CSE 332: Data Structures & Parallelism
Lecture 19: Topological Sort, Traversals

Arthur Liu

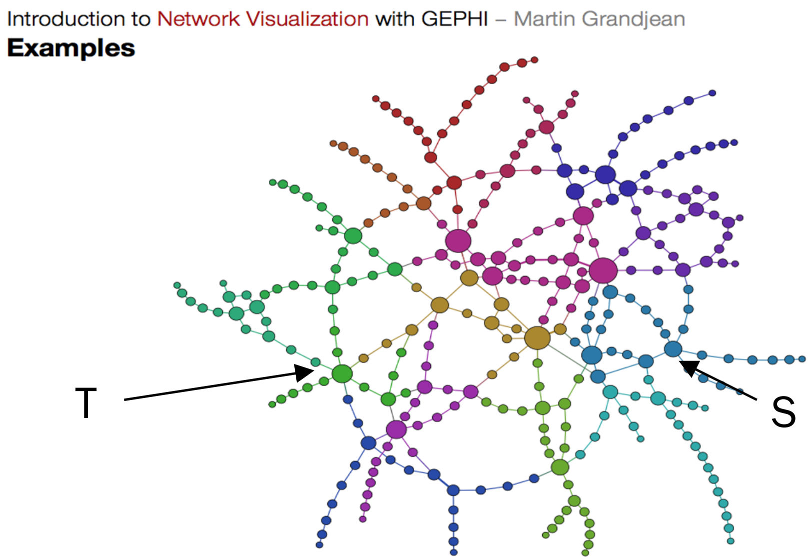
Summer 2022

Outline for Today

- Topological Sort
- BFS, DFS

Graph Problems

- Lots of interesting questions we can ask about a graph:
  - What is the shortest route from S to T? What is the longest route without cycles?
  - Are there cycles in this graph?
  - Is there a cycle that uses each *vertex* exactly once?
  - Is there a cycle that uses each *edge* exactly once?



Graph Problems More Theoretically

- Some well known graph problems and their common names:
  - **s-t Path.** Is there a path between vertices s and t?
  - **Connectivity.** Is the graph connected?
  - **Biconnectivity.** Is there a vertex whose removal disconnects the graph?
  - **Shortest s-t Path.** What is the shortest path between vertices s and t?
  - **Cycle Detection.** Does the graph contain any cycles?

    O(E)
  - **Euler Tour.** Is there a cycle that uses every edge exactly once?
  - **Hamilton Tour.** Is there a cycle that uses every vertex exactly once?

    NP-Hard
  - **Planarity.** Can you draw the graph on paper with no crossing edges?
  - **Isomorphism.** Are two graphs the same graph (in disguise)?

- Often can't tell how difficult a graph problem is without very deep consideration.

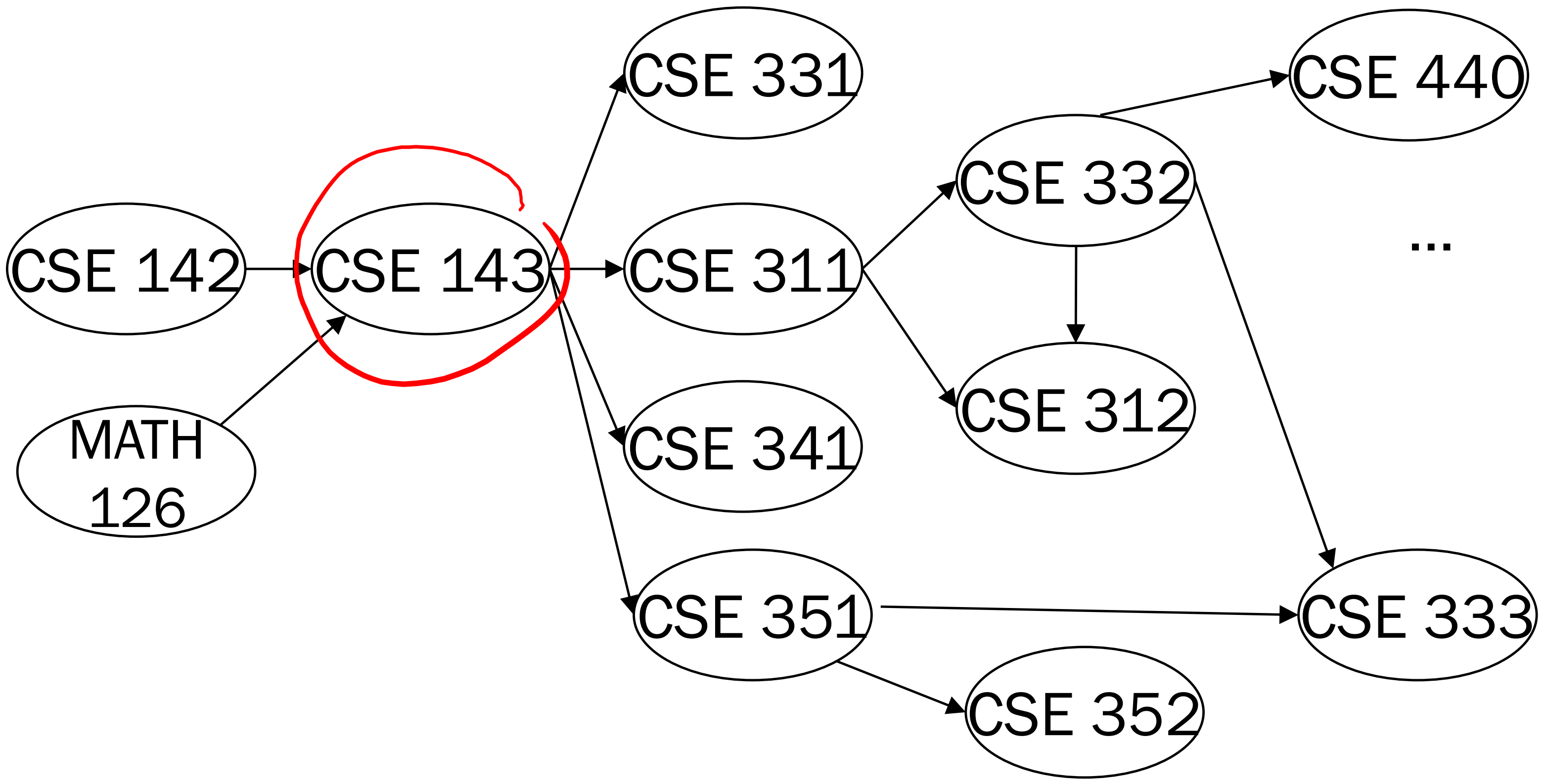
First graph algorithm!

Topological Sort

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

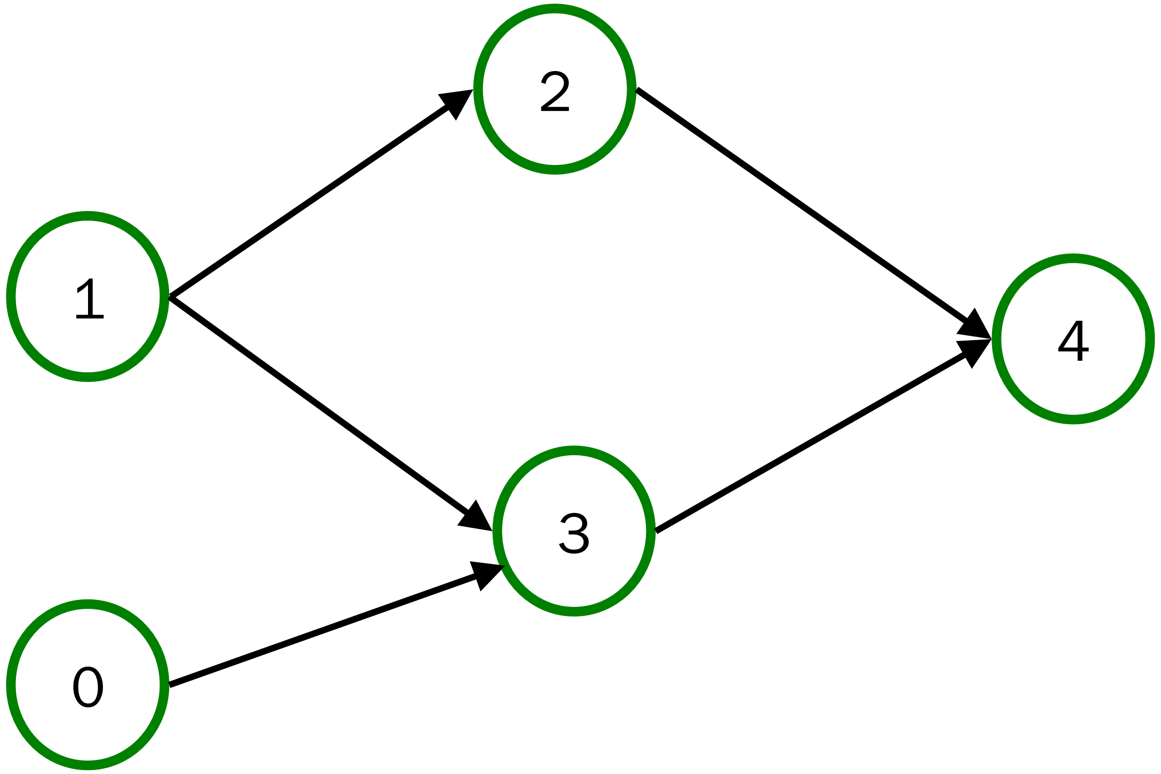
Problem: Given a DAG **G=(V,E)**, output all the vertices in order such that no vertex appears before any other vertex that has an edge to it

Example input:



Example output:

