Today's Outline

• Admin:
  – Homework #4 - due Thurs, Nov 8th at 11pm
  – Midterm 2, Fri Nov 16

• Graphs
  – Representations
  – Topological Sort
  – Graph Traversals

Topological Sort

Problem: Given a DAG $G = (V, E)$, output all the vertices in order such that if no vertex appears before any other vertex that has an edge to it.

Example input:

Example output:

Questions and comments

• Why do we perform topological sorts only on DAGs?

• Is there always a unique answer?

• What DAGs have exactly 1 answer?

• Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it.
Questions and comments

• Why do we perform topological sorts only on DAGs?
  – Because a cycle means there is no correct answer

• Is there always a unique answer?
  – No, there can be 1 or more answers; depends on the graph

• What DAGs have exactly 1 answer?
  – Lists

• Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it

Uses

• Figuring out how to graduate
• Computing the order in which to recompute cells in a spreadsheet
• Determining the order to compile files using a Makefile
• In general, taking a dependency graph and coming up with an order of execution

A first algorithm for topological sort

1. Label each vertex with its in-degree
   – Labeling also called marking
   – Think “write in a field in the vertex”, though you could also do this with a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled in-degree of 0
   b) Output \( v \) and “remove it” (conceptually) from the graph
   c) For each vertex \( u \) adjacent to \( v \) (i.e. \( (v,u) \) in \( E \)), decrement the in-degree of \( u \)

Example

Output:

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed?: 1 1 1 1 1 1 1 1 1 1 1 1
In-degree: 0 0 2 1 2 1 1 1 1 1 1 1

Example

Output: 126

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed?: x x
In-degree: 0 0 2 1 2 1 1 1 1 1 1 1

Example

Output: 126

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed?: x x
In-degree: 0 0 2 1 2 1 1 1 1 1 1 1
A couple of things to note

- Needed a vertex with in-degree of 0 to start
  - No cycles
- Ties between vertices with in-degrees of 0 can be broken arbitrarily
  - Potentially many different correct orders

Topological Sort: Running time?

- What is the worst-case running time?
  - Initialization $O(|V| + |E|)$
  - Sum of all find-new vertex $O(|V|^2)$ (because each $O(|V|)$)
  - Sum of all decrements $O(|E|)$ (assuming adjacency list)
  - So total is $O(|V|^2 + |E|)$ – not good for a sparse graph!

Doing better

The trick is to avoid searching for a zero-degree node every time!
- Keep the “pending” zero-degree nodes in a list, stack, queue, box, table, or something
- Order we process them affects output but not correctness or efficiency provided add/remove are both $O(1)$

Using a queue:
1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) $v =$ dequeue()
   b) Output $v$ and remove it from the graph
   c) For each vertex $u$ adjacent to $v$ (i.e. $u$ such that $(v, u)$ in $E$), decrement the in-degree of $u$, if new degree is 0, enqueue it
Optimized Topological Sort:

```java
labelAllAndEnqueueZeros();
for(ctr=0; ctr < numVertices; ctr++){
    v = dequeue();
    put v next in output
    for each w adjacent to v {
        w.indegree--;
        if(w.indegree==0) enqueue(w);
    }
}
```

- What is the worst-case running time?
  - Initialization: \( O(|V| + |E|) \)
  - Sum of all enqueues and dequeues: \( O(|V|) \)
  - Sum of all decrements: \( O(|E|) \) (assuming adjacency list)
  - So total is \( O(|E| + |V|) \) – much better for sparse graph!

Graph Traversals

Next problem: For an arbitrary graph and a starting node \( v \), find all nodes reachable (i.e., there exists a path) from \( v \)
- Possibly "do something" for each node (an iterator!)
  - E.g. Print to output, set some field, etc.

Related:
- Is an undirected graph connected?
- Is a directed graph weakly / strongly connected?
  - For strongly, need a cycle back to starting node

Basic idea:
- Keep following nodes
- But "mark" nodes after visiting them, so the traversal terminates and processes each reachable node exactly once

Running time and options

- Assuming add and remove are \( O(1) \), entire traversal is \( O(|E|) \)
- The order we traverse depends entirely on add and remove
  - Popular choice: a stack “depth-first graph search” (DFS)
  - Popular choice: a queue “breadth-first graph search” (BFS)
- DFS and BFS are “big ideas” in computer science
  - Depth: recursively explore one part before going back to the other parts not yet explored
  - Breadth: Explore areas closer to the start node first

Recursive DFS, Example: trees

- A tree is a graph and DFS and BFS are particularly easy to "see"

```java
DFS(Node start) {
    mark and "process" (e.g., print) start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```

- Order processed: A, B, D, E, C, F, G, H
- Exactly what we called a “pre-order traversal” for trees
  - The marking is because we support arbitrary graphs and we want to process each node exactly once
Depth First Search (DFS) with a stack:

```java
DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and "process"
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}
```

- Order processed: A, C, F, H, B, E, D
- A different but perfectly fine traversal

Breadth First Search (BFS) with a queue:

```java
BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue // and "process"
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}
```

- Order processed: A, B, C, D, E, F, G, H
- A "level-order" traversal

What if I want to find the “shortest” path?

- Breadth-first always finds shortest paths in terms of minimum number of edges from the starting node.

An aside: Depth-first can use less space in finding a path
- If longest path in the graph is $p$ and highest out-degree is $d$ then DFS stack never has more than $dp$ elements
- But a queue for BFS may hold $O(|V|)$ nodes

Saving the path

- Our graph traversals can answer the "reachability question":
  - "Is there a path from node $x$ to node $y$?"
- Q: But what if we want to output the actual path?
  - Like getting driving directions rather than just knowing it’s possible to get there!
- A: Like this:
  - Instead of just “marking” a node, store the previous node along the path (when processing $v$ causes us to add $v$ to the search, set $v$.path field to be $u$)
  - When you reach the goal, follow $v$.path fields backwards to where you started (and then reverse the answer)
  - If just wanted path length, could put the integer distance at each node instead
Example using BFS

What is a path from Seattle to Tyler
- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique

Example using BFS

What is a path from Seattle to Tyler
- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique