CSE 373: Data Structures and Algorithms

Topological Sort

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

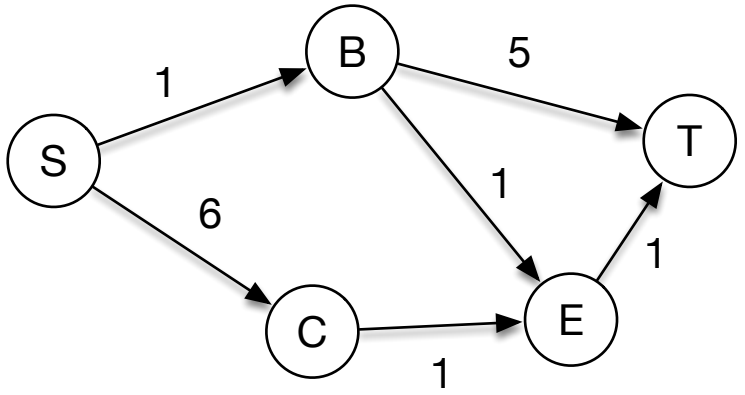
Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

Dijkstra's algorithm

Review

```
1:  function Dijkstra(Graph G, Vertex source)                ▷ with MPQ
2:      initialize distances to ∞, source.dist = 0
3:      mark all vertices unprocessed
4:      initialize MPQ as a min priority queue; add source with priority 0
5:      while MPQ is not empty do
6:          u = MPQ.getMin()
7:          for each edge (u,v) leaving u do
8:              if u.dist + w(u,v) < v.dist then
9:                  if v.dist == ∞ then
10:                     MPQ.insert(v, u.dist + w(u, v))
11:                 else
12:                     MPQ.decreasePriority(v, u.dist + w(u,v))
13:                 end if
14:                 v.dist = u.dist + w(u,v)
15:                 v.predecessor = u
16:             end if
17:         end for
18:         mark u as processed
19:     end while
20: end function
```

Vertex	Distance	Predecessor	Processed

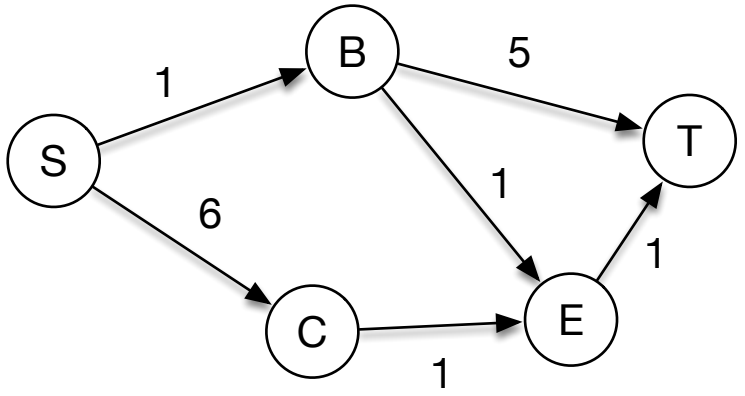


Dijkstra's algorithm

Review

```
1:  function Dijkstra(Graph G, Vertex source)                ▷ with MPQ
2:      initialize distances to ∞, source.dist = 0
3:      mark all vertices unprocessed
4:      initialize MPQ as a min priority queue; add source with priority 0
5:      while MPQ is not empty do
6:          u = MPQ.getMin()
7:          for each edge (u,v) leaving u do
8:              if u.dist + w(u,v) < v.dist then
9:                  if v.dist == ∞ then
10:                     MPQ.insert(v, u.dist + w(u, v))
11:                 else
12:                     MPQ.decreasePriority(v, u.dist + w(u,v))
13:                 end if
14:                 v.dist = u.dist + w(u,v)
15:                 v.predecessor = u
16:             end if
17:         end for
18:         mark u as processed
19:     end while
20: end function
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
B	1	S	Yes
C	6	S	Yes
E	2	B	Yes
T	3	E	Yes



Running time analysis

```
1:  function Dijkstra(Graph G, Vertex source)                    ▷ with MPQ
2:      initialize distances to ∞, source.dist = 0
3:      mark all vertices unprocessed
4:      initialize MPQ as a min priority queue; add source with priority 0
5:      while MPQ is not empty do
6:          u = MPQ.getMin()
7:          for each edge (u,v) leaving u do
8:              if u.dist + w(u,v) < v.dist then
9:                  if v.dist == ∞ then
10:                     MPQ.insert(v, u.dist + w(u, v))
11:                 else
12:                     MPQ.decreasePriority(v, u.dist + w(u,v))
13:                 end if
14:                 v.dist = u.dist + w(u,v)
15:                 v.predecessor = u
16:             end if
17:         end for
18:         mark u as processed
19:     end while
20: end function
```

History of shortest path algorithms

Algorithm/Author	Time complexity	Year
Ford	$O(\|V\|^2\|E\|)$	1956
Bellman-Ford. Shimbel	$O(\|V\|\|E\|)$	1958
Dijkstra's algorithm with binary heap	$\mathbf{O(\|E\|\log\|V\| + \|V\|\log\|V\|)}$	**1959**
Dijkstra's algorithm with Fibonacchi heap	$O(\|E\| + \|V\|\log\|V\|)$	1984
Gabow's algorithm	$O(\|E\| + \|V\|\sqrt{\log\|V\|})$	1990
Thorup	$O(\|E\| + \|V\|\log\log\|V\|)$	2004

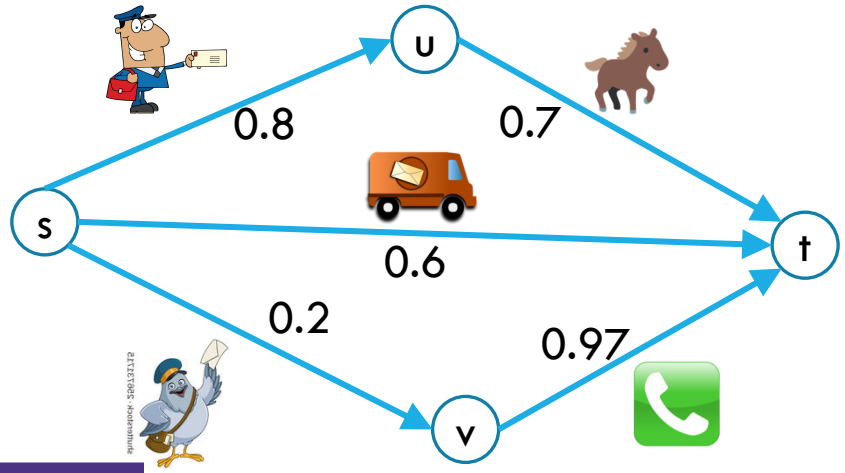
History of shortest path algorithms. In this class, from this table, you are expected to know only Dijkstra's algorithm with binary heap and its time complexity. You are not expected to know the other algorithms or their time complexities.

Other applications of shortest paths

Shortest path algorithms are obviously useful for GoogleMaps.

The wonderful thing about graphs is they can encode **arbitrary** relationships among objects.

I have a message I need to get from point s to point t.
But the connections are unreliable.
What path should I send the message along so it has the best chance of arriving?



Maximum Probability Path Problem

Given: a directed graph G, where each edge weight is the probability of successfully transmitting a message across that edge

Find: the path from s to t with maximum probability of message transmission

Other applications of shortest paths

Robot navigation

Urban traffic planning

Tramp steamer problem

Optimal pipelining of VLSI chips

Operator scheduling

Subroutine in higher level algorithms

Exploiting arbitrage opportunities in currency exchanges

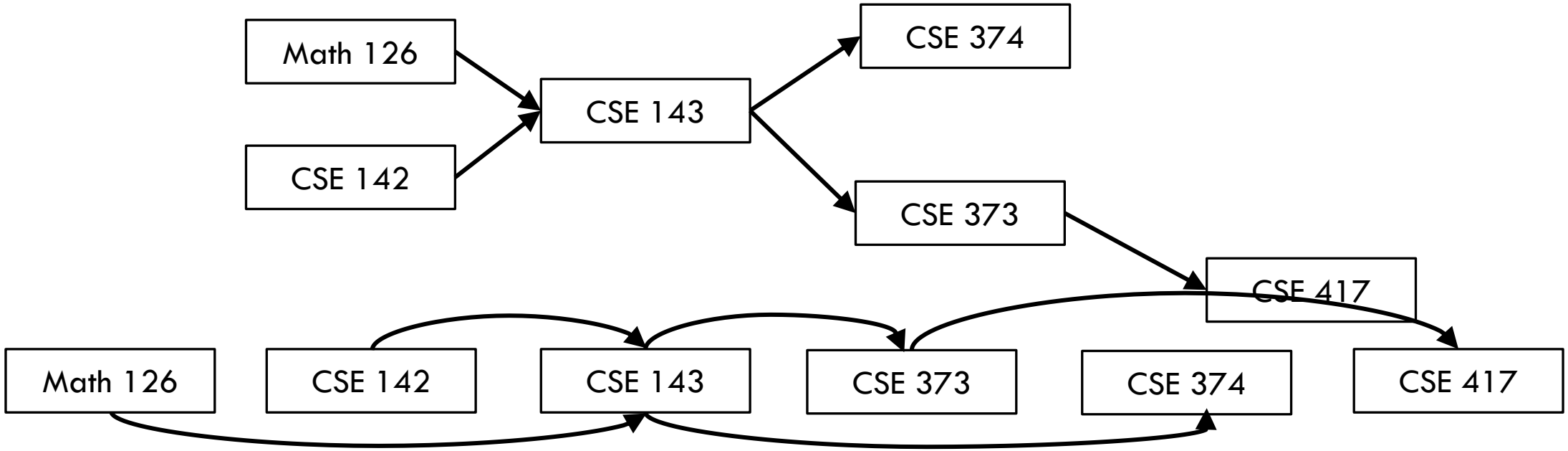
Open shortest path first routing protocol for IP

Optimal truck routing through given traffic congestion

Topological Sort

Problem 1: Ordering Dependencies

Given a bunch of courses with prerequisites, find an order to take the courses in.



Problem 1: Ordering Dependencies

Given a directed graph G, where we have an edge from u to v if u must happen before v.

We can only do things one at a time, can we find an order that **respects dependencies**?

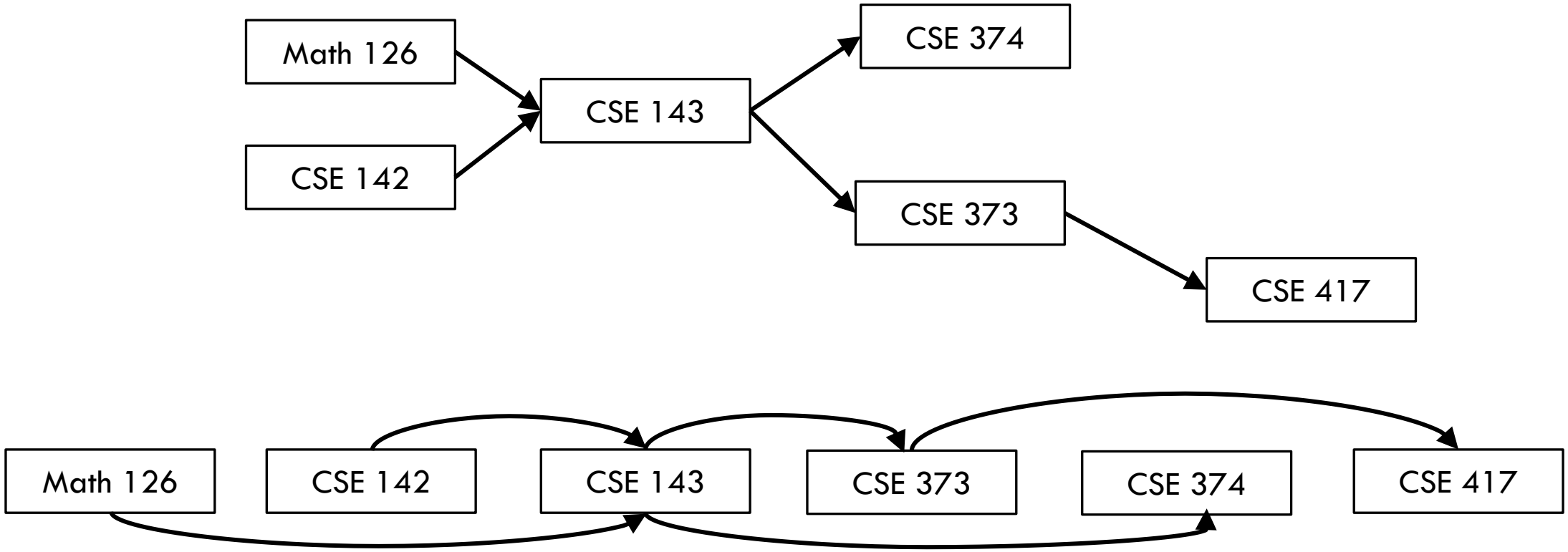
Topological Sort (aka Topological Ordering)

Given: a directed graph G
Find: an ordering of the vertices so all edges go from left to right.

Uses:
Compiling multiple files
Graduating
Manufacturing workflows (assembly lines)

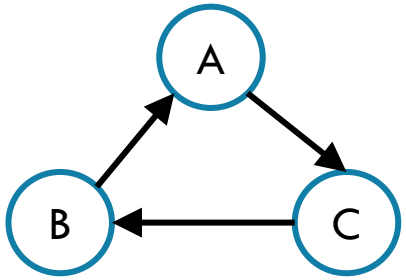
Topological Ordering

A course prerequisite chart and a possible topological ordering.



Can we always order a graph?

Can you topologically order this graph?



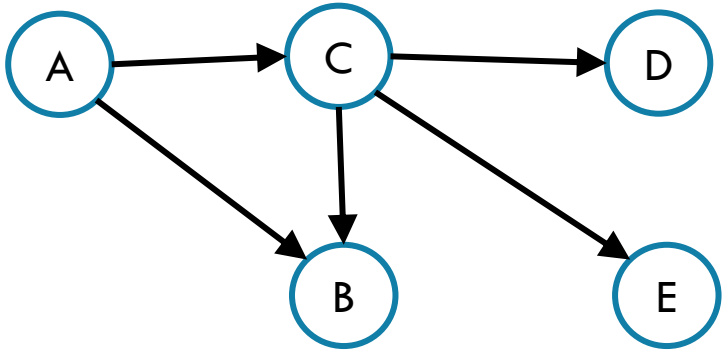
Directed Acyclic Graph (DAG)

A directed graph without any cycles.

A graph has a topological ordering if and only if it is a DAG.

Ordering a DAG

Does this graph have a topological ordering? If so find one.



If a vertex doesn't have any edges going into it, we can add it to the ordering.
More generally, if the only incoming edges are from vertices already in the ordering, it's safe to add.

How Do We Find a Topological Ordering?