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



~~1, 2, 4~~

~~0, 3~~

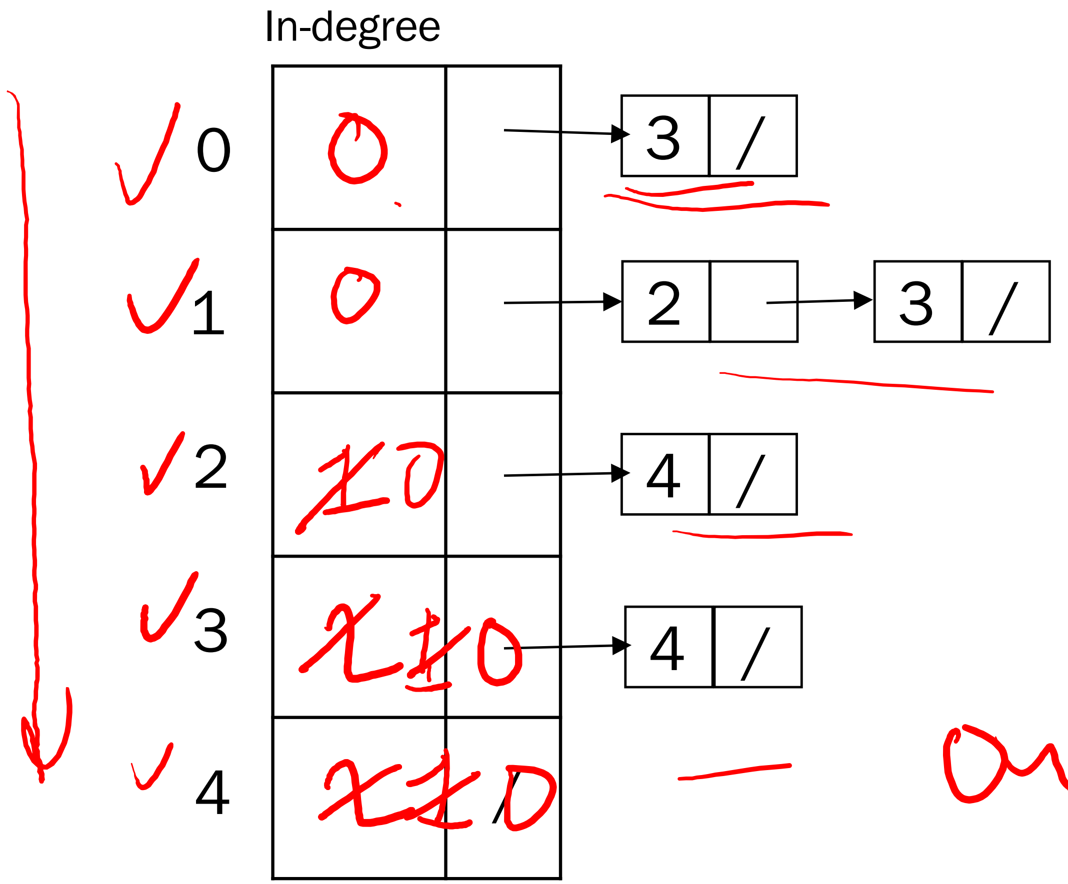
Valid Topological Sorts:

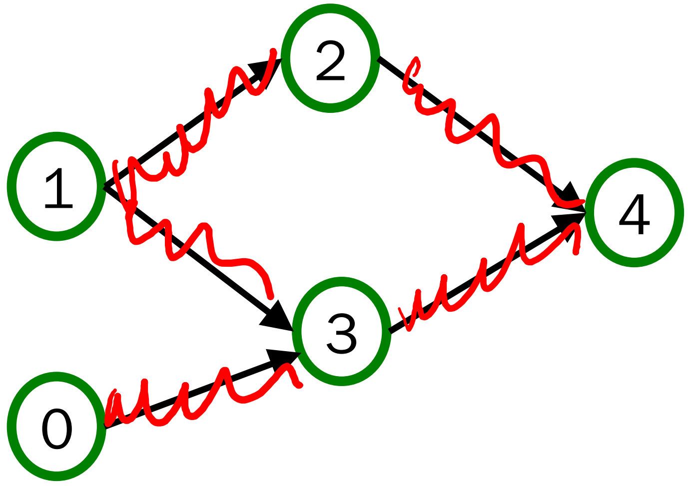
0, 1, 2, 3, 4
1, 0, 3, 2, 4
1, 0, 2, 3, 4
0, 1, 2, 3, 4
1, 2, 0, 3, 4

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** 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**

O(V+E)

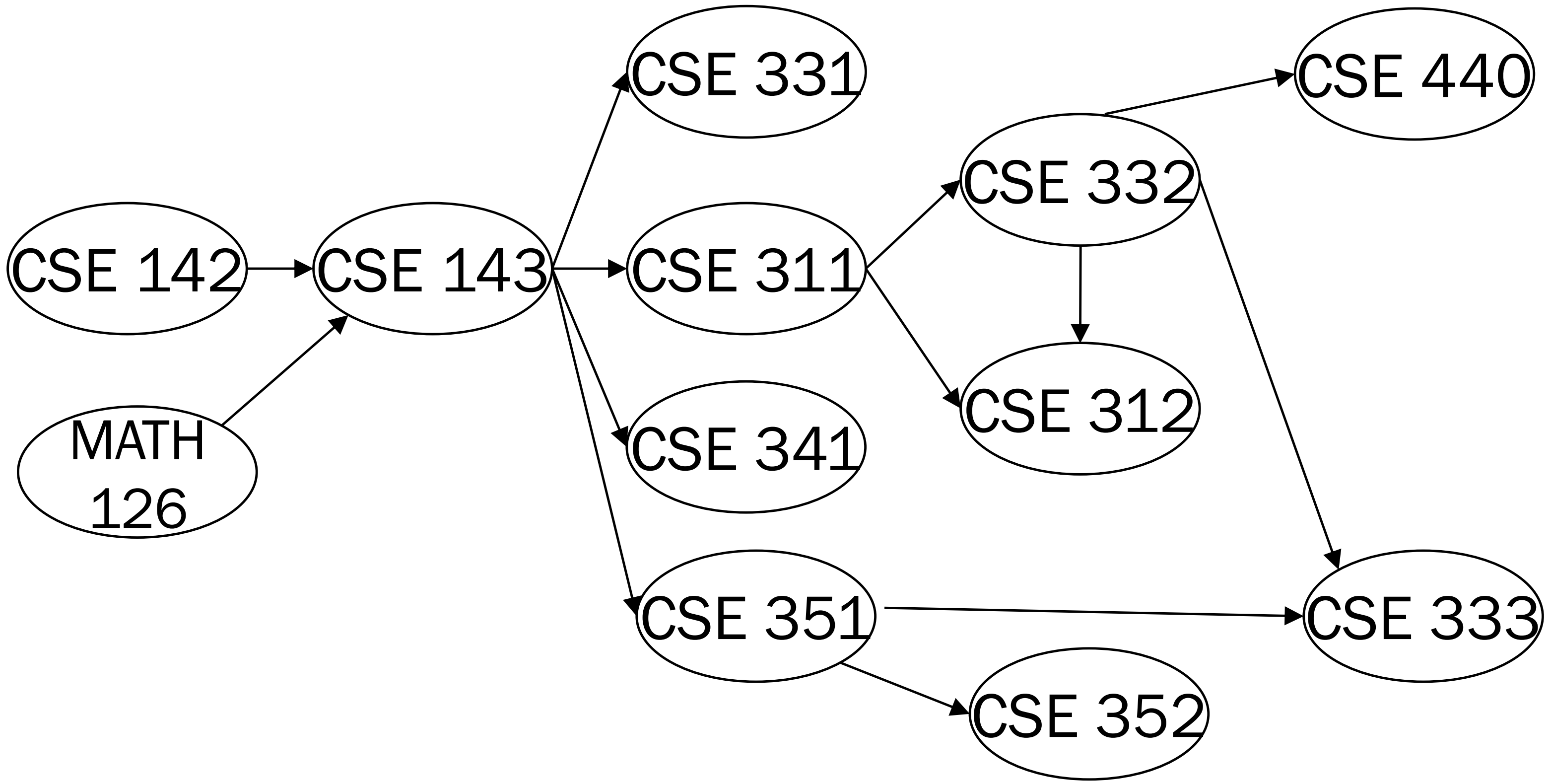




Output: 1, 0, 2, 3, 4

Example

Output:

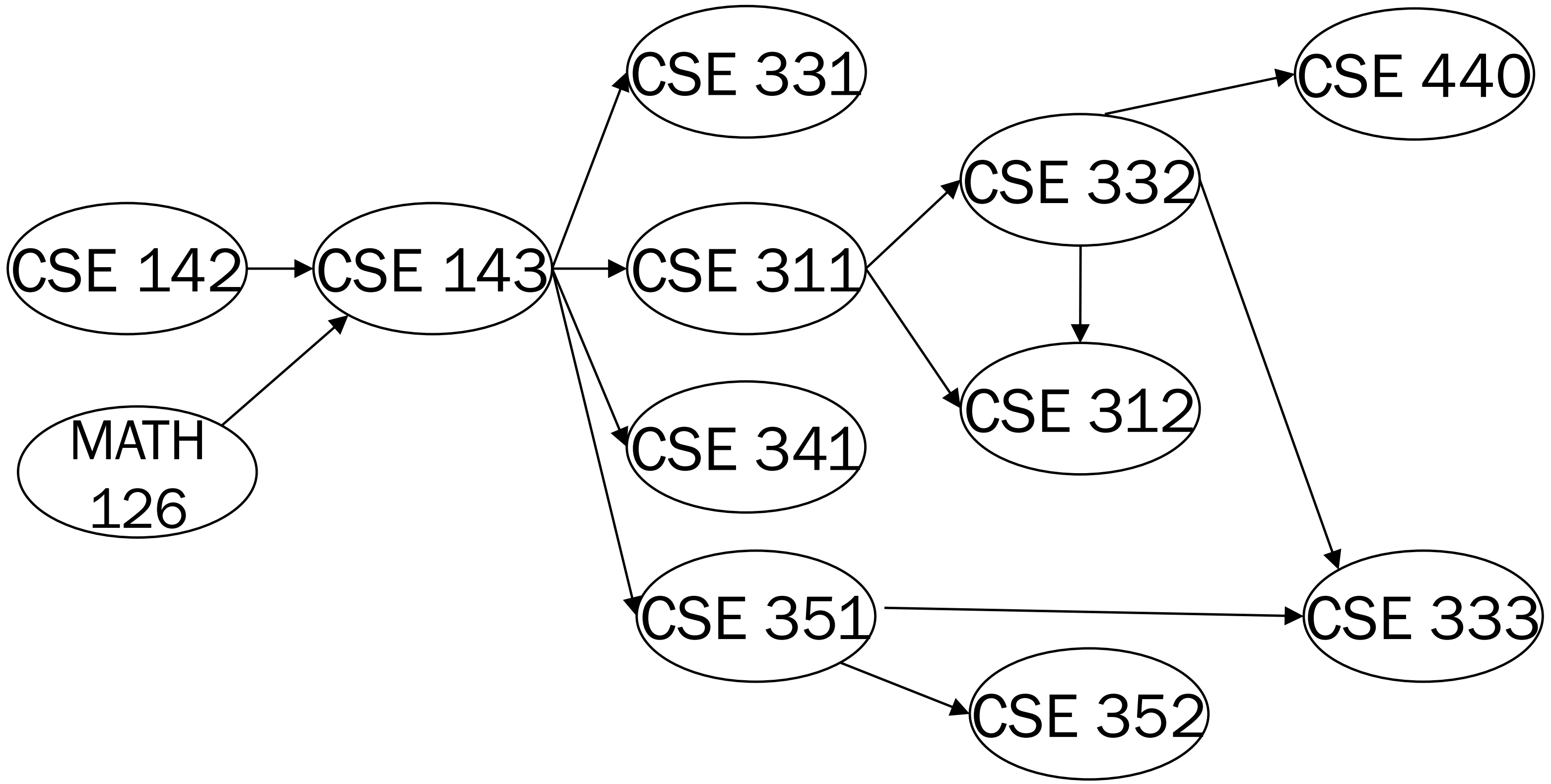


Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?												
In-degree	0	0	2	1	2	1	1	2	1	1	1	1

Example

Output:

126



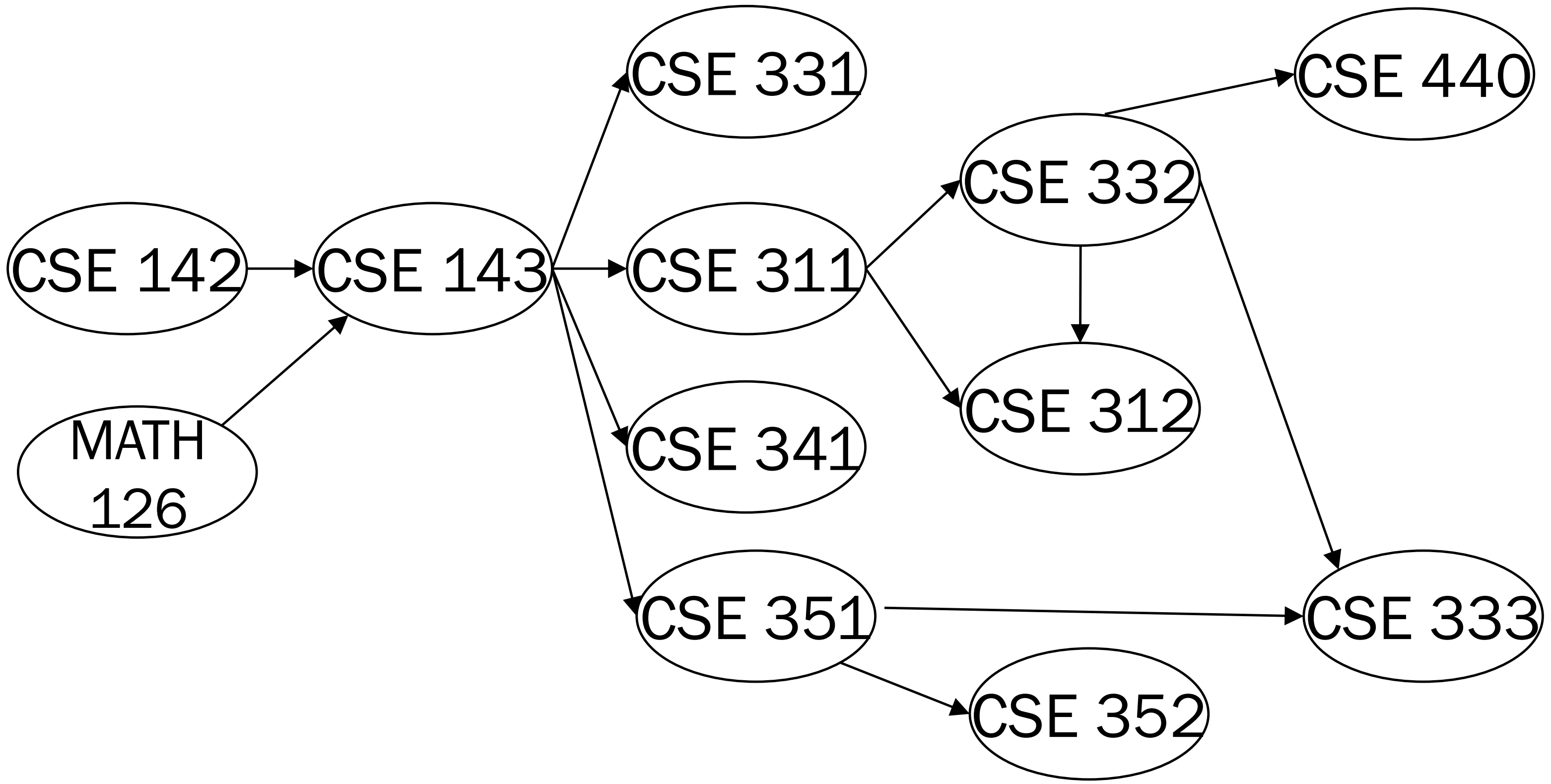
Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x											
In-degree	0	0	~~2~~ 1	1	2	1	1	2	1	1	1	1

Example

Output:

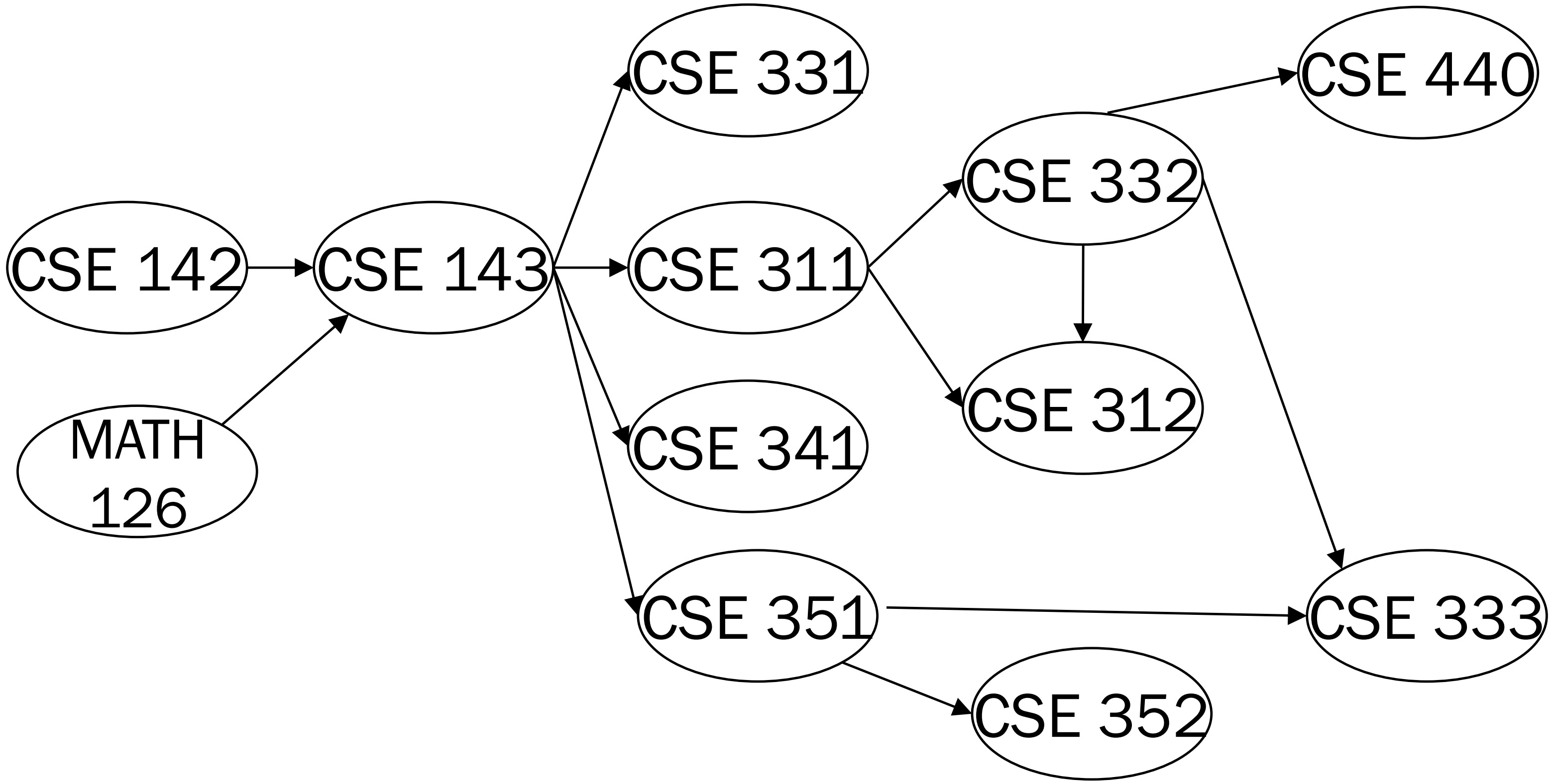
126

142



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

Example



Output:

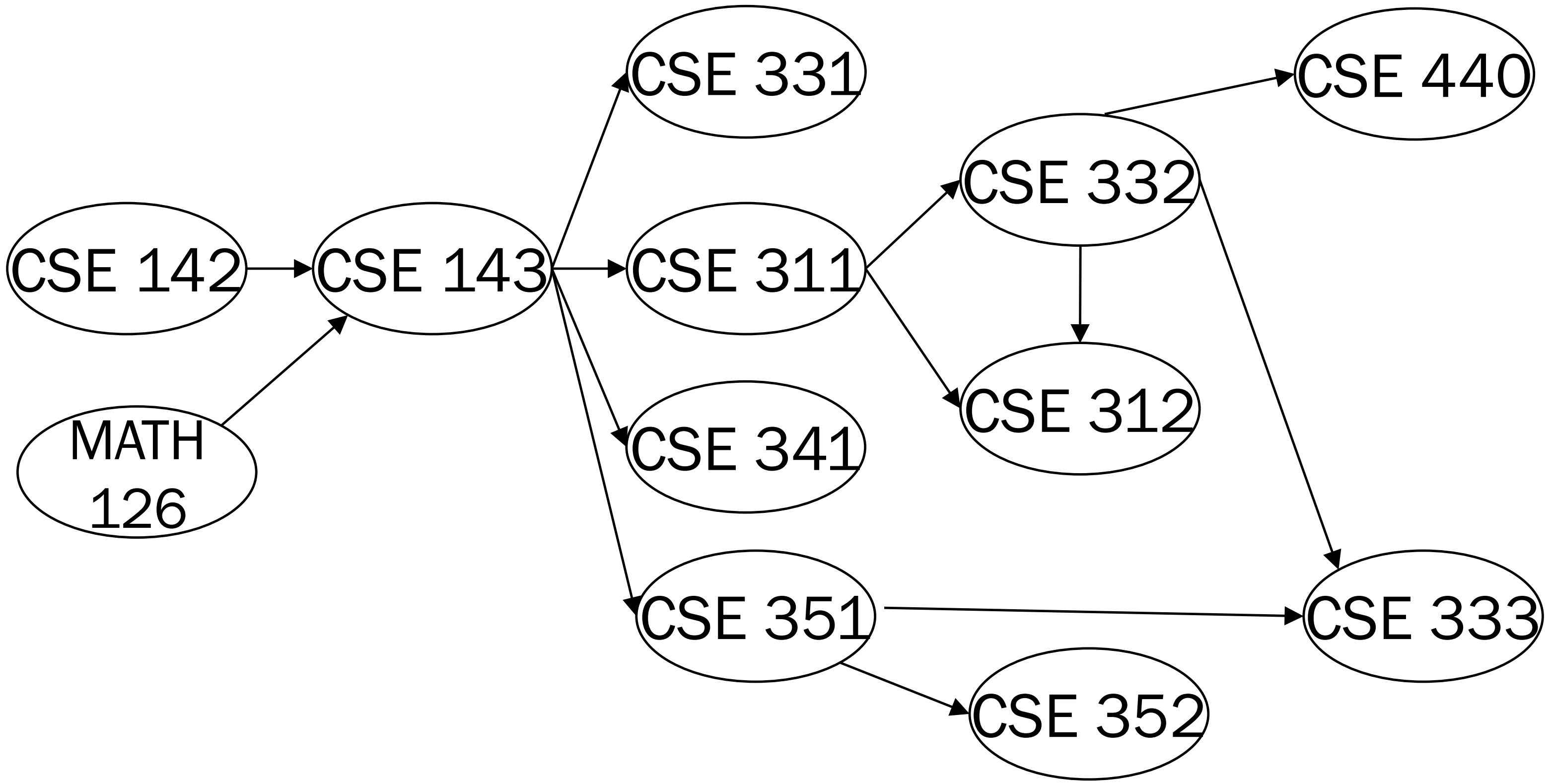
126

142

143

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

Example



Output:

126

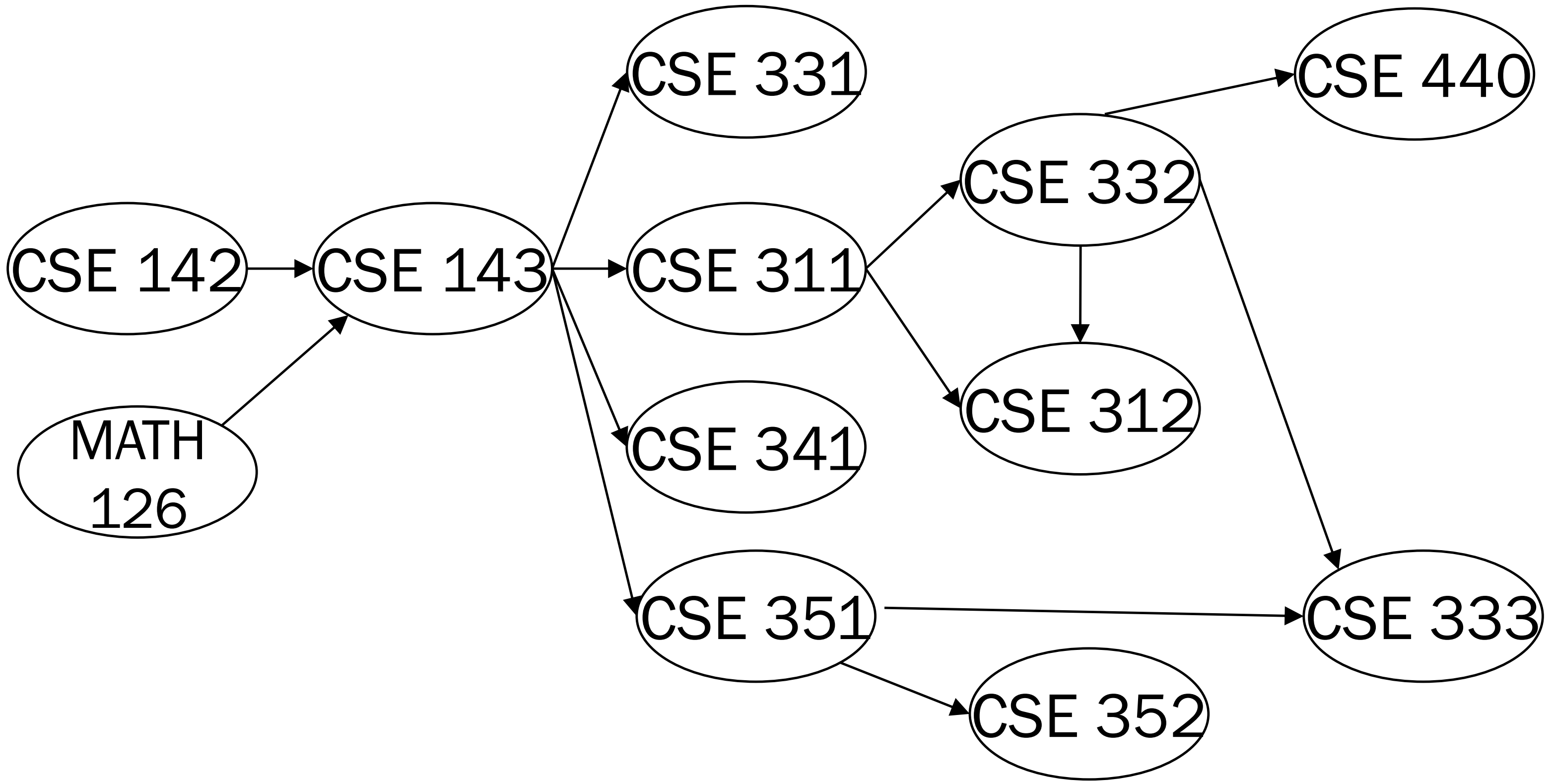
142

143

311

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

Example



Output:

126

142

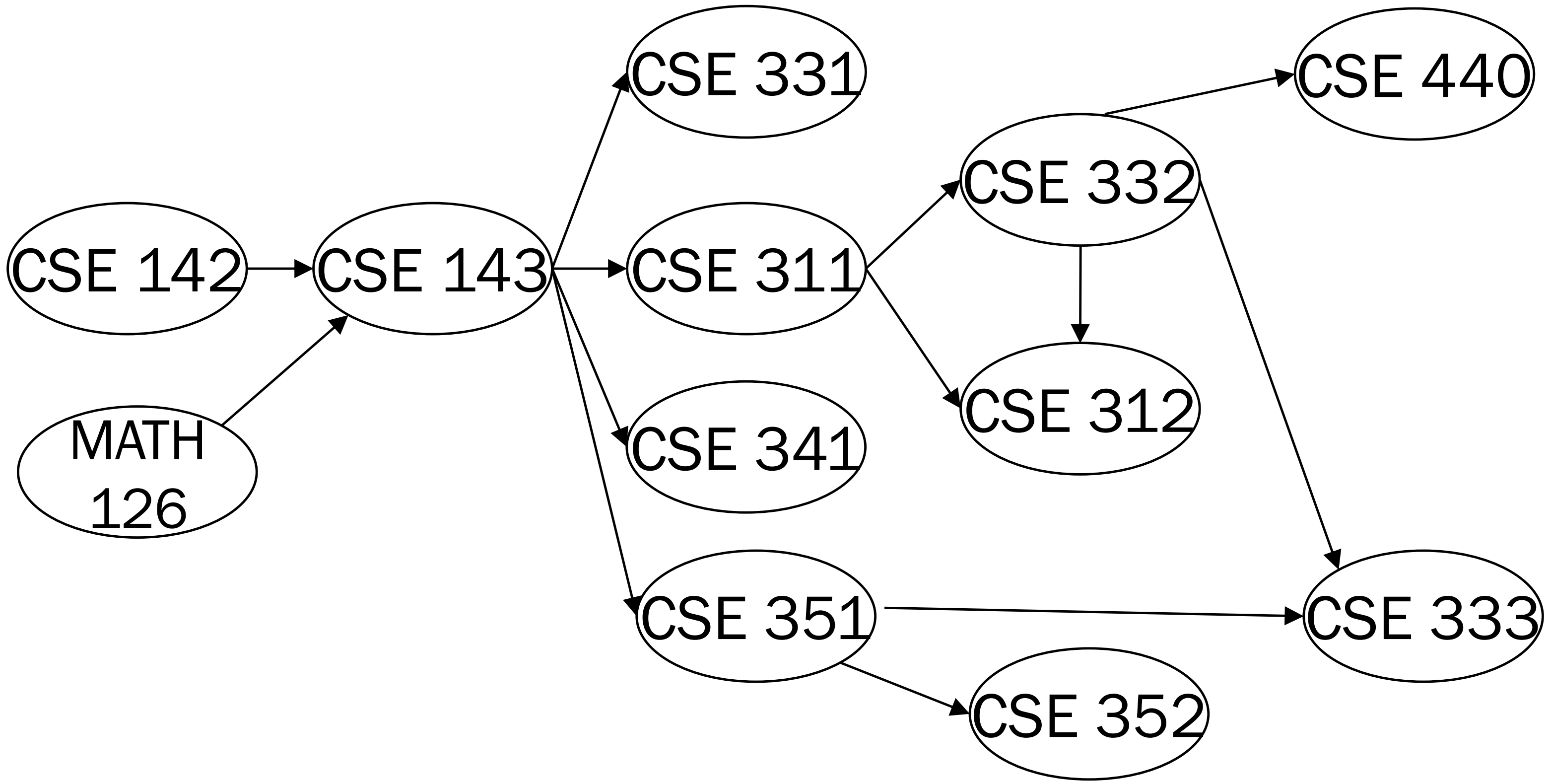
143

311

331

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

Example



Output:

126

142

143

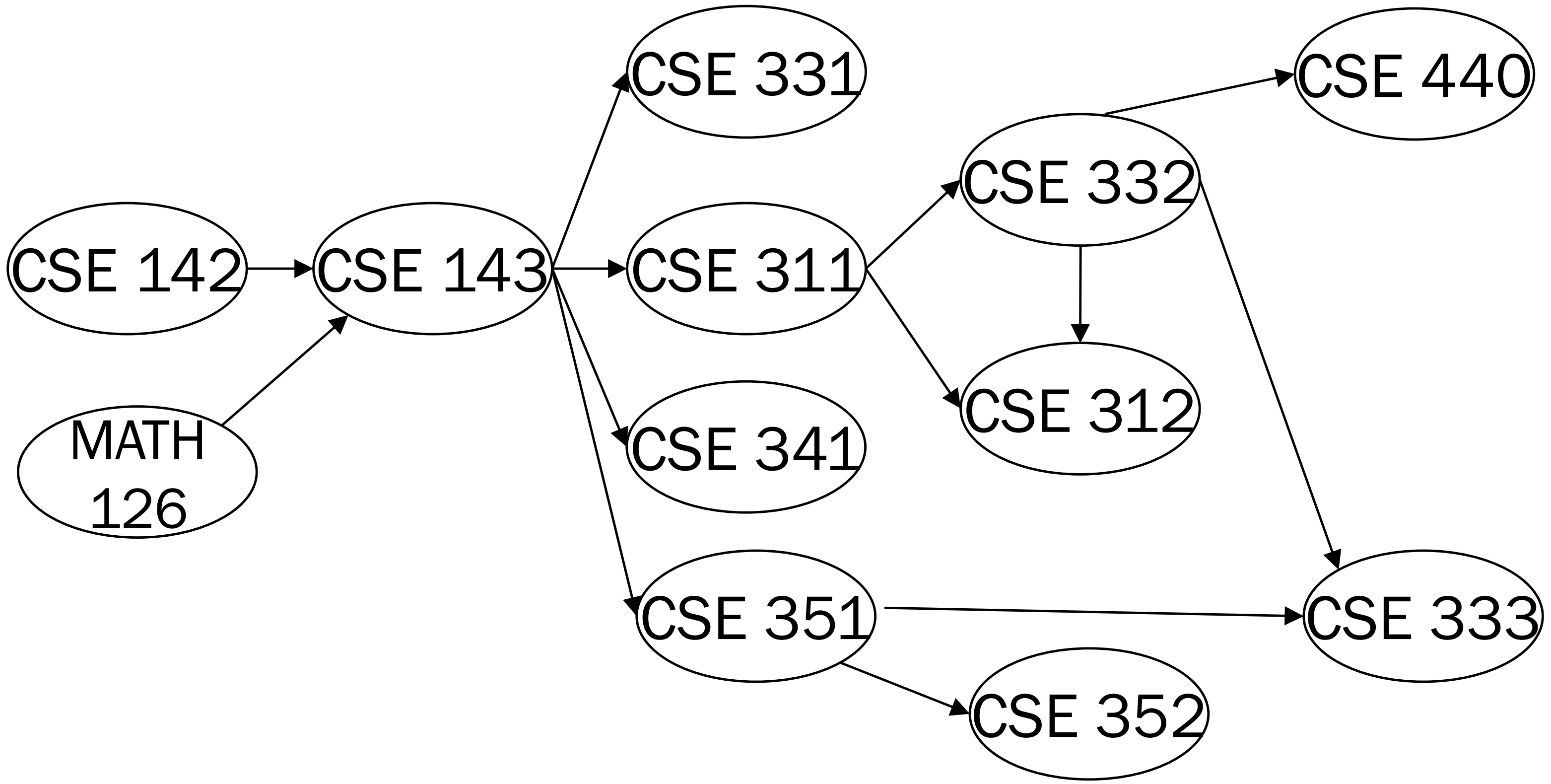
311

331

332

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

Example



Output:

126

142

143

311

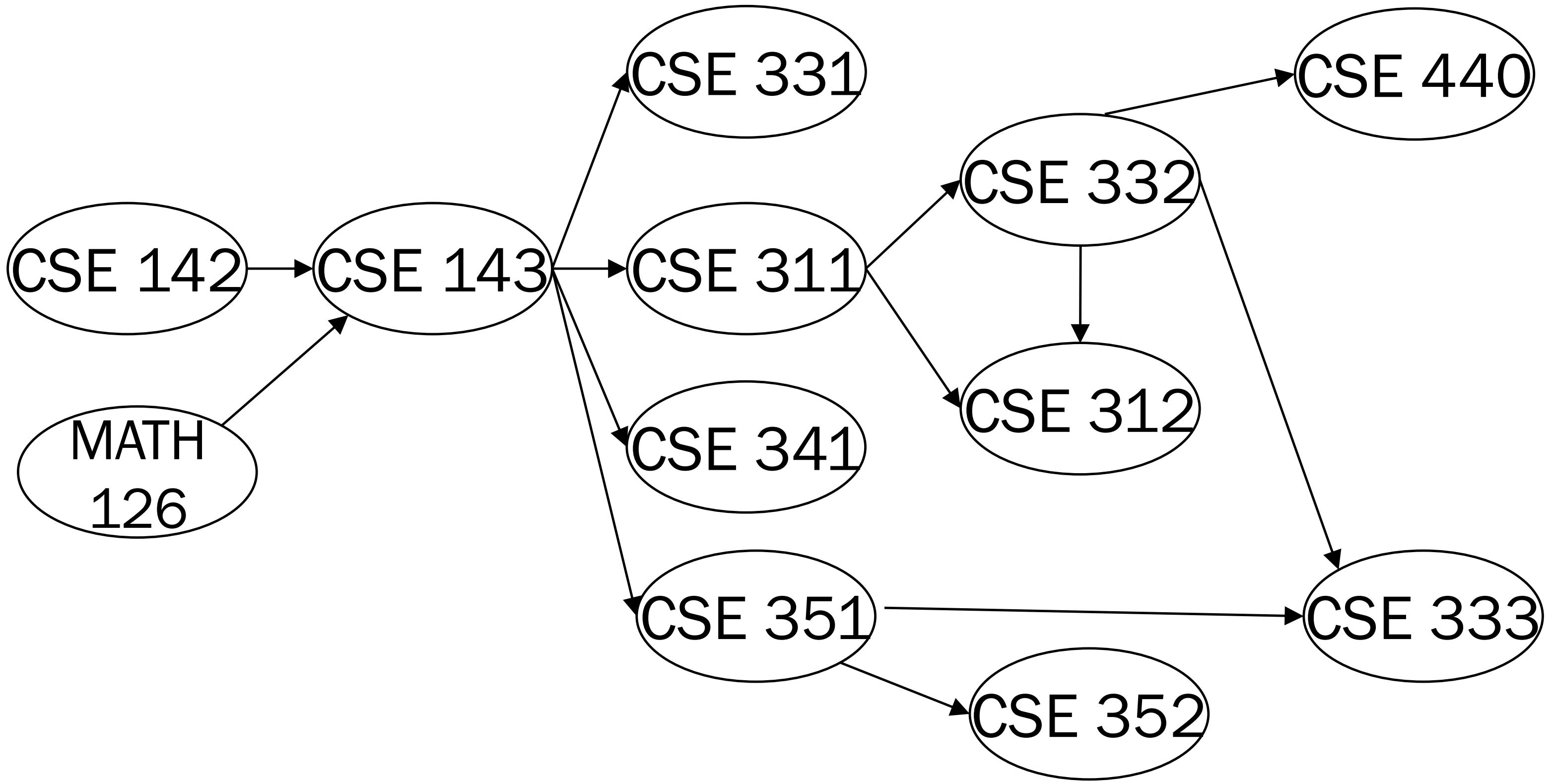
331

332

312

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

Example



Output:

126

142

143

311

331

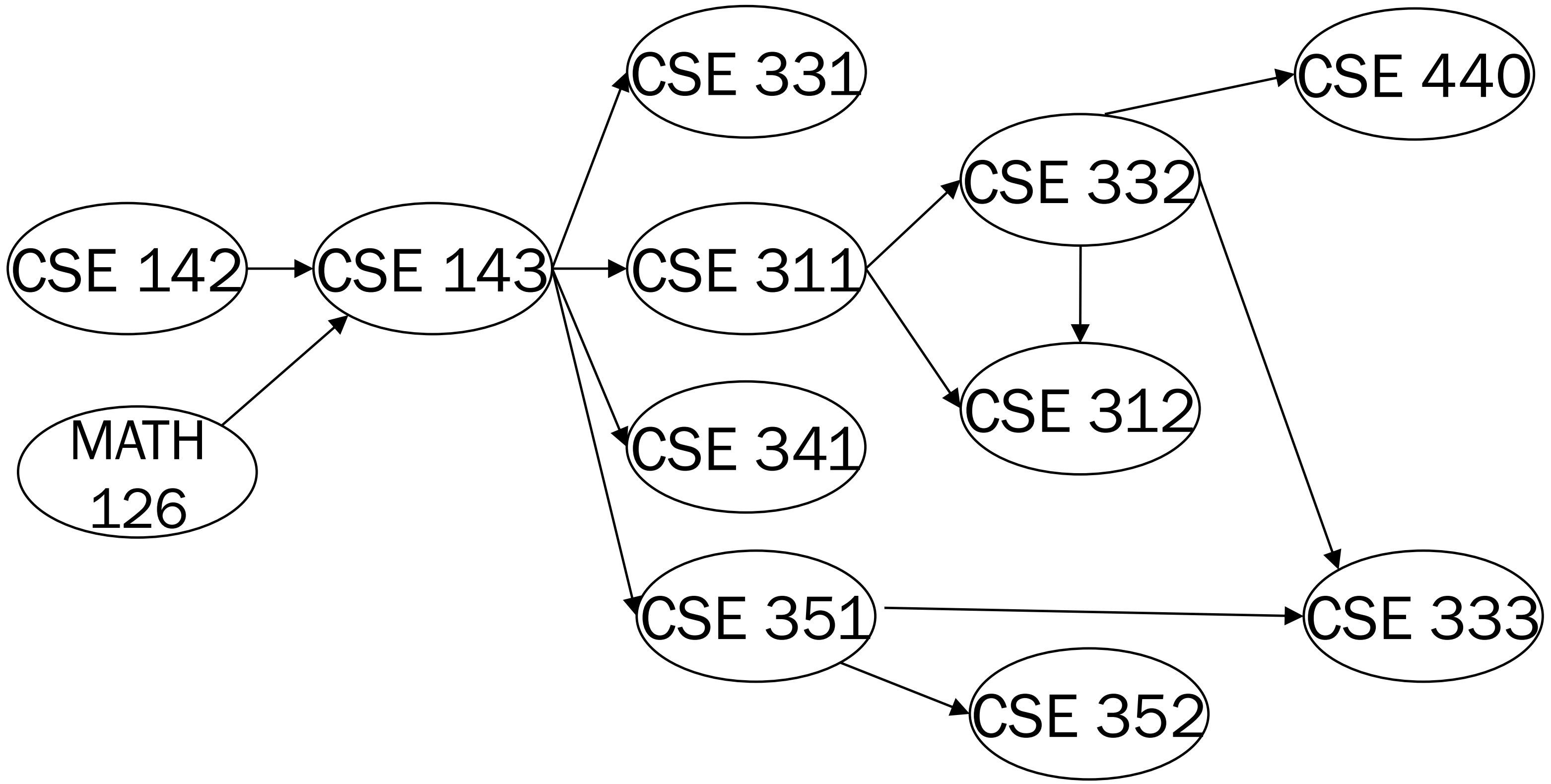
332

312

341

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	0	0	0	0	0	1	0	0	1	0

Example



Output:

126

142

143

311

331

332

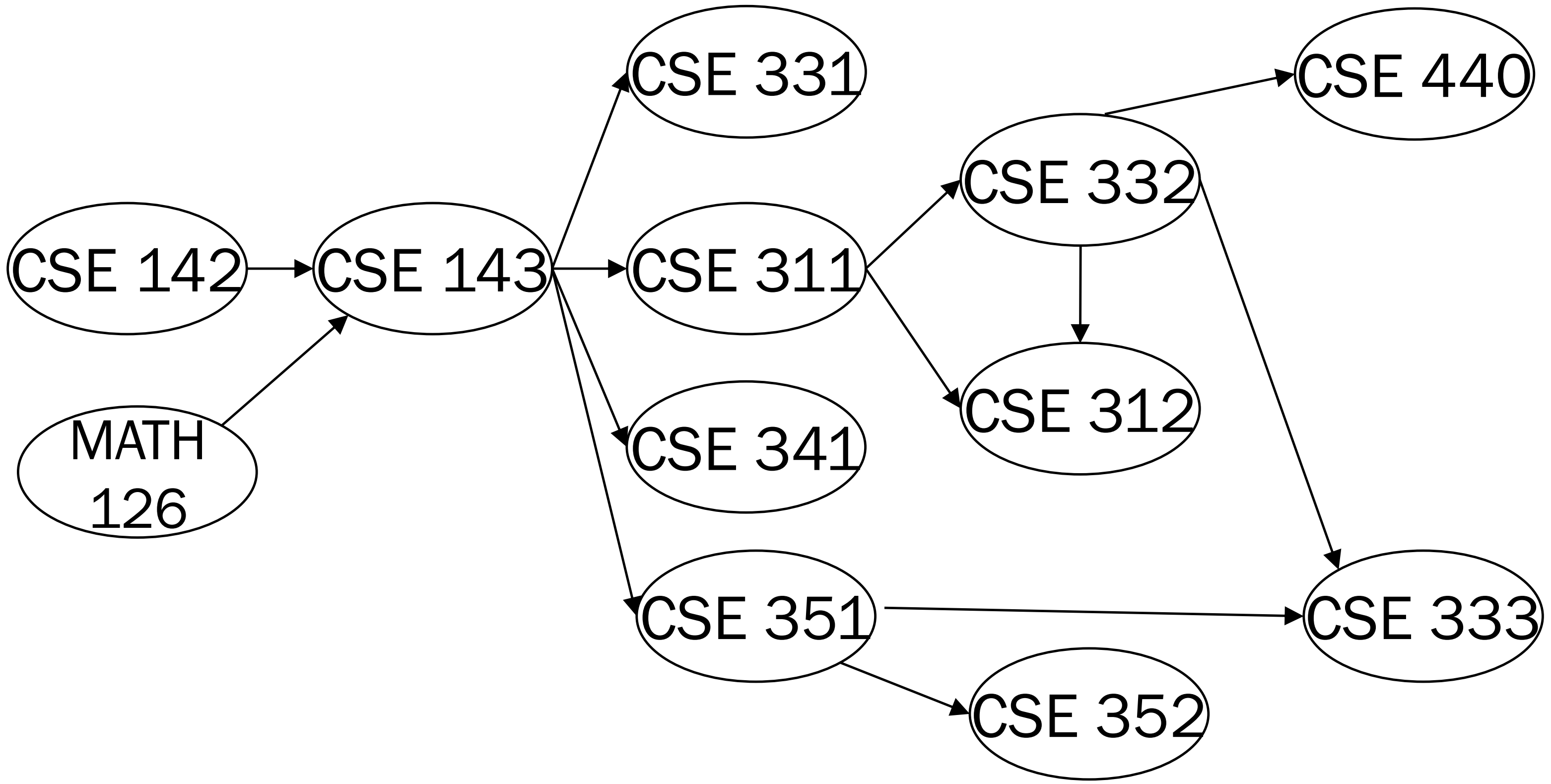
312

341

351

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	0	0	0	0	0	~~1~~ 0	0	0	~~1~~ 0	0

Example

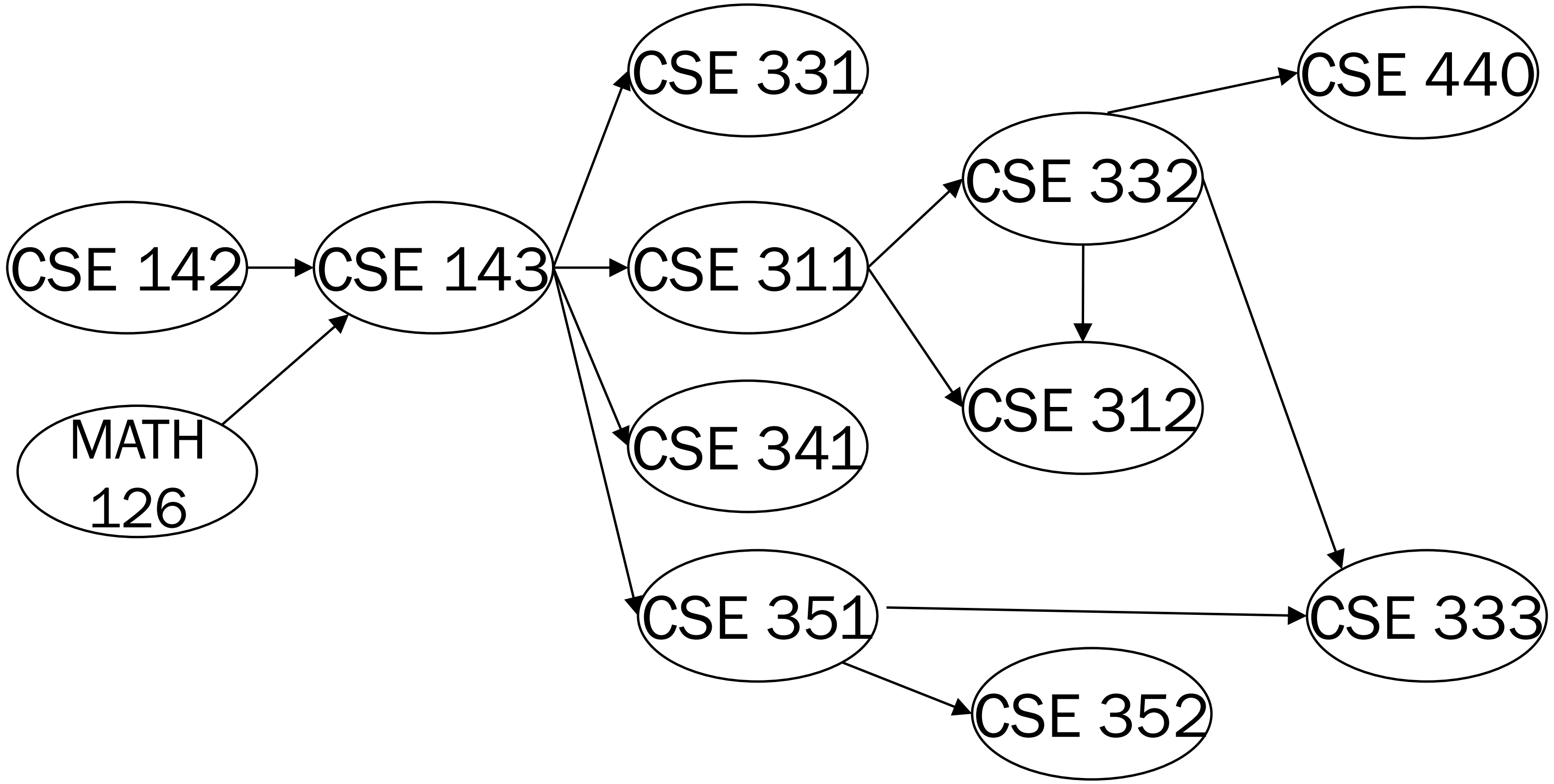


Output:

126
142
143
311
331
332
312
341
351
333

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

Example



Output:

126

142

143

311

331

332

312

341

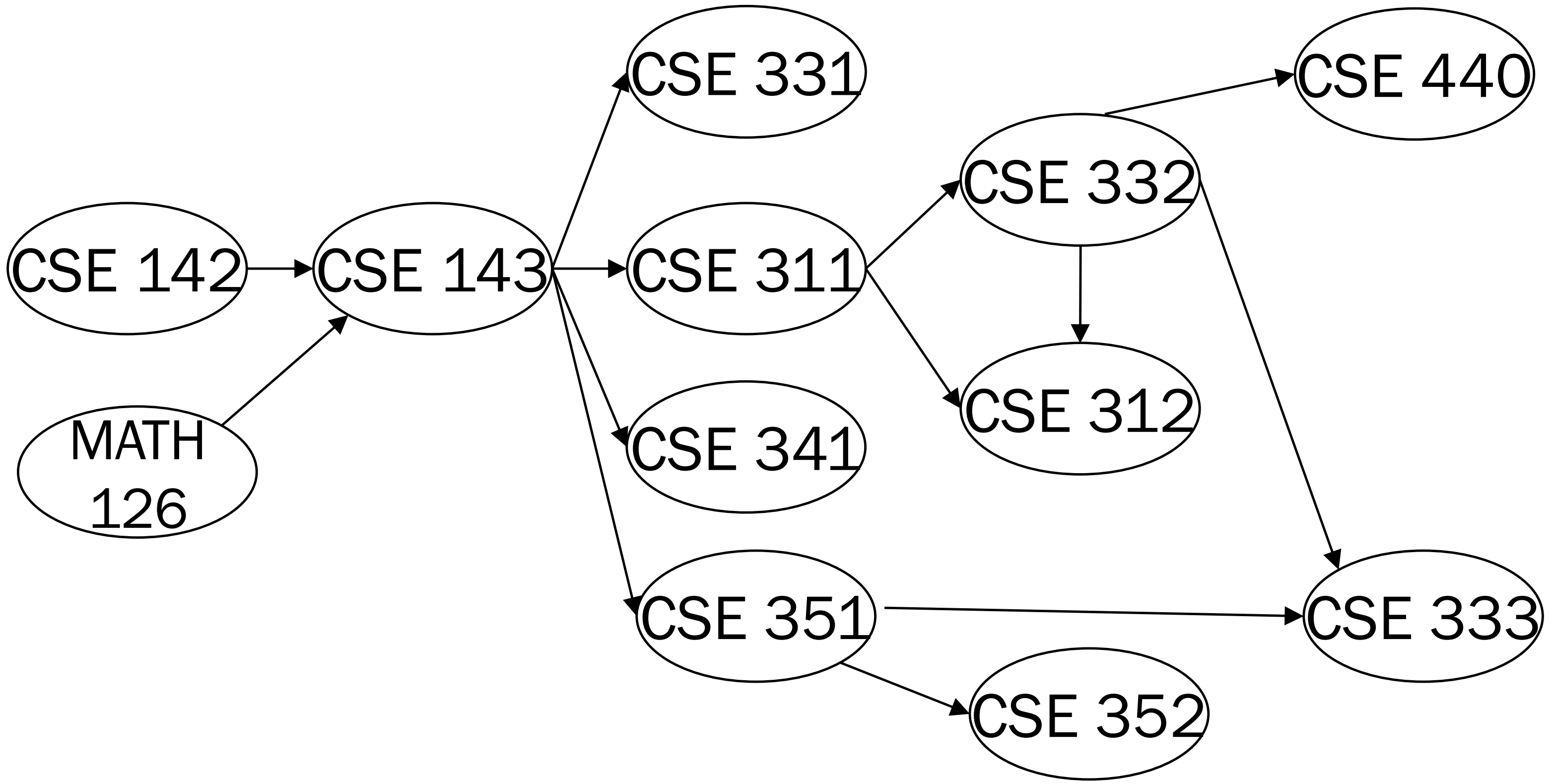
351

333

352

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	
In-degree	0	0	0	0	0	0	0	0	0	0	0	0

Example

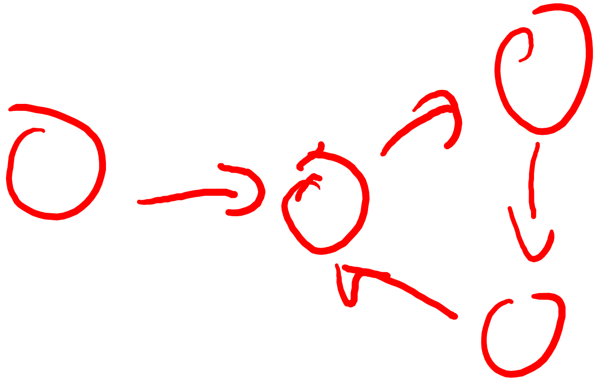


Output:

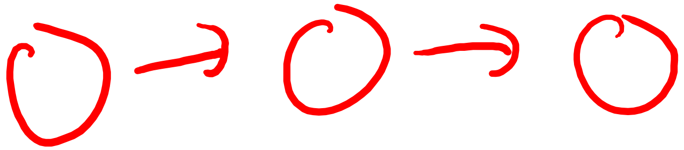
126
142
143
311
331
332
312
341
351
333
352
440

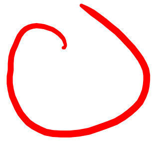
Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	x
In-degree	0	0	0	0	0	0	0	0	0	0	0	0

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
- What DAGs have exactly 1 topological ordering?





Topological Sort: Running time?

```
labelEachVertexWithItsInDegree();

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

$\rightarrow O(V+E)$

V

V

1

d

1

$O(V+E+V(V+1+d))$

$O(V+E+V^2+V+E)$

$O(E+V^2) \rightarrow O(V^2+V^2)$ $O(V^2)$

Topological Sort: Running time?

```
labelEachVertexWithItsInDegree();

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

- What is the worst-case running time?
  - Initialization $O(|V| + |E|)$ (assuming adjacency list)
  - 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 **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?

pollev.com/artliu

```
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);
  }
}
```

$O(V+E)$ V 1 1 d 1 1 1

$O(V+E + V(2+3d))$

$O(V+E + 2V + 3E)$

$O(V+E)$

$0 \ldots V^2$ $O(V+V^2)$

Topological Sort(optimized): Running time?

```
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: $O(|V|+|E|)$ (assuming adjacency list)
  - 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!

Topological Sort Uses

- Figuring out how to finish your degree
- Determining the order to compile files using a Makefile
- Determining what order a processor should execute threads
- Determining what assignment you should work on next

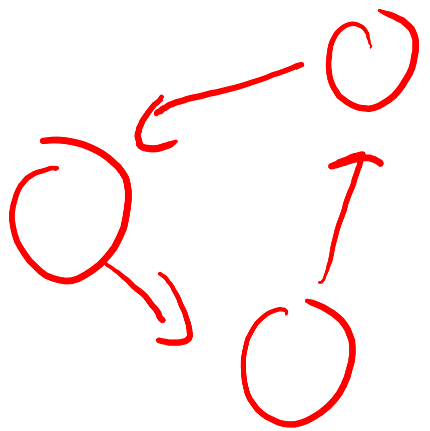
- In general, taking a dependency graph and coming up with an order of execution

Another graph algorithm!

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.



Basic idea:

- Keep following adjacent nodes
- But "mark" nodes after visiting them, so the traversal terminates, and we process each reachable node exactly once

Graph Traversal: Abstract Idea

O(E)

```
traverseGraph(Node start) {
    Set pending = emptySet();     1
    pending.add(start)     1
    mark start as visited     1
    while(pending is not empty) {     V
      next = pending.remove()     1
      for each node u adjacent to next     d
          if(u is not marked) {     1
            mark u     1
            pending.add(u)     1
          }
    }
}
```

O(V·d) = O(E)

Running time and options

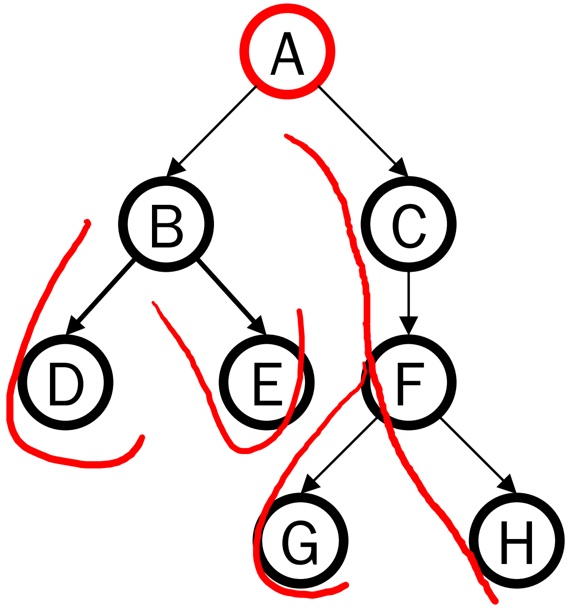
- Assuming add and remove are $O(1)$, entire traversal is $O(|E|)$
  - Use an adjacency list representation

- 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"

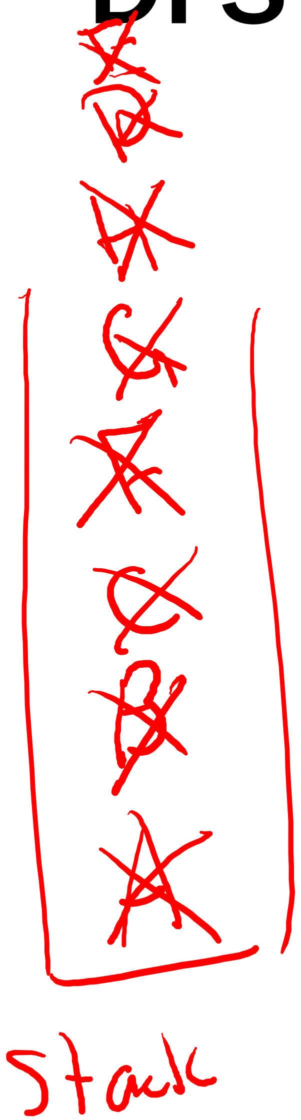


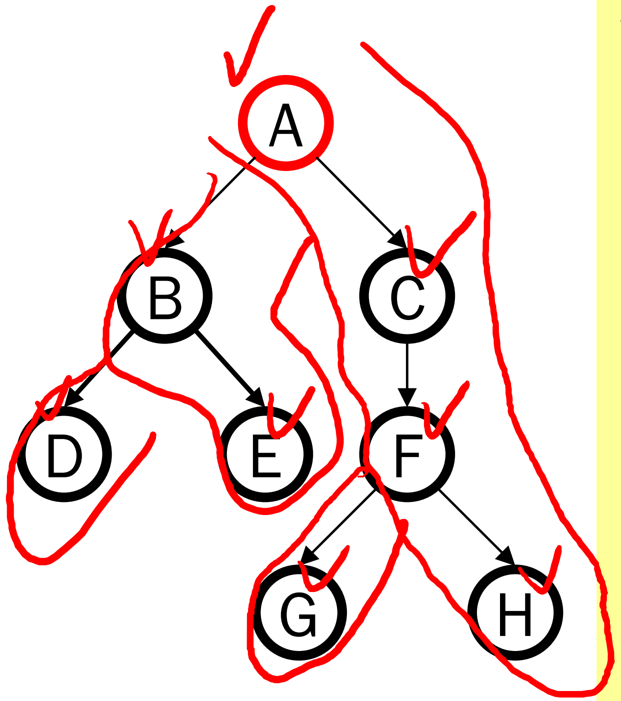

```
DFS(Node start) {
  mark and "process"(e.g. print) start
   for each node u adjacent to start
     if u is not marked
       DFS(u)
}
```

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



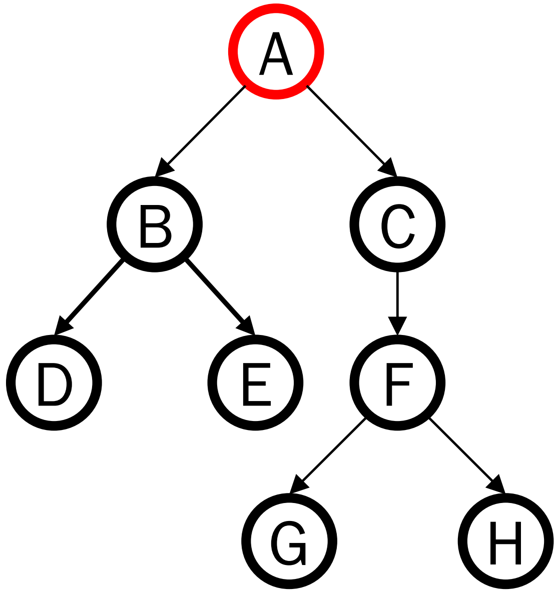



```
DFS2(Node start) {
  initialize stack s to hold start
  mark start as visited
  while(s is not empty) {
    next = s.pop() // and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and push onto s
  }
}
```

Order processed: A, C, F, H, G, B, E, D

- A different but perfectly fine traversal

DFS with a stack, Example: trees

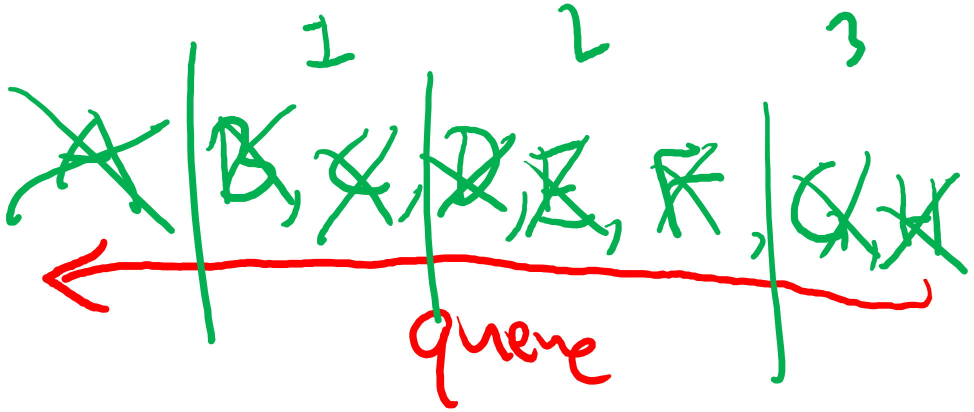


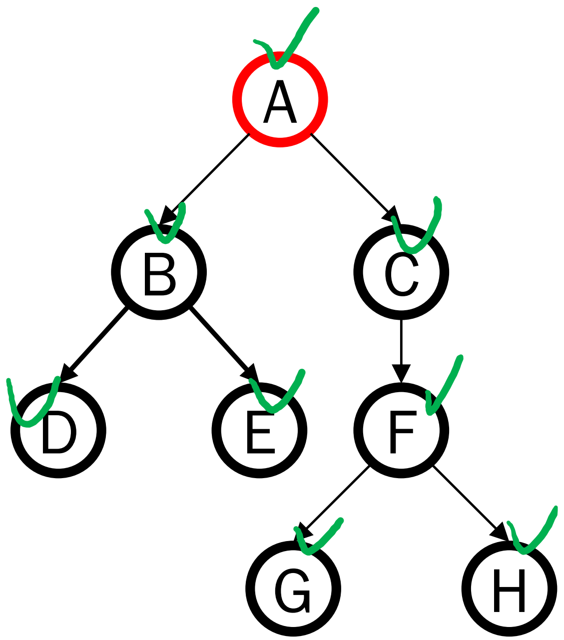

```
DFS2(Node start) {
  initialize stack s to hold start
  mark start as visited
  while(s is not empty) {
    next = s.pop() // and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and push onto s
  }
}
```

Order processed: A, C, F, H, G, B, E, D

- A different but perfectly fine traversal

BFS with a queue, Example: trees



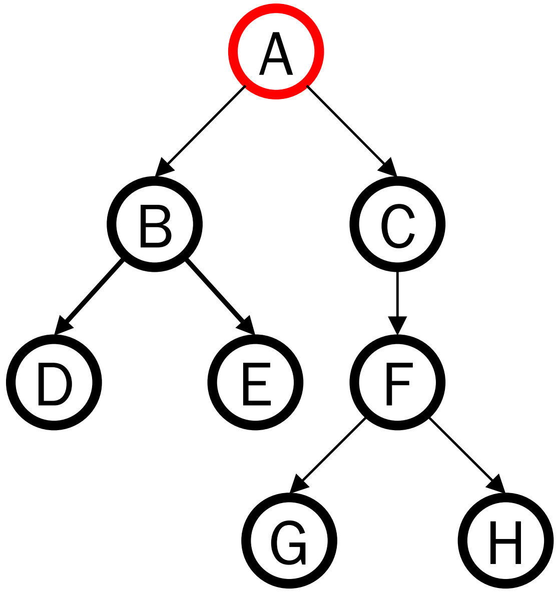



```
BFS(Node start) {
  initialize queue q to hold start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue()// and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and enqueue onto q
  }
}
```

Order processed: A, B, C, D, E, F, G, H

- A "level-order" traversal

BFS with a queue, Example: trees



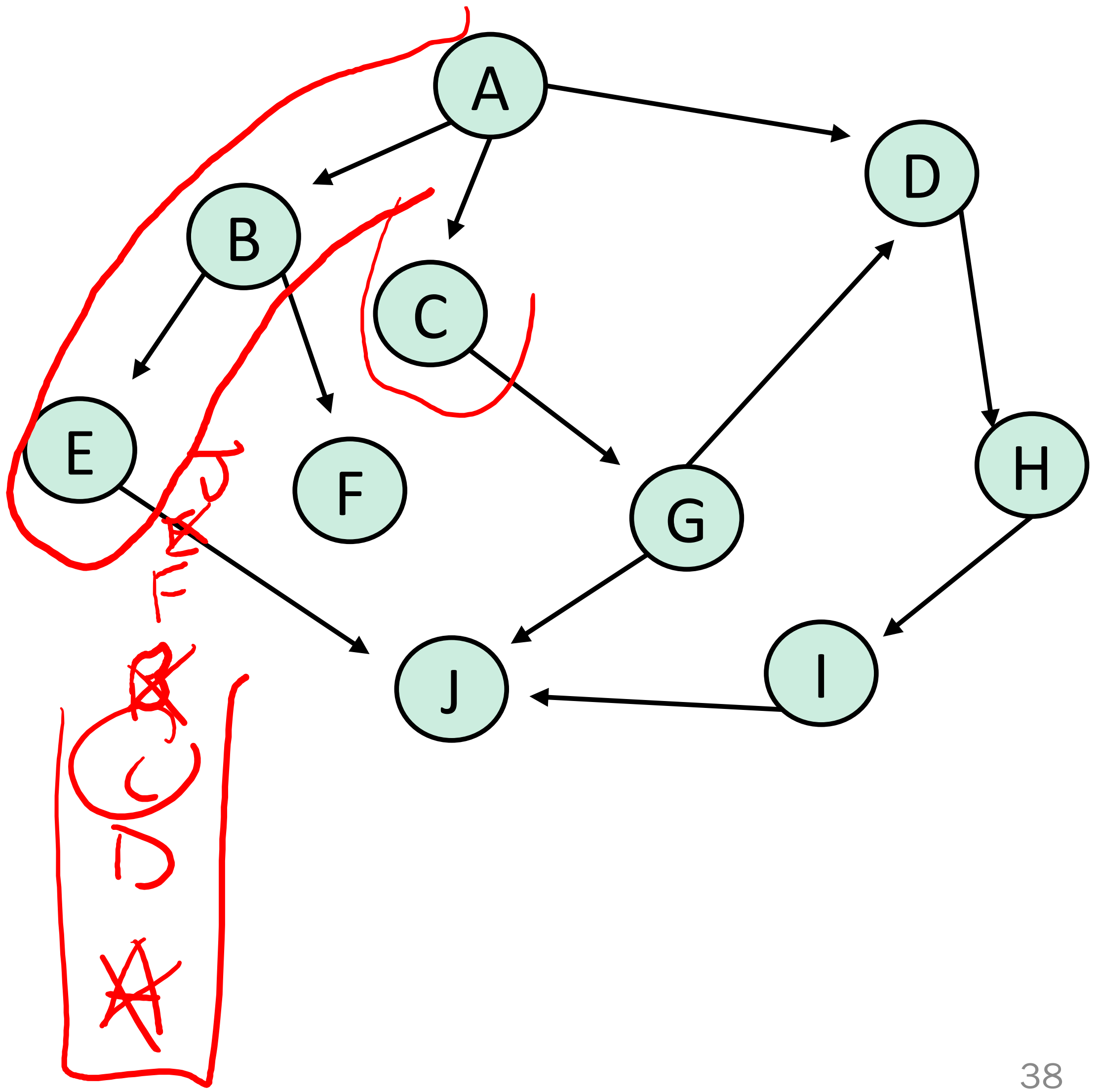

```
BFS(Node start) {
  initialize queue q to hold start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue()// and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and enqueue onto q
  }
}
```

Order processed: A, B, C, D, E, F, G, H

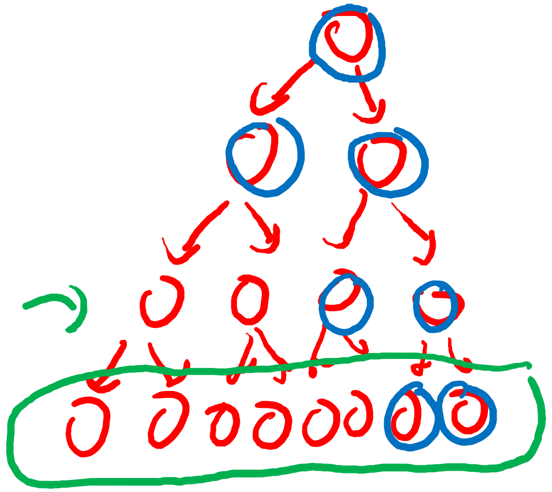
- A "level-order" traversal

For each of the following, indicate what traversal could have processed the graph in that order

1. A, D, H, I, J, C, G, B, F, E DFS
2. A, B, C, D, E, F, G, H, J, I BFS
3. A, D, C, B, H, G, F, E, I, J BFS
4. A, B, E, C, G, F, J, D, H, I None



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.

IDDFS and AI

- IDDFS ideas can be applied to AI search algorithms to prune out bad branches earlier instead of traversing them too far
  - Helps us figure out how to "break ties" when picking a path

Take CSE473 AI
(CSE415 AI Non-Majors)

Saving the path

- Our graph traversals can answer the “reachability question”:
  - “***Is there*** a path from node x to node y?”

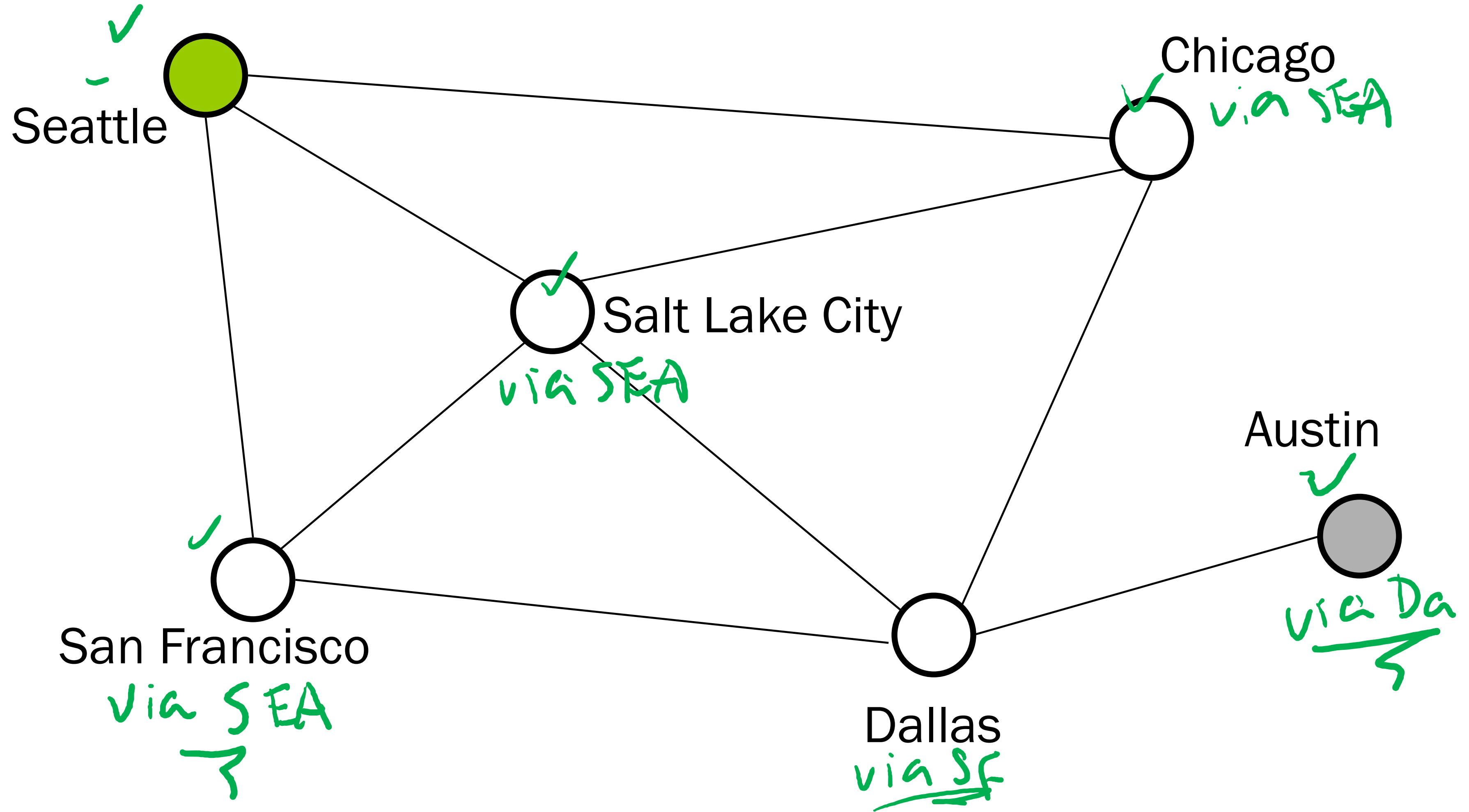
- Q: But what if we want to ***output the actual path***?

- 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**)

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