CSE 332: Data Structures & Parallelism
Lecture 20: Topological Sort / Graph Traversals

Ruth Anderson
Autumn 2018
Today

- Graphs
  - 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:

142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352

Disclaimer: Do not use for official advising purposes! (Implies that CSE 332 is a pre-req for CSE 312 – not true)
Valid Topological Sorts:
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
Topological Sort Uses

- Figuring out how to finish your degree
- 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 (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

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

<table>
<thead>
<tr>
<th>In-degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>/</td>
</tr>
</tbody>
</table>

11/26/2018
Example

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

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

Output: 126
        142

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 2 1 1 1 1
          1
          0

11/26/2018
Example

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

Output: 126
        142
        143
Example

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

Output: 126
         142
         143
         311
...
Example

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

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

[Graph showing relationships between nodes]

Output: 126
142
143
311
331
332

11/26/2018
Example

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

Output: 126
142
143
311
331
332
312
Example

Node:  126  142  143  311  312  331  332  333  341  351  352  440

Removed?  x  x  x  x  x  x  x  x  x  x  x  x

In-degree: 0  0  2  1  2  1  1  2  1  1  1  1
0  1  0  0  0  1  0  0  0  0  0  0
0  0

Output:  126  142  143  311  331  332  312  341  351  352
Example

Output: 126 142 143 311 312 331 332 333 341 351 352 440

Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed?  x  x  x  x  x  x  x  x  x  x  x  x

In-degree: 0 0 2 1 2 1 1 2 1 1 1 1

11/26/2018
Example

Output: 126 142 143 311 331 332 312 341 351 333 352 440

Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed? x x x x x x x x x x x x x

In-degree: 0 0 2 1 2 1 1 2 1 1 1 1

1 0 1 0 0 1 0 0 0 0 0 0

0 0 0
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?

```c
labelEachVertexWithItsInDegree();
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```
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) $v = \text{dequeue}()$
   b) Output $v$ and remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v, w)$ in $E$), decrement the in-degree of $w$, if new degree is 0, enqueue it
Topological Sort (optimized): Running time?

```c
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);
    }
}
```
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 Questions:
- 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
Graph Traversal: Abstract Idea

```java
traverseGraph(Node start) {
    Set pending = emptySet();
pending.add(start)
mark start as visited
while(pending is not empty) {
    next = pending.remove()
    for each node u adjacent to next
        if(u is not marked) {
            mark u
            pending.add(u)
        }
}
}
```
Running time and options

• Assuming add and remove are $O(1)$, entire traversal is $O(|E|)$
  • Use an adjacency list representation

• The order we traverse depends entirely on how add and remove work/are implemented
  – Depth-first graph search (DFS): a stack
  – Breadth-first graph search (BFS): a queue

• 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”

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 not needed here, but we need it to support arbitrary graphs, we need a way to process each node exactly once
DFS with a stack, Example: trees

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 different but perfectly fine traversal
**BFS with a queue, Example: trees**

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 “level-order” traversal
**DFS/BFS Comparison**

Breadth-first search:
- Always finds shortest paths, i.e., “optimal solutions
  - Better for “what is the shortest path from x to y”
- Queue may hold $O(|V|)$ nodes (e.g. at the bottom level of binary tree of height $h$, $2^h$ nodes in queue)

Depth-first search:
- 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 $d*p$ elements

A third approach: *Iterative deepening (IDDFS)*:
- Try DFS but don’t allow recursion more than $k$ levels deep.
  - If that fails, increment $k$ and start the entire search over
- Like BFS, finds shortest paths. Like DFS, less space.
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 u causes us to add v to the search, set v.path field to be u)
  – When you reach the goal, follow 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 Austin
  – Remember marked nodes are not re-enqueued
  – Note shortest paths may not be unique
Example using BFS

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