CSE 373

MAY 8TH – DIJKSTRAS

GRAPHS REVIEW

- **What is some of the terminology for graphs and what do those terms mean?**
  - Vertices and Edges
  - Directed v. Undirected
  - In-degree and out-degree
  - Connected
  - Weighted v. unweighted
  - Cyclic v. acyclic
  - DAG: Directed Acyclic Graph

TRAVERSALS

- **For an arbitrary graph and starting node v, find all nodes *reachable* from v.**
  - There exists a path from v
  - Doing something or "processing" each node
  - Determines if an undirected graph is connected? If a traversal goes through all vertices, then it is connected
- **Basic idea**
  - Traverse through the nodes like a tree
  - Mark the nodes as visited to prevent cycles and from processing the same node twice

ABSTRACT IDEA IN PSEUDOCODE

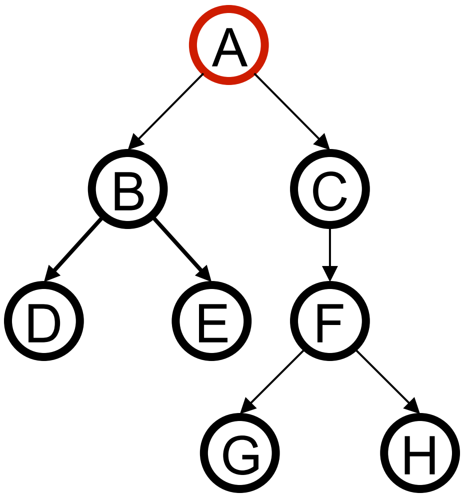
```
void traverseGraph(Node start) {
      Set pending = emptySet()
      pending.add(start)
     mark start as visited
     while(pending is not empty) {
       next = pending.remove()
       for each node u adjacent to next
          if (u is not marked visited) {
             mark u
             pending.add(u)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

RUNTIME AND OPTIONS

- **Assuming we can add and remove from our "pending" DS in O(1) time, the entire traversal is O(|E|)**
- **Our traversal order depends on what we use for our pending DS.**
  - Stack : DFS
  - Queue: BFS
- **These are the main traversal techniques in CS, but there are others!**

EXAMPLE: TREES

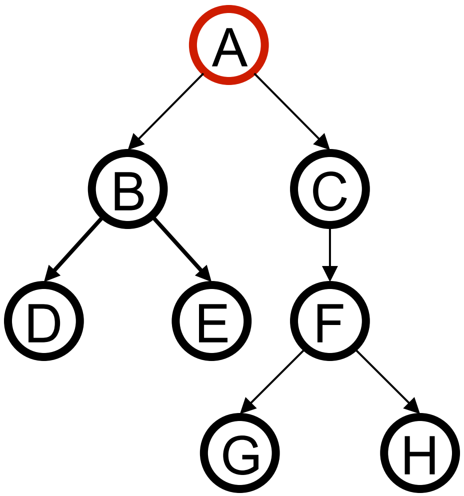
A tree is a graph and make DFS and BFS are easier to "see"




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

- 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

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
  }
}
```

- A, C, F, H, G, B, E, D
- A different but perfectly fine depth traversal

COMPARISON

Breadth-first always finds shortest length paths, i.e., "optimal solutions"

- Better for "what is the shortest path from **`x`** to **`y`**"

But 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 **`d*p`** elements
- But a queue for BFS may hold *O*(|V|) nodes

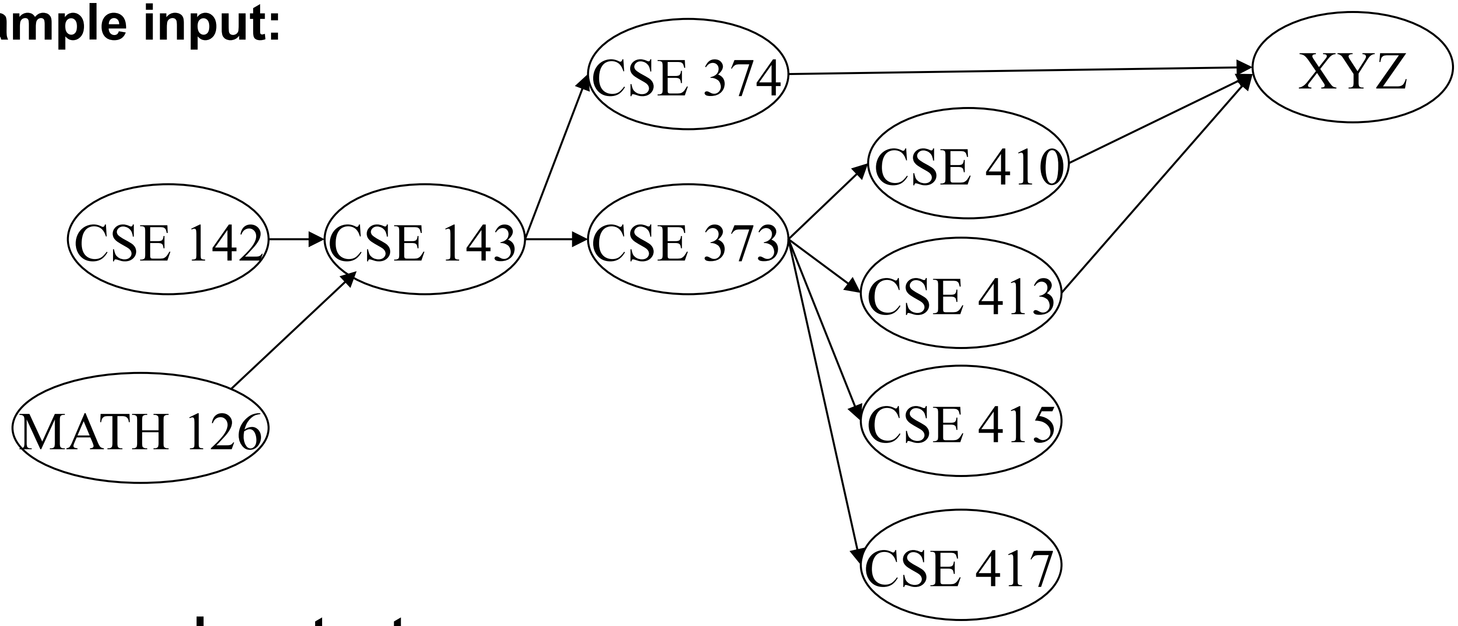
A third approach (useful in Artificial Intelligence)

- *Iterative deepening (IDFS)*:
  - Try DFS but disallow 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.

TOPOLOGICAL SORT

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

Example input:



One example output:

126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415

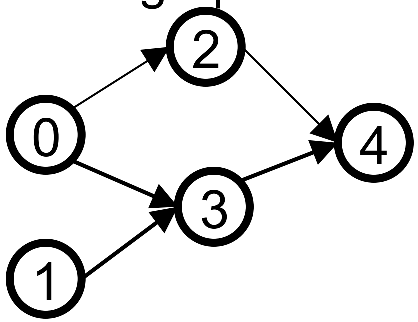
QUESTIONS AND COMMENTS

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
- Graph with 5 topological orders:



Do some DAGs have exactly 1 answer?

- Yes, including all lists

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

USES OF TOPOLOGICAL SORT

Figuring out how to graduate

Computing an order in which to recompute cells in a spreadsheet

Determining an order to compile files using a Makefile

In general, taking a dependency graph and finding an order of execution

…

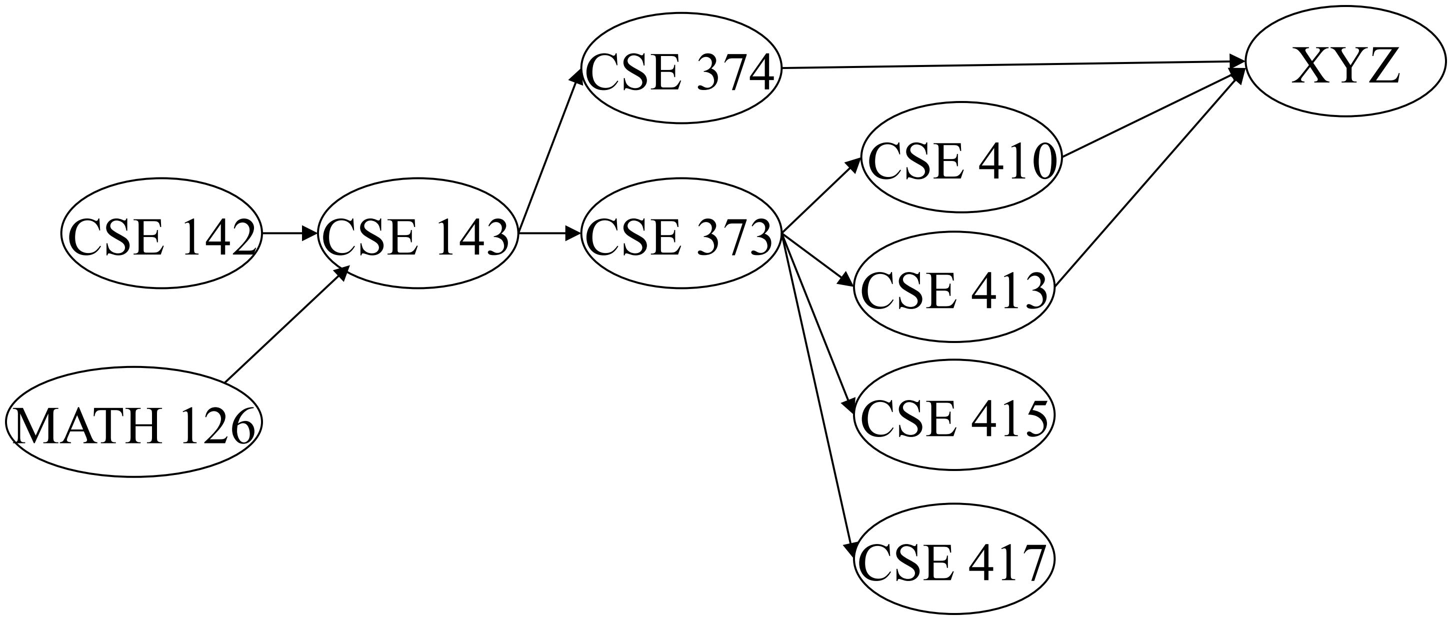
TOPOLOGICAL SORT

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 **u** adjacent to **v** (i.e. **u** such that (**v**,**u**) in `E`), decrement the in-degree of **u**

