle members still un-enqueued.
- Use a counting trick to add cycle-detection to this algorithm

Pseudocode for Topological Sort with Cycle Detection

```
 Calculate in-degrees for nodes (using Map from node-key
to int in-degree).
Create a queue q and enqueue all nodes of in-degree 0.
Create an empty list t for top-sorted nodes
Set loopcount to 0     <---added
Loop while queue is not empty
    Increment loopcount     <---added
    Dequeue from q a node v and append it to list t
    Loop over nodes w adjacent to v
        Decrement w's indegree (in Map)
        If w's indegree is 0, enqueue w in q
If loopcount < number of nodes in graph,
    return cycle-in-graph     <---added
Return top-sorted list t
```

Topological Sort Algorithm

- This algorithm is $O(|E|)$, which is as good as it can be, so this is a good cycle-detection algorithm.
- Just drop the list of top-sorted vertices if you don't want the top-sorted result.
- Weiss has related code on pg 559, method "acyclic" of his Graph class, using direct access to private data of his Graph class.
- Note that our algorithm is a client of our (more powerful) Graph class.
- We don't need access to private data of Graph to loop through vertices or loop through vertices adjacent to a given vertex.

Topological Sort Algorithm

- Weiss' code doesn't need the Map to hold in-degrees, because he has put an extra field in his vertex class called "scratch" for extra numbers for various algorithms.
- However, this is not clean programming.
- Our use of Map is fast and appropriate.
- It allows us to associate more information with a vertex whenever we need to.

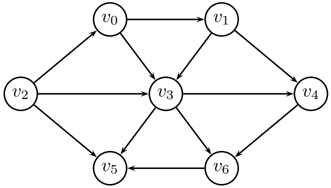
Graph Traversal

- Recall tree traversals from CS210.
- For a quick review, look at Fig. 18.23, pg. 668.
- In the preorder traversal, we start at the root, and plunge down the leftmost side, then back and down the next child of root, down, down, back up, back up, next child, down, down, back up 3x, no more children of root, done.
- You should also know the postorder and inorder, but the preorder is the one that generalizes to "depth-first search" (DFS) of a directed graph.

Depth-First Search (DFS)

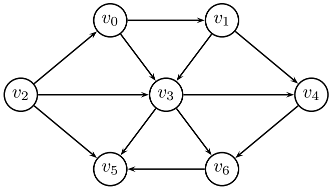
- DFS starting from V_2, etc. where we use the lower numbered adjacent vertex first. (We need some way of deciding which edge to visit first.
- In implementations, we simply follow the order of elements on the adjacency list for the vertex.)
- See how we can follow out-edges down and down:

$V_2 \rightarrow V_0 \rightarrow V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6 \rightarrow V_5$

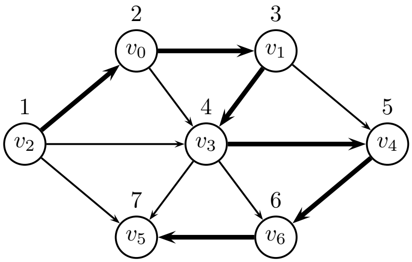


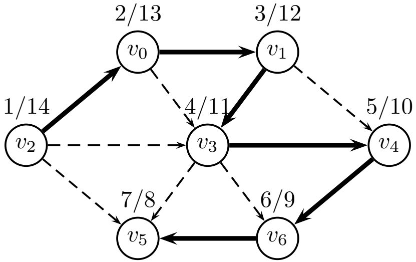
Depth-First Search (DFS)

- We're stuck with no place to go at V_5.
- We can backtrack back to V_6, back to V_4, back to V_3, see another adjacent vertex, V_6, but already visited, back to V_1, see another adj vertex, V_4, but already visited, back to V_0, see another adj vertex, V_3, but already visited, back to V_2, but V_0 and V_5 were already visited. Done.
- Notice that we don't necessarily have to start from V_2.



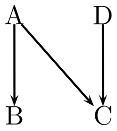
Depth-First Search (DFS)





Another DFS Example

For a simpler graph, consider the following:



- All the edges point downward.
- The DFS starting from A visits A, then B, then backtracks and visits C, backtracks back to A and is done for one tree.
- It starts over at D, but finds no unvisited vertices, so D is alone as the second tree.
- So the DFS visitation order is A, B, C, D.
- Notify vertices in DFS visitation order

Pseudocode for DFS

```
 Set up Set of ''unvisited'' vertices, initially
containing all the vertices.
for each vertex v
    // notify DFS tree starting
    if (v is in unvisited)
        dfs(v)


 dfs(vertex v)
// notify DFS visitation of v
remove v from unvisited set
for each vertex n adjacent to v
    if (n is in unvisited)
        dfs(n)
```

DFS Implementation

- Java: see DFS.java handout. This is generic <V,E> code, and uses Observable-to-Observer notifications.
- We can use DFS to do topological sort, by changing when we do the vertex notification.
- For a topological sort of this graph, we need A and D before C, so the DFS order isn't good with D at the end.
- It turns out we need to notify the points of last visit to each node: B, C, A, D
- Then the reverse of this is a topological sort.

DFS – Some Comments

- We can do a DFS of any directed (or undirected) graph, and if it's acyclic, the DFS yields a topological sort.
- If there is a cycle in the graph, it doesn't cause an infinite loop because a DFS doesn't revisit a vertex.
- In general a DFS works in phases, finding trees, so the whole thing finds a forest.

DFS on a DAG

- The first tree is a tree being traversed in preorder.
- Once this traversal returns to a vertex, it has classified all vertices it has visited as “downwind” from it, i.e. belonging to the right of it in topological sort. (And no edge can be inbound from those, because of the acyclic requirement.)
- The extreme is V2, the last revisited after going over all the edges, and it should be first in the top sort order for that tree.
- The second-to-last vertex re-visited should be second, and so on.

DFS – Some Comments

- The following trees in the forest have only inter-group edges back to previous groups, so they need to be ordered last-first in top-sort order.
- The ability of the DFS to turn a graph into a forest of trees is useful in many algorithms.
- Trees are a lot easier to work with than general graphs.
- Note that a graph does not have to be acyclic to do a DFS. It's just that you can only get a topological sort out of it if it's acyclic.

DFS – Some Comments

- The graph sources have cs310.DFS.java to do DFS traversals, and DFSTest.java as a sample application (no Swing).
- DFSDemo creates random graphs and shows them using Swing.
- The handout shows the code needed to set up for a simple screen display of the final graph as specified in pa6.
- You can use DFS.java to do the topological sort problem on pa6 if you want, but hide a modified DFS class as a private inner class.