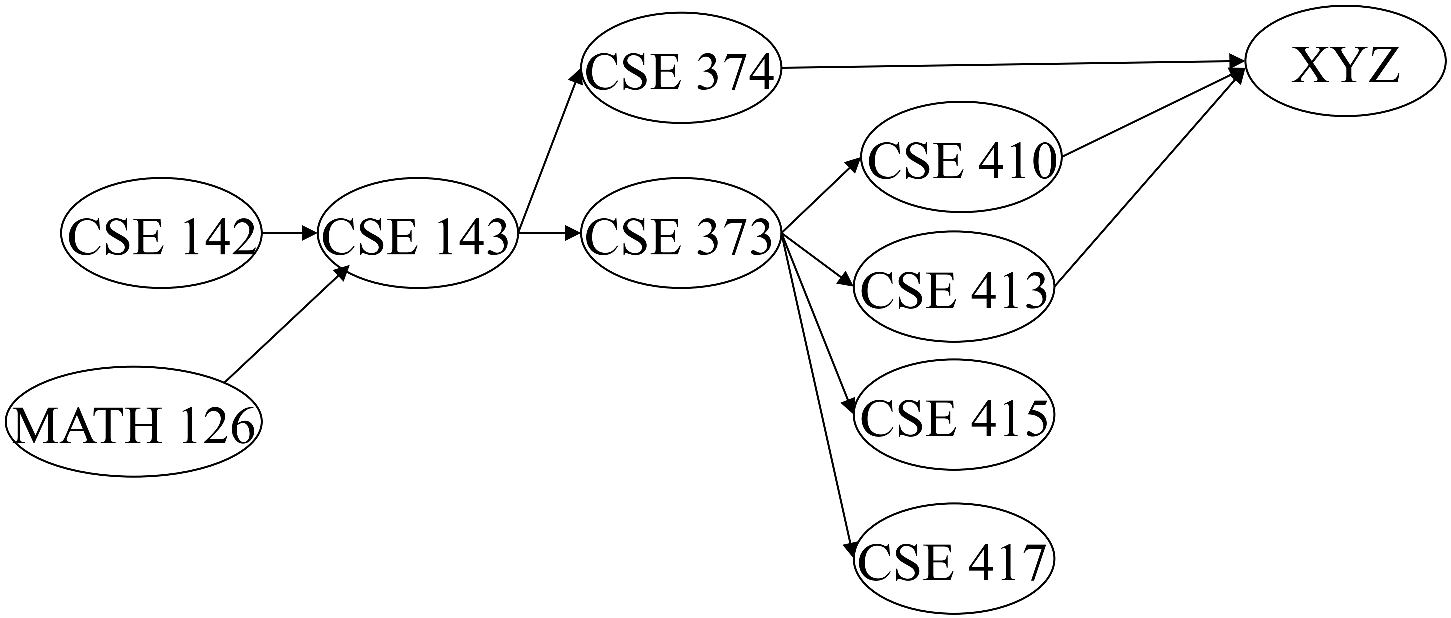
Example

Output
:



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?										
In-degree:	0	0	2	1	1	1	1	1	1	3

Example

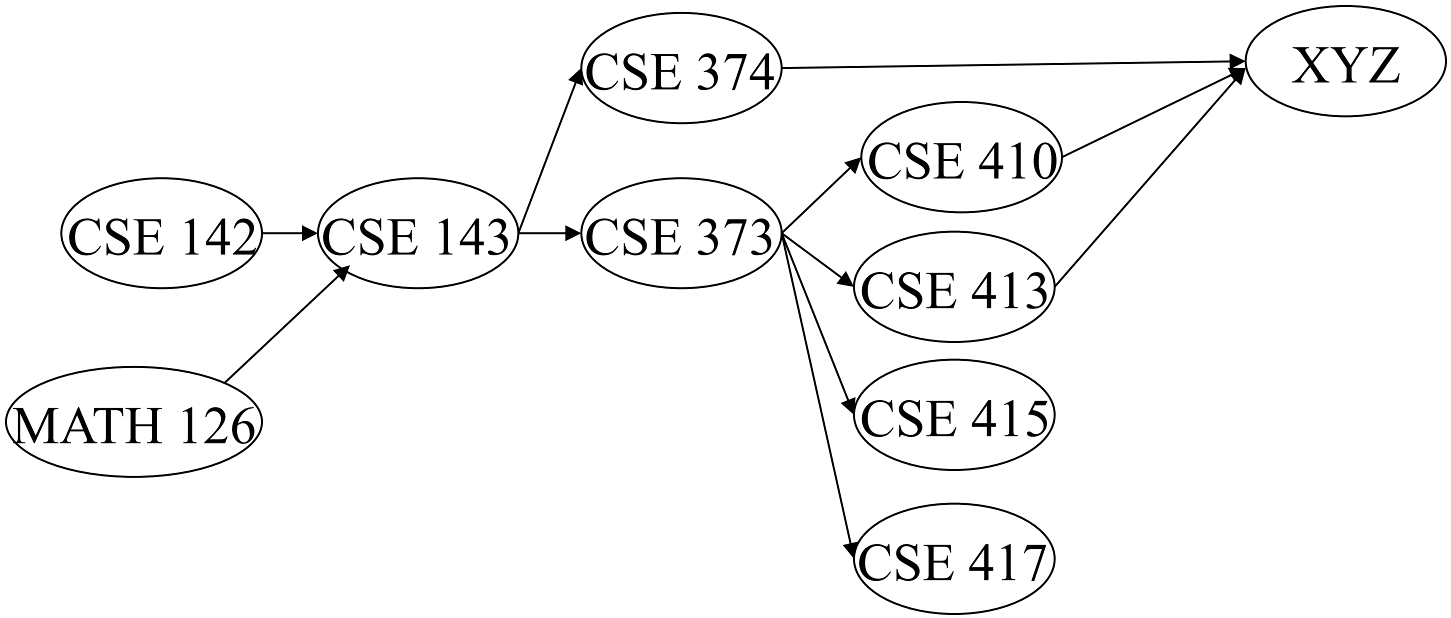


Output:

126

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x									
In-degree:	0	0	~~2~~ 1	1	1	1	1	1	1	3

Example



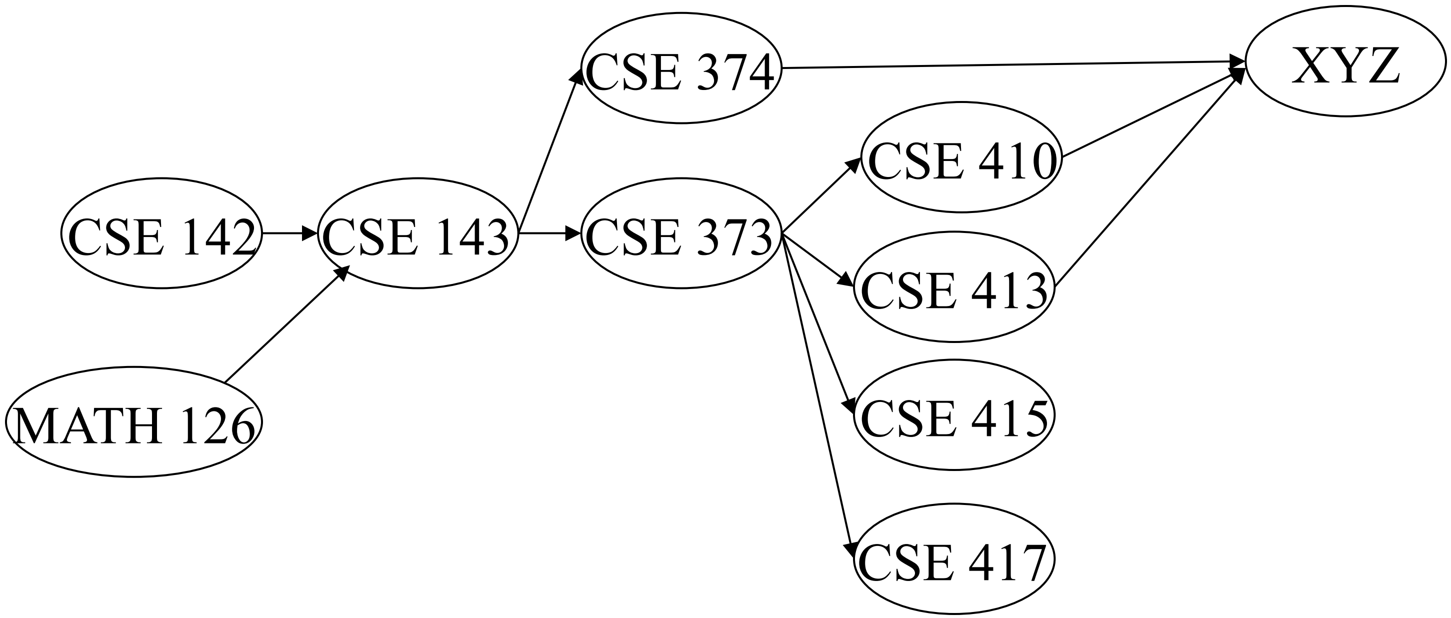
Output:

126

142

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x								
In-degree:	0	0	~~2~~ ~~1~~ 0	1	1	1	1	1	1	3

Example



Output:

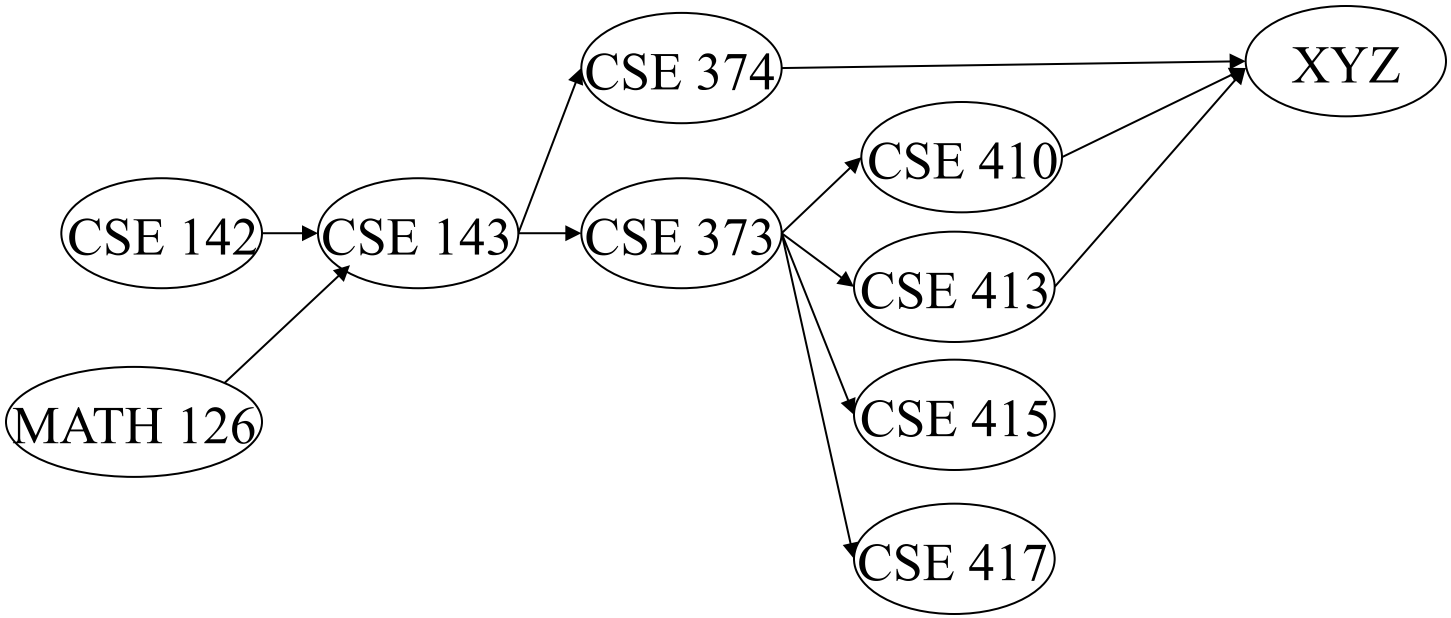
126

142

143

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x							
In-degree:	0	0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	1	1	1	1	3

Example



Output:

126

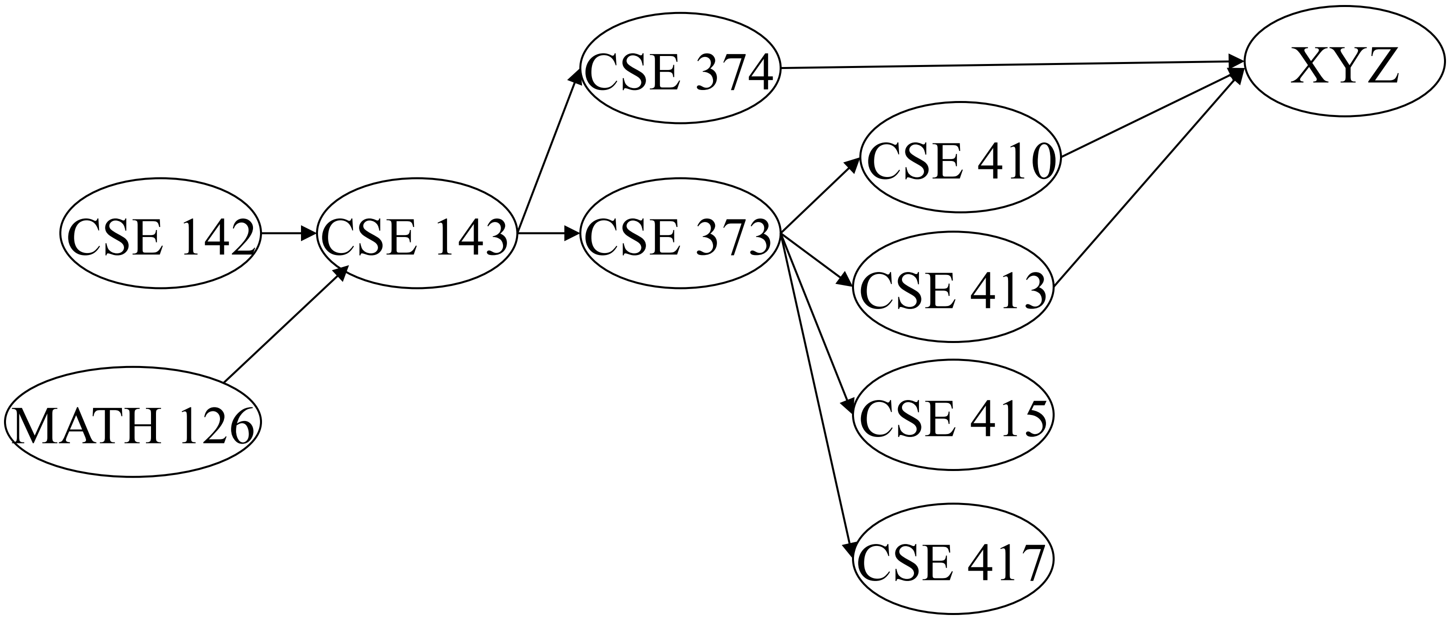
142

143

374

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x						
In-degree:	0	0	~~2~~	~~1~~	~~1~~	1	1	1	1	~~3~~
			~~1~~	0	0					2
			0							

Example



Output:

126

142

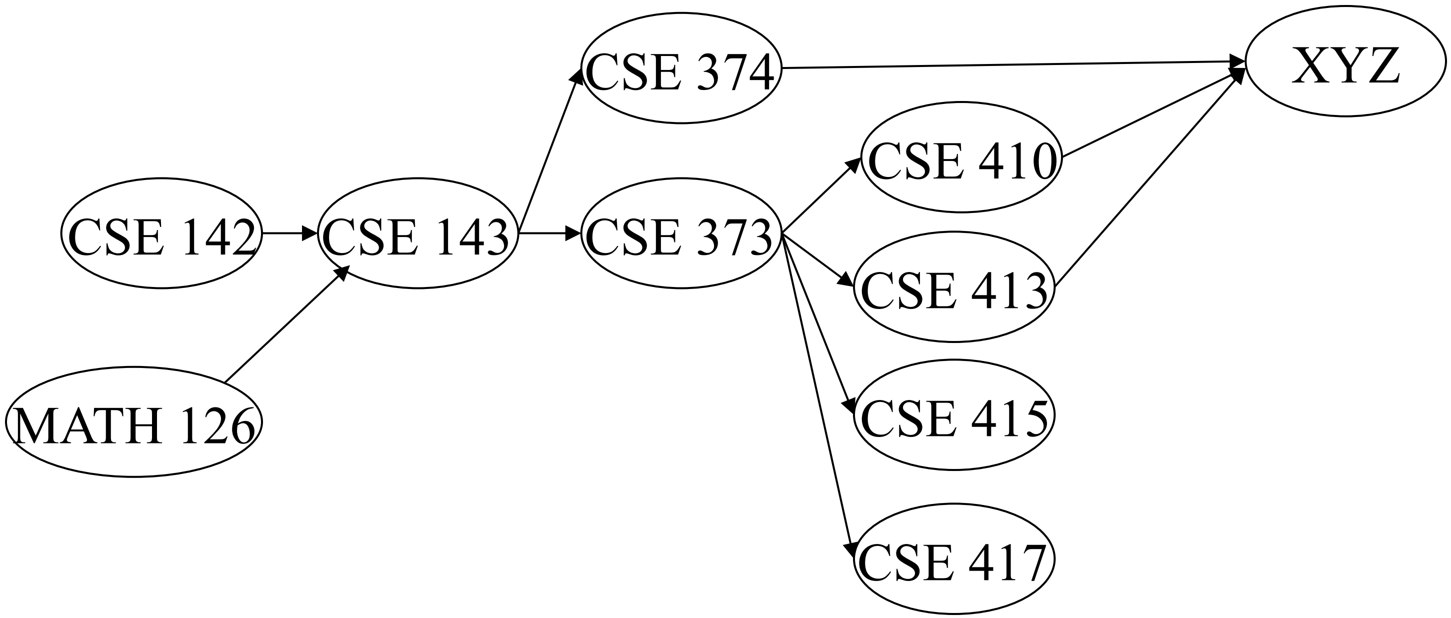
143

374

373

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x					
In-degree:	0	0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~3~~ 2

Example

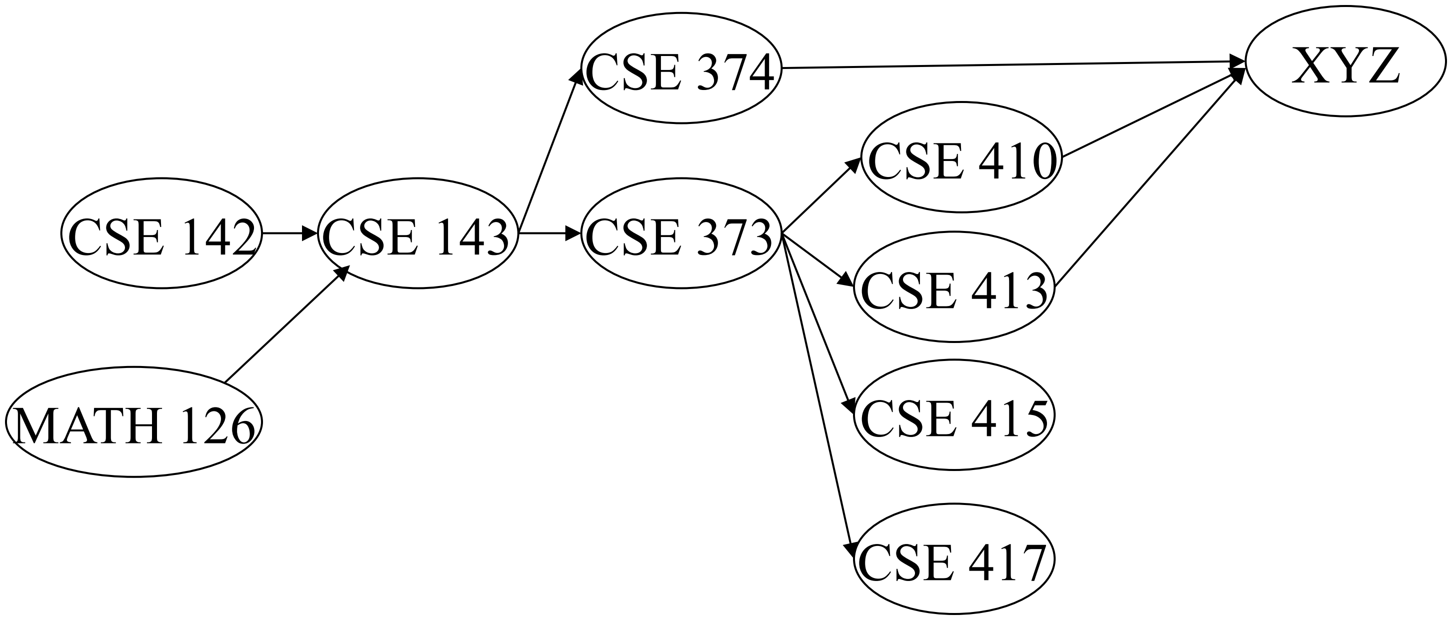


Output:

126
142
143
374
373
417

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x				x	
In-degree:	0	0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~3~~ 2

Example



Output:

126

142

143

374

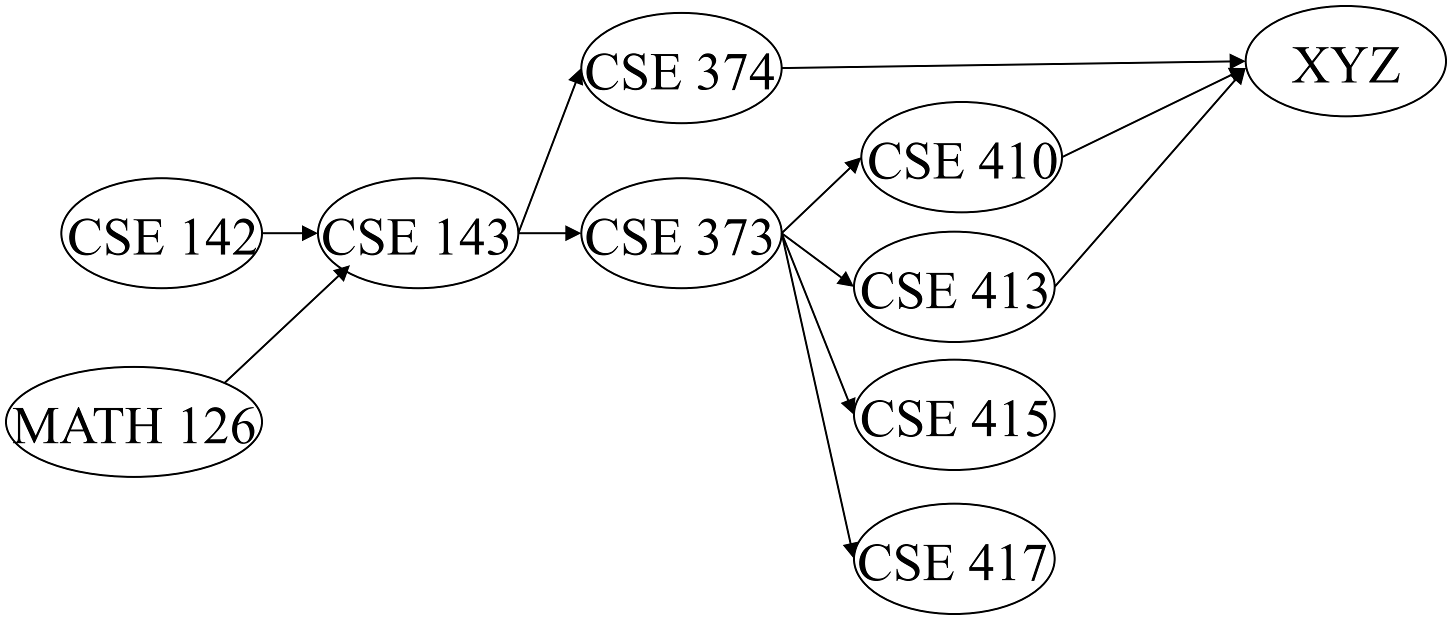
373

417

410

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x			x	
In-degree:	0	0	~~2~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~3~~
			~~1~~	0	0	0	0	0	0	~~2~~
			0							1

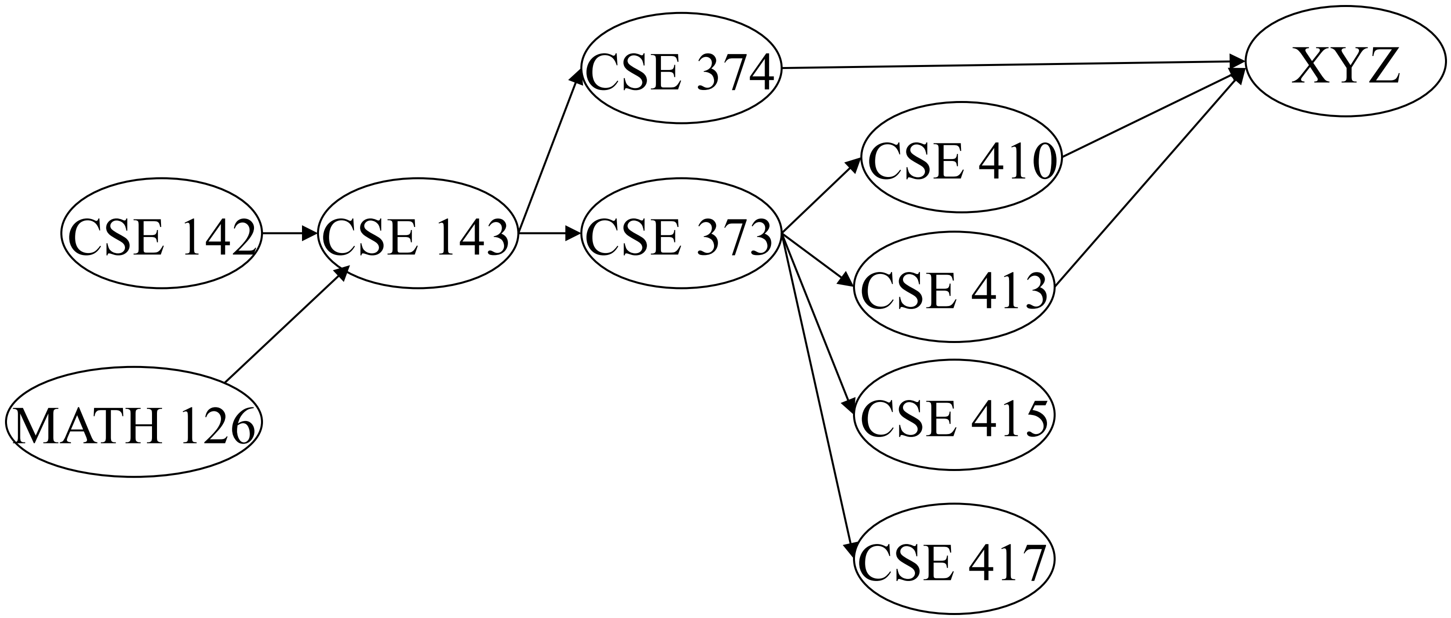
Example



Output:
126
142
143
374
373
417
410
413

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x		x	
In-degree:	0	0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~3~~ ~~2~~ ~~1~~ 0

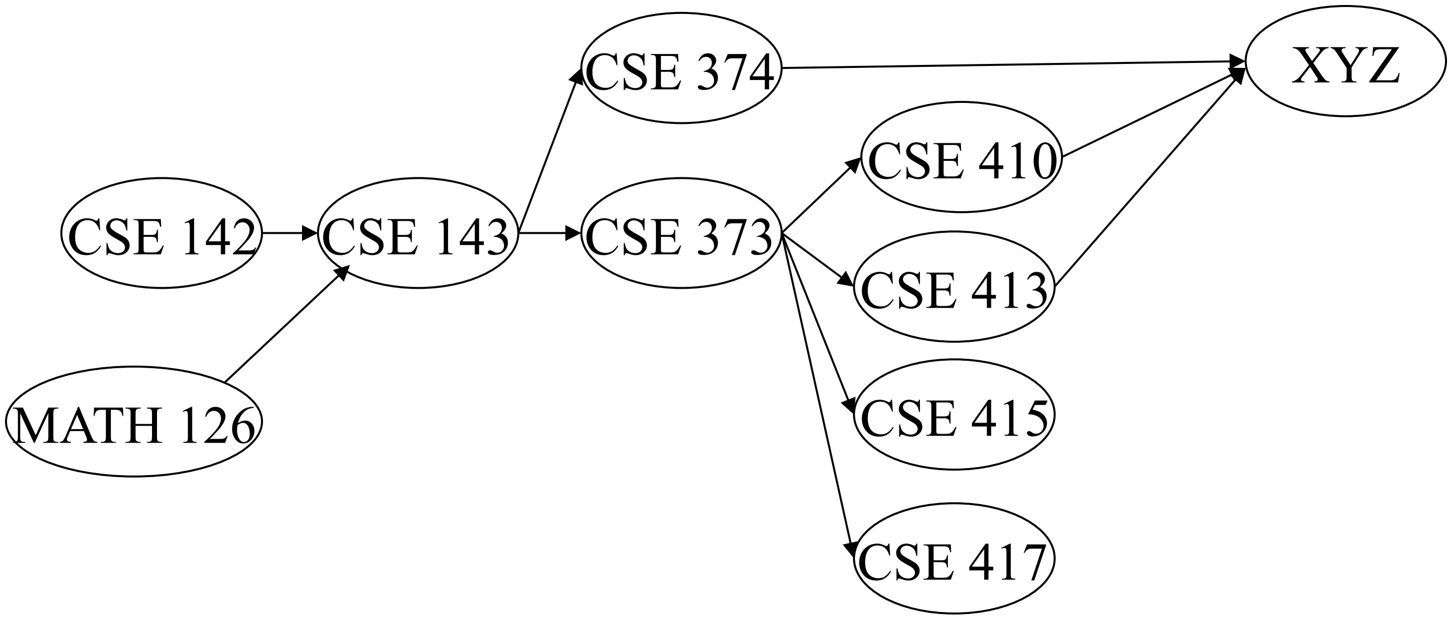
Example



Output:
126
142
143
374
373
417
410
413
XYZ

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x		x	x
In-degree:	0	0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~3~~ ~~2~~ ~~1~~ 0

Example



Output:
126
142
143
374
373
417
410
413
XYZ
415

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x	x	x	x
In-degree:	0	0	~~2~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~3~~
			~~1~~	0	0	0	0	0	0	~~2~~
			0							~~1~~
										0

NOTICE

Needed a vertex with in-degree 0 to start

- Will always have at least 1 because no cycles

Ties among vertices with in-degrees of 0 can be broken arbitrarily

- Can be more than one correct answer, by definition, depending on the graph

IMPLEMENTATION

The trick is to avoid searching for a zero-degree node every time!

- Keep the "pending" zero-degree nodes in a list, stack, queue, bag, 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

SINGLE SOURCE SHORTEST PATHS

Done: BFS to find the minimum path length from v to u in

$O(|E|+|V|)$

Actually, can find the minimum path length from v to *every node*

- Still $O(|E|+|V|)$
- No faster way for a “distinguished” destination in the worst-case

Now: Weighted graphs

Given a weighted graph and node v,

find the minimum-cost path from v to every node

As before, asymptotically no harder than for one destination

Unlike before, BFS will not work -> only looks at path length.

SHORTEST PATH: APPLICATIONS

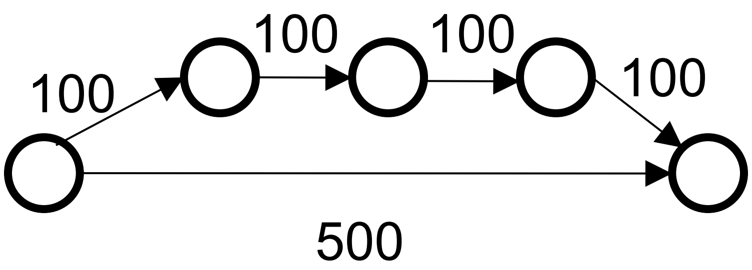
Driving directions

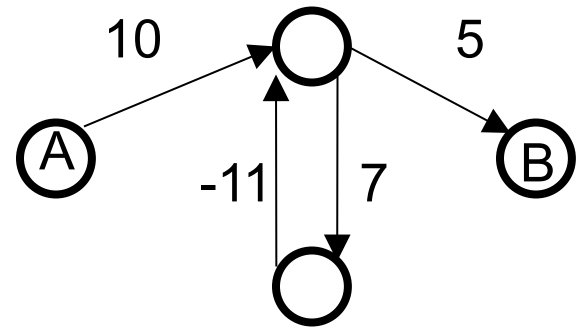
Cheap flight itineraries

Network routing

Critical paths in project management

NOT AS EASY





Why BFS won't work: Shortest path may not have the fewest edges

- Annoying when this happens with costs of flights

We will assume there are no negative weights

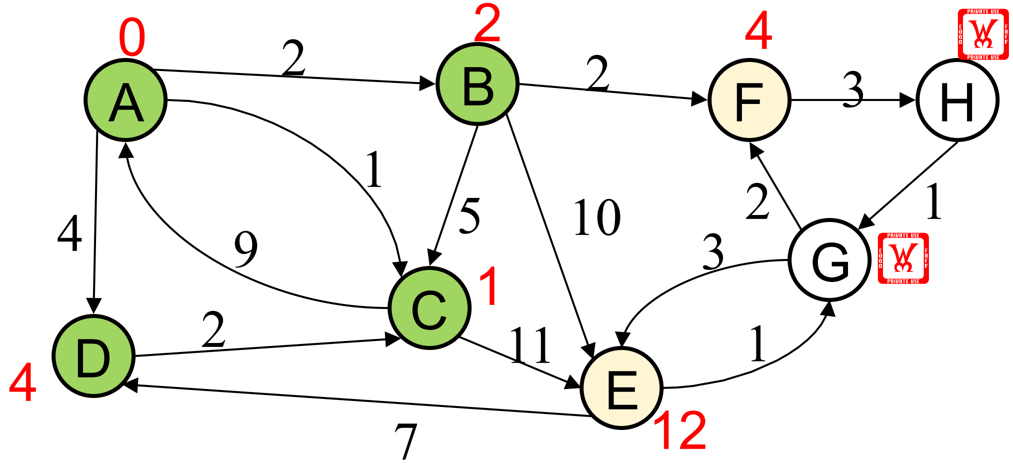
- *Problem* is *ill-defined* if there are negative-cost *cycles*
- *Today's algorithm* is *wrong* if *edges* can be negative
  - There are other, slower (but not terrible) algorithms

DIJKSTRA'S ALGORITHM

The idea: reminiscent of BFS, but adapted to handle weights

- Grow the set of nodes whose shortest distance has been computed
- Nodes not in the set will have a "best distance so far"
- A priority queue will turn out to be useful for efficiency

DIJKSTRA'S ALGORITHM



Initially, start node has cost 0 and all other nodes have cost ∞

At each step:

- Pick closest unknown vertex **v**
- Add it to the "cloud" of known vertices
- Update distances for nodes with edges from **v**

That's it! (But we need to prove it produces correct answers)

THE ALGORITHM

1. **For each node `v`, set `v.cost = ∞` and `v.known = false`**
2. **Set `source.cost = 0`**
3. **While there are unknown nodes in the graph**
   a) Select the unknown node **`v`** with lowest cost
   b) Mark **`v`** as known
   c) For each edge **`(v,u)`** with weight **`w`**,

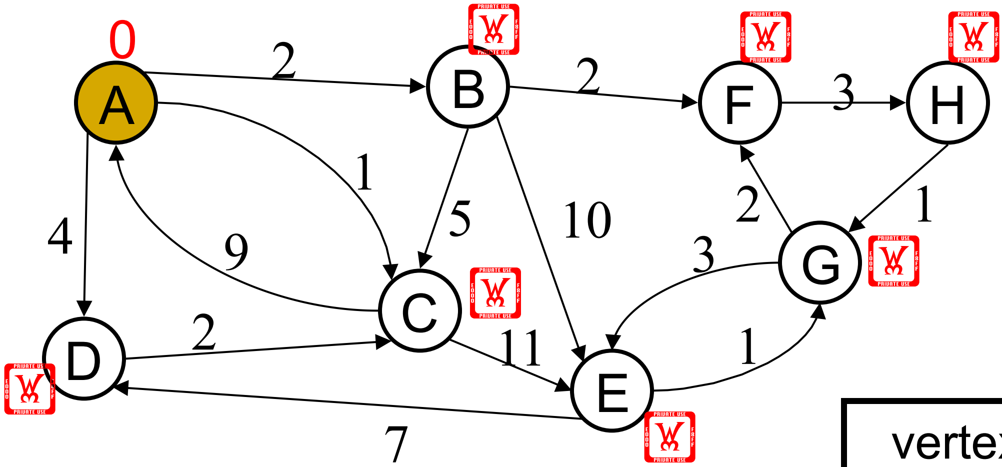
```
c1 = v.cost + w // cost of best path through v to u
c2 = u.cost  // cost of best path to u previously known
if(c1 < c2){ // if the path through v is better
   u.cost = c1
   u.path = v // for computing actual paths
}
```

IMPORTANT FEATURES

When a vertex is marked known, the cost of the shortest path to that node is known

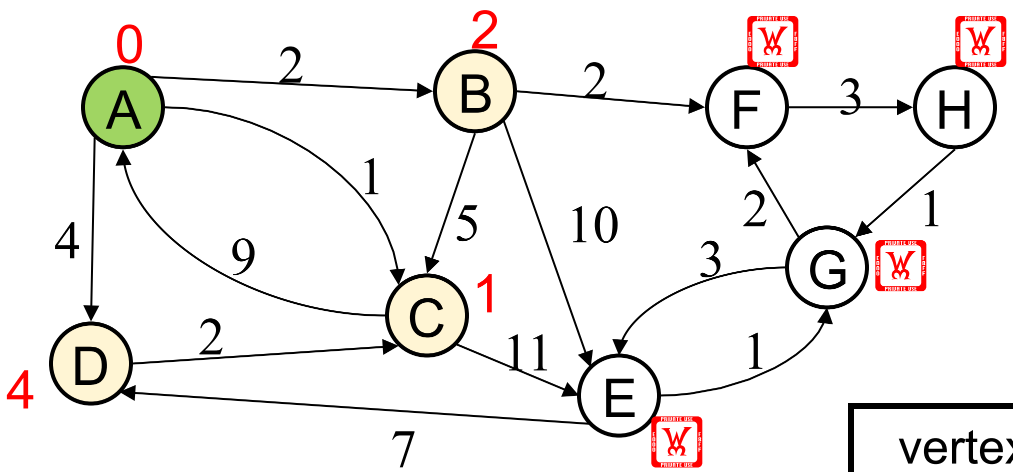
- The path is also known by following back-pointers

While a vertex is still not known, another shorter path to it *might* still be found



vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	
H		??	

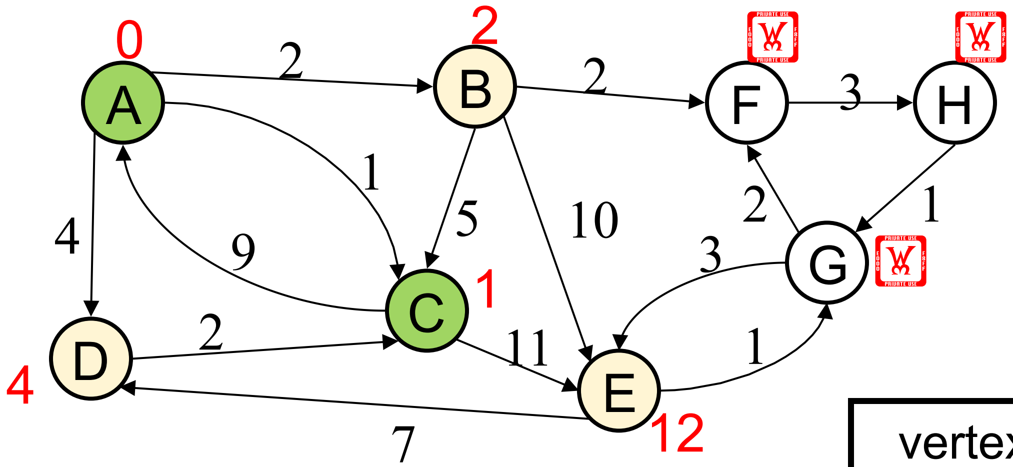
Order Added to Known Set:



vertex	known?	cost	path
A	Y	0	
B		≤ 2	A
C		≤ 1	A
D		≤ 4	A
E		??	
F		??	
G		??	
H		??	

<u>Order Added to Known Set:</u>

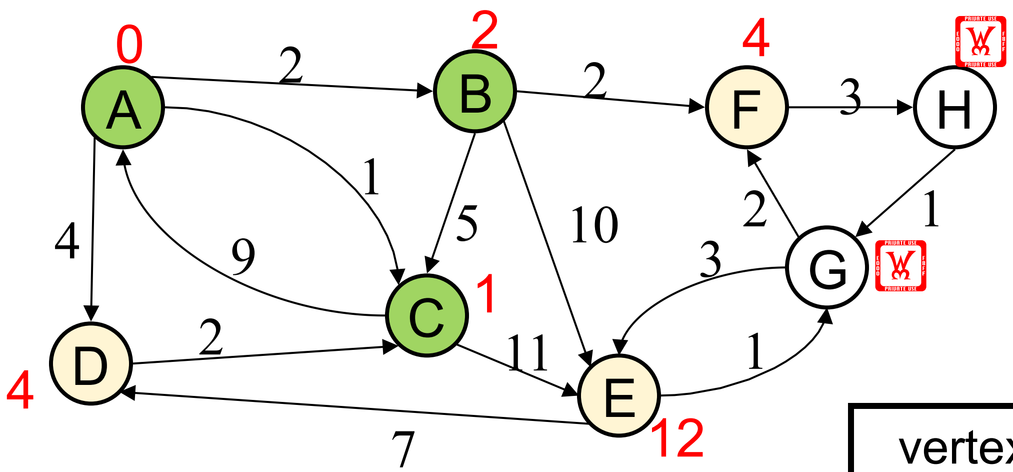
A



vertex	known?	cost	path
A	Y	0	
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		??	
G		??	
H		??	

<u>Order Added to Known Set:</u>

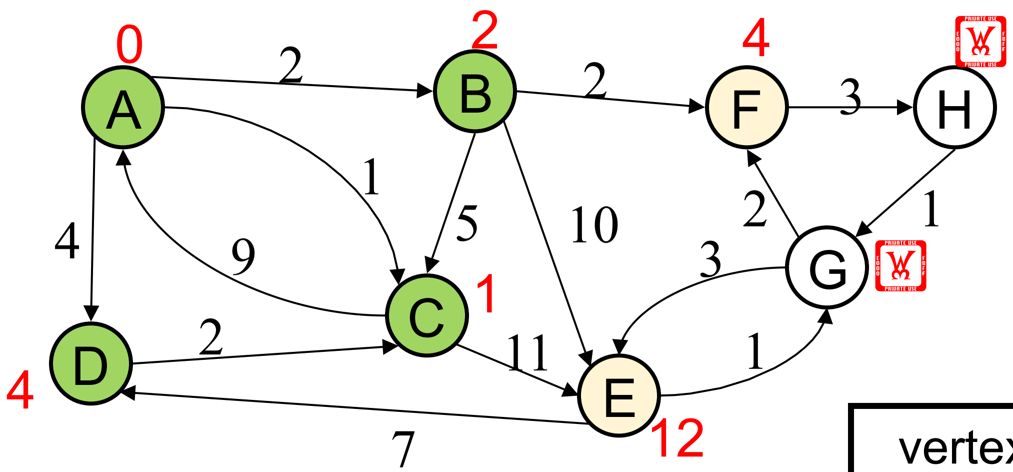
A, C



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		≤ 4	B
G		??	
H		??	

<u>Order Added to Known Set:</u>

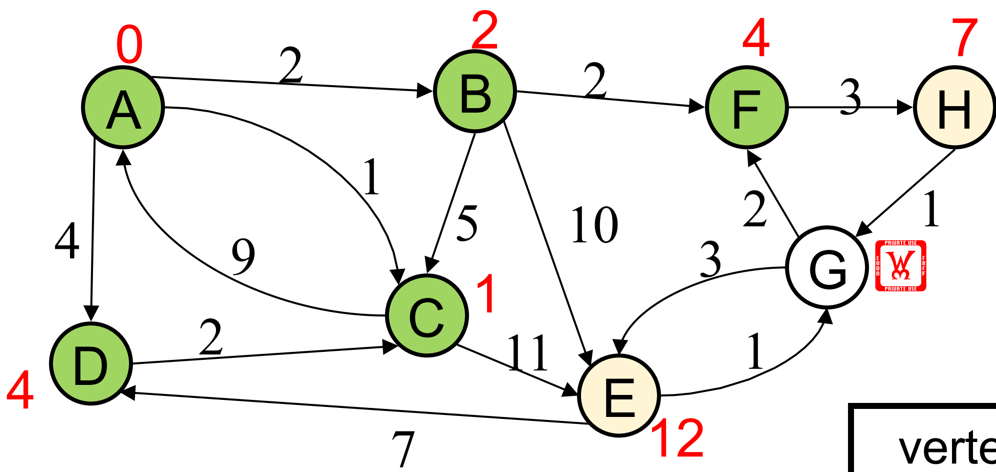
A, C, B



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		??	
H		??	

<u>Order Added to Known Set:</u>

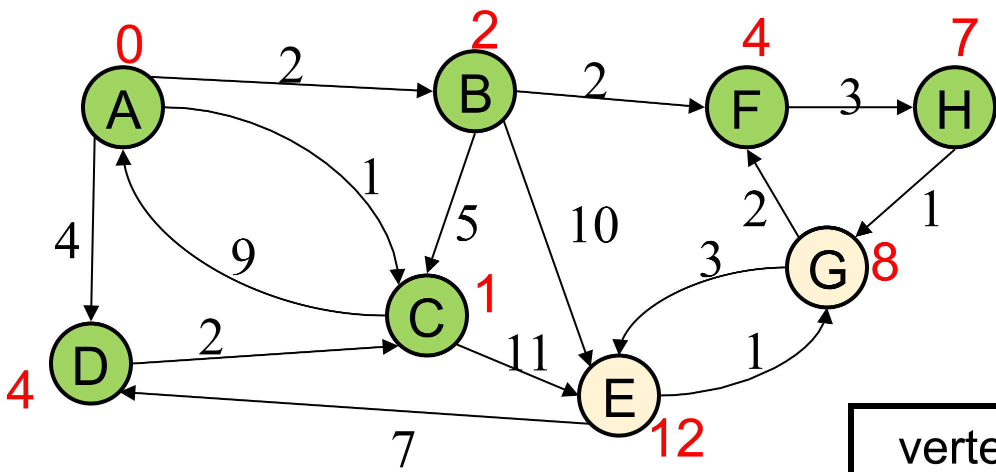
A, C, B, D



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		??	
H		≤ 7	F

<u>Order Added to Known Set:</u>

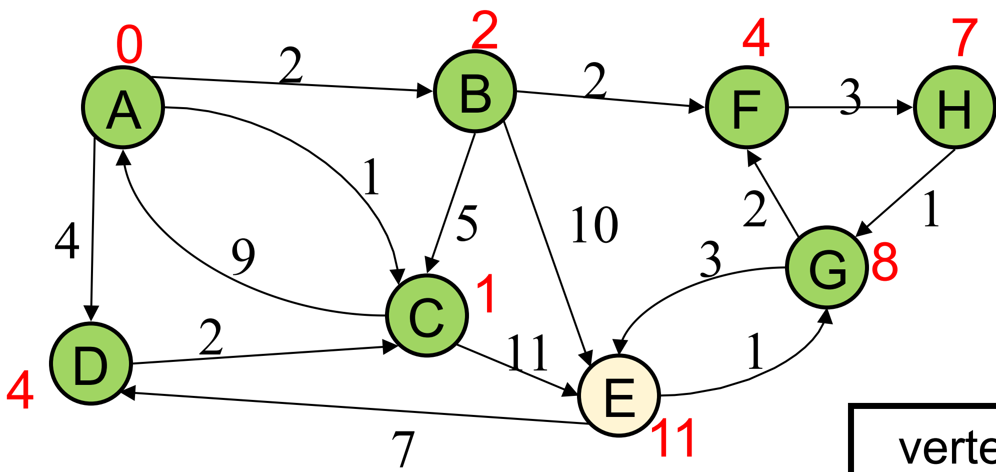
A, C, B, D, F



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		≤ 8	H
H	Y	7	F

Order Added to Known Set:

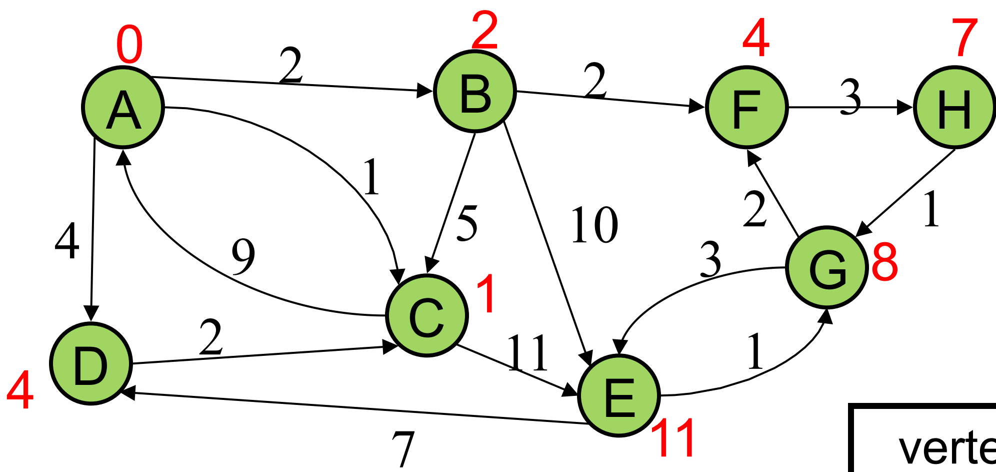
A, C, B, D, F, H



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G, E

FEATURES

When a vertex is marked known,
the cost of the shortest path to that node is known

- The path is also known by following back-pointers

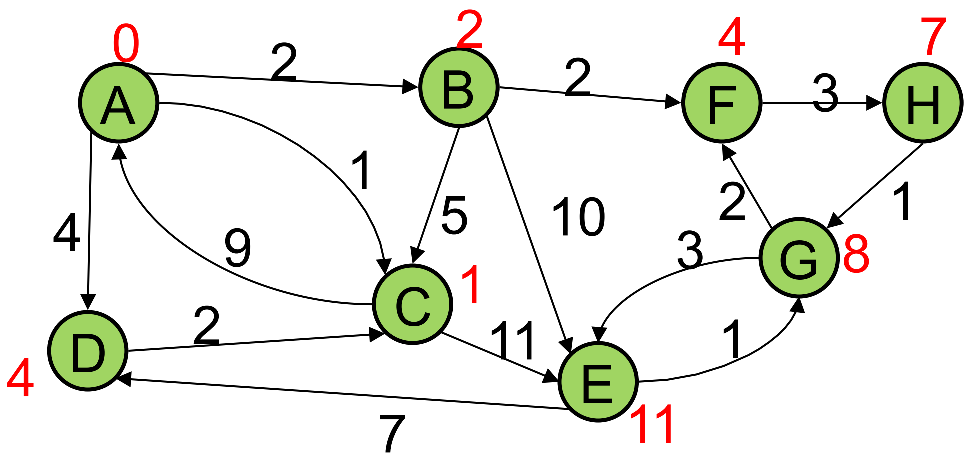
While a vertex is still not known,
another shorter path to it might still be found

Note: The "Order Added to Known Set" is not important

- A detail about how the algorithm works (client doesn't care)
- Not used by the algorithm (implementation doesn't care)
- It is sorted by path-cost, resolving ties in some way
    - Helps give intuition of why the algorithm works

INTERPRETING THE RESULTS

Now that we're done, how do we get the path from, say, A to E?



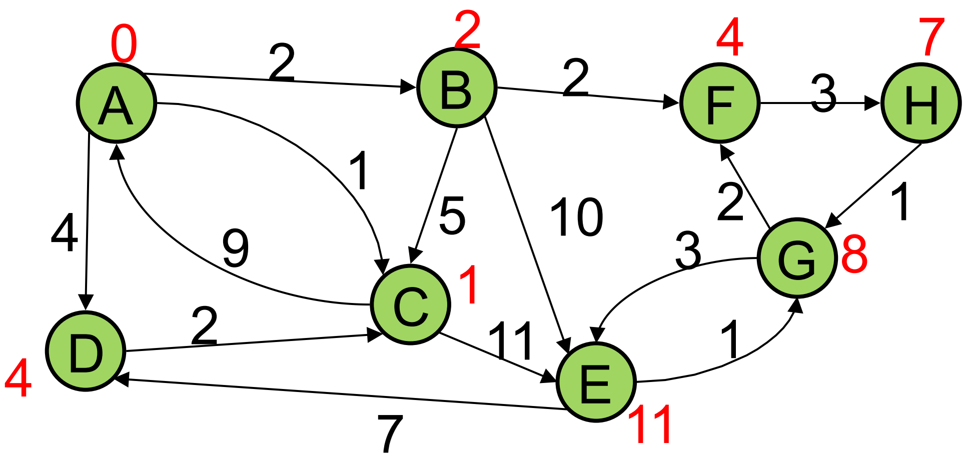
vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G, E

How would this have worked differently if we were only interested in:

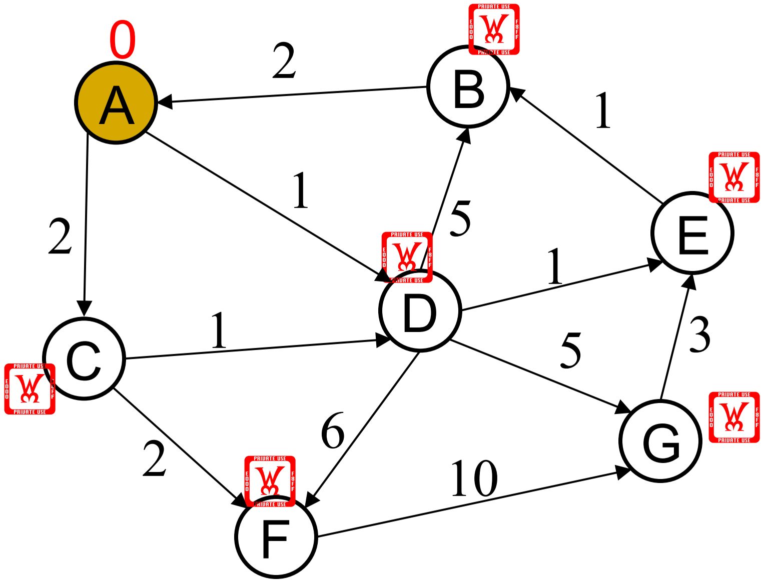
- The path from A to G?
- The path from A to E?



<u>Order Added to Known Set:</u>

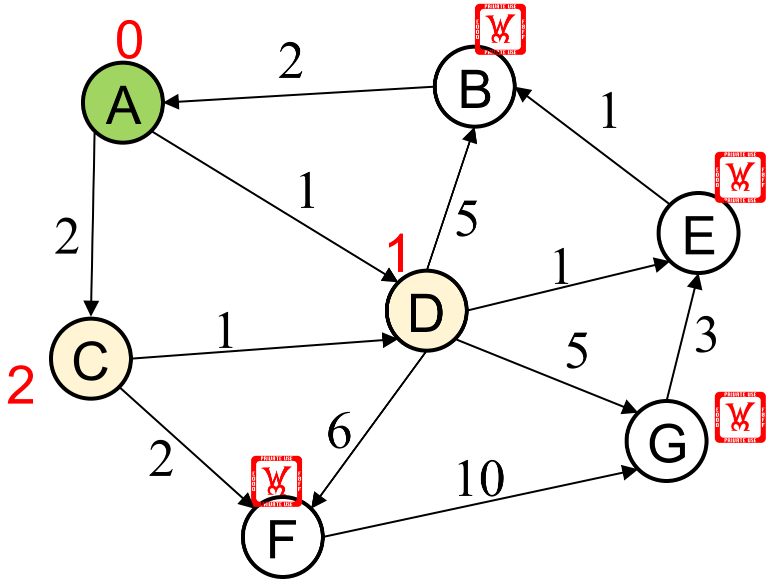
A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F



Order Added to Known Set:

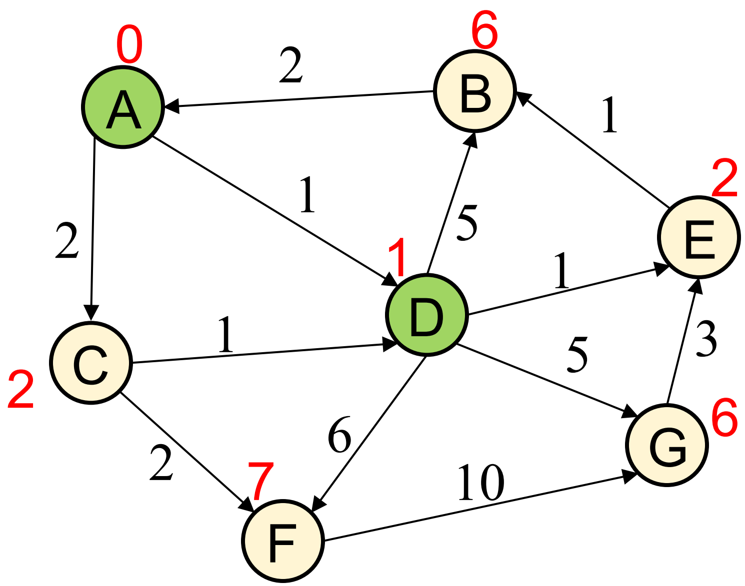
vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	



vertex	known?	cost	path
A	Y	0	
B		??	
C		≤ 2	A
D		≤ 1	A
E		??	
F		??	
G		??	

Order Added to Known Set:

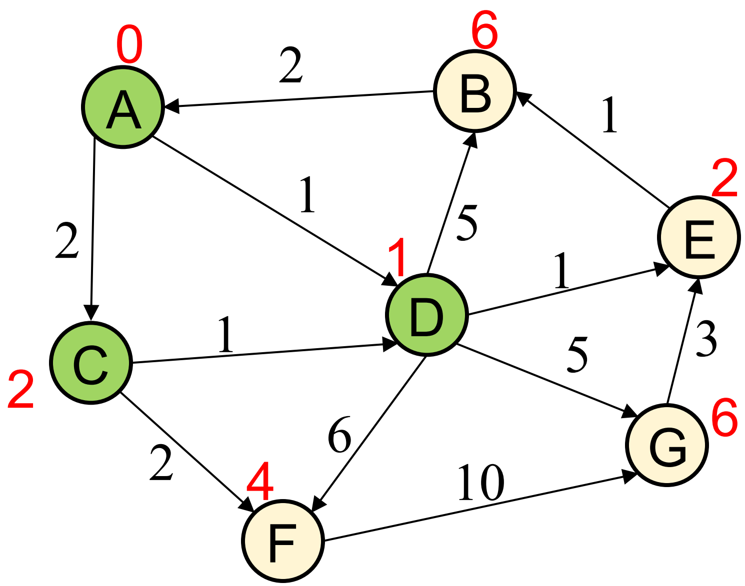
A



vertex	known?	cost	path
A	Y	0	
B		≤ 6	D
C		≤ 2	A
D	Y	1	A
E		≤ 2	D
F		≤ 7	D
G		≤ 6	D

<u>Order Added to Known Set:</u>

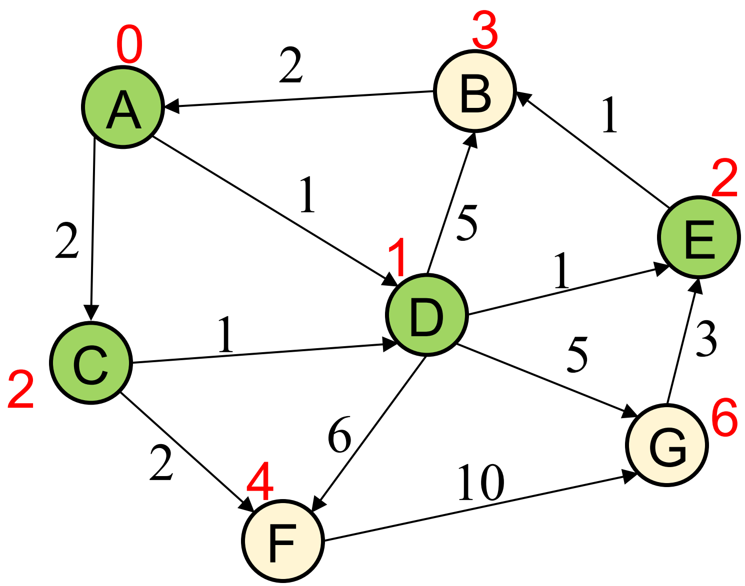
A, D



Order Added to Known Set:

A, D, C

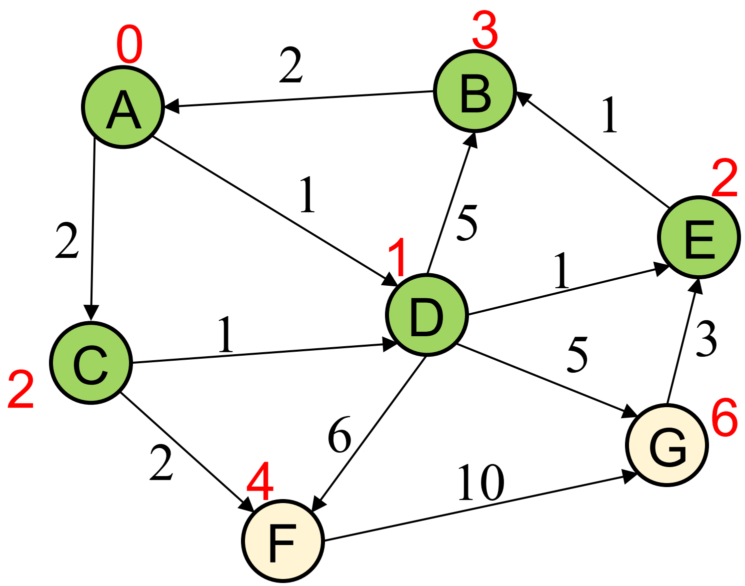
vertex	known?	cost	path
A	Y	0	
B		≤ 6	D
C	Y	2	A
D	Y	1	A
E		≤ 2	D
F		≤ 4	C
G		≤ 6	D



vertex	known?	cost	path
A	Y	0	
B		≤ 3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

Order Added to Known Set:

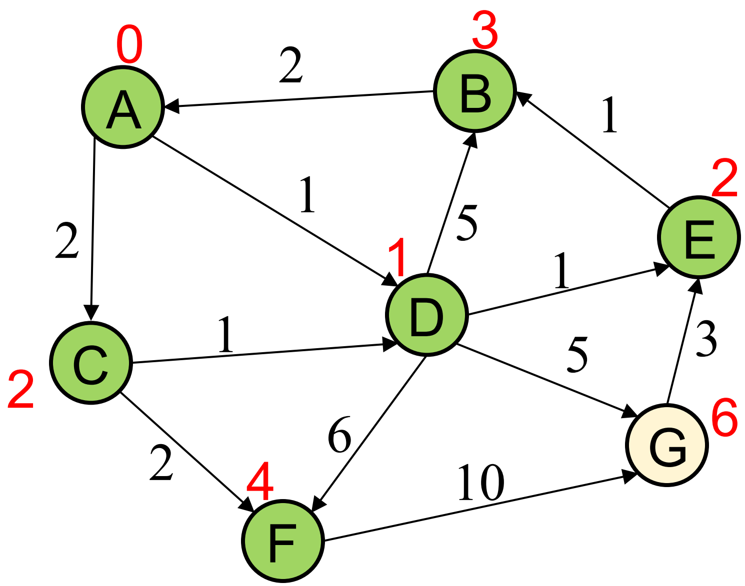
A, D, C, E



Order Added to Known Set:

A, D, C, E, B

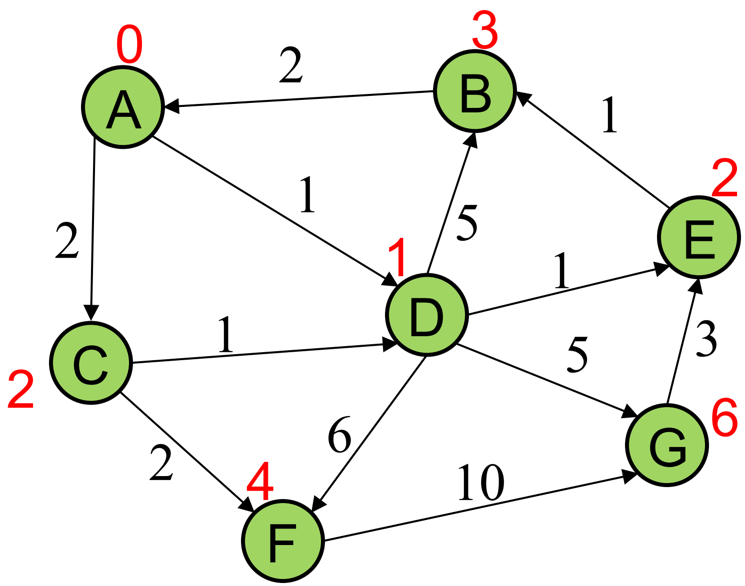
vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D



Order Added to Known Set:

A, D, C, E, B, F

vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G		≤ 6	D



Order Added to Known Set:

A, D, C, E, B, F, G

vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

NEXT WEEK

- Another topological sort problem
- Weights and pathfinding
- Start Dijkstra’s algorithm