```
1:  function TopologicalSort(Graph G, Vertex source)
2:      count how many incoming edges each vertex has
3:      Collection toProcess = new Collection()
4:      for each Vertex v in G do
5:          if v.edgesRemaining == 0 then
6:              toProcess.insert(v)
7:          end if
8:      end for
9:      topOrder = new List()
10:     while toProcess is not empty do
11:         u = toProcess.remove()
12:         topOrder.insert(u)
13:         for each edge (u,v) leaving u do
14:             v.edgesRemaining = v.edgesRemaining - 1
15:             if v.edgesRemaining == 0 then
16:                 toProcess.insert(v)
17:             end if
18:         end for
19:     end while
20: end function
```

What's the running time?

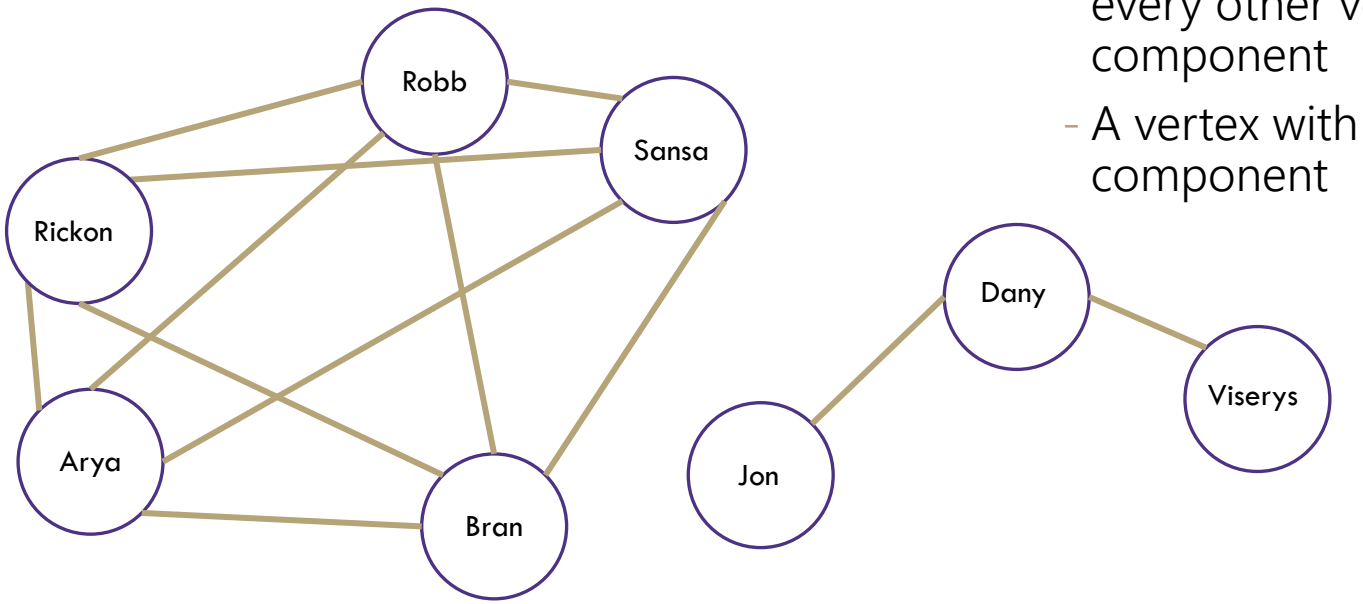
```
1:  function TopologicalSort(Graph G, Vertex source)
2:      count how many incoming edges each vertex has
3:      Collection toProcess = new Collection()
4:      for each Vertex v in G do
5:          if v.edgesRemaining == 0 then
6:              toProcess.insert(v)
7:          end if
8:      end for
9:      topOrder = new List()
10:     while toProcess is not empty do
11:         u = toProcess.remove()
12:         topOrder.insert(u)
13:         for each edge (u,v) leaving u do
14:             v.edgesRemaining = v.edgesRemaining - 1
15:             if v.edgesRemaining == 0 then
16:                 toProcess.insert(v)
17:             end if
18:         end for
19:     end while
20: end function
```

Strongly Connected Components

Connected [Undirected] Graphs

Connected graph – a graph where every vertex is connected to every other vertex via some path. It is not required for every vertex to have an edge to every other vertex

There exists some way to get from each vertex to every other vertex



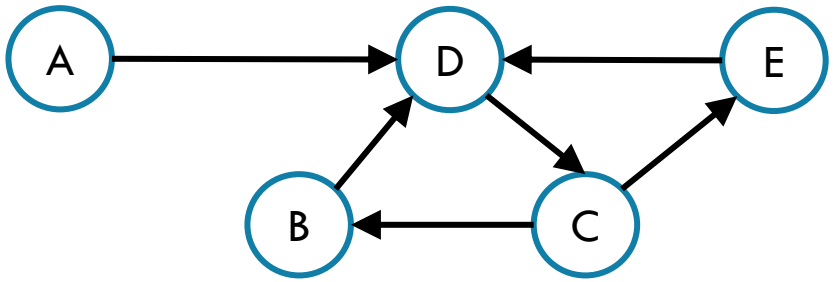
Connected Component – a *subgraph* in which any two vertices are connected via some path, but is connected to no additional vertices in the *supergraph*

- There exists some way to get from each vertex within the connected component to every other vertex in the connected component
- A vertex with no edges is itself a connected component

Strongly Connected Components

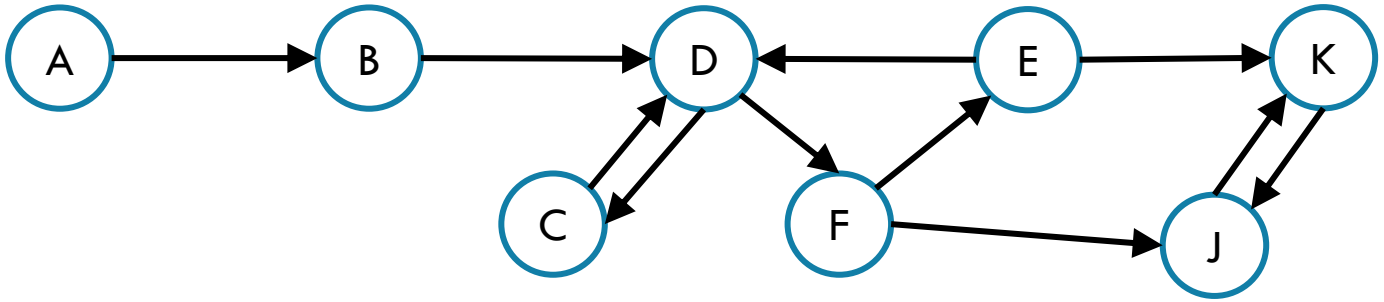
Strongly Connected Component

A subgraph C such that every pair of vertices in C is connected via some path **in both directions**, and there is no other vertex which is connected to every vertex of C in both directions.



Note: the direction of the edges matters!

Strongly Connected Components Problem



{A}, {B}, {C,D,E,F}, {J,K}

Strongly Connected Components Problem

Given: A directed graph G

Find: The strongly connected components of G

SCC Algorithm

Ok. How do we make a computer do this?

You could:

- run a [B/D]FS from every vertex,
- For each vertex record what other vertices it can get to
- and figure it out from there.

But you can do better. There's actually an O(|V|+|E|) algorithm!

I only want you to remember two things about the algorithm:

- It is an application of depth first search.
- It runs in linear time

The problem with running a [B/D]FS from every vertex is you recompute a lot of information.

The time you are popped off the stack in DFS contains a "smart" ordering to do a second DFS where you don't need to recompute that information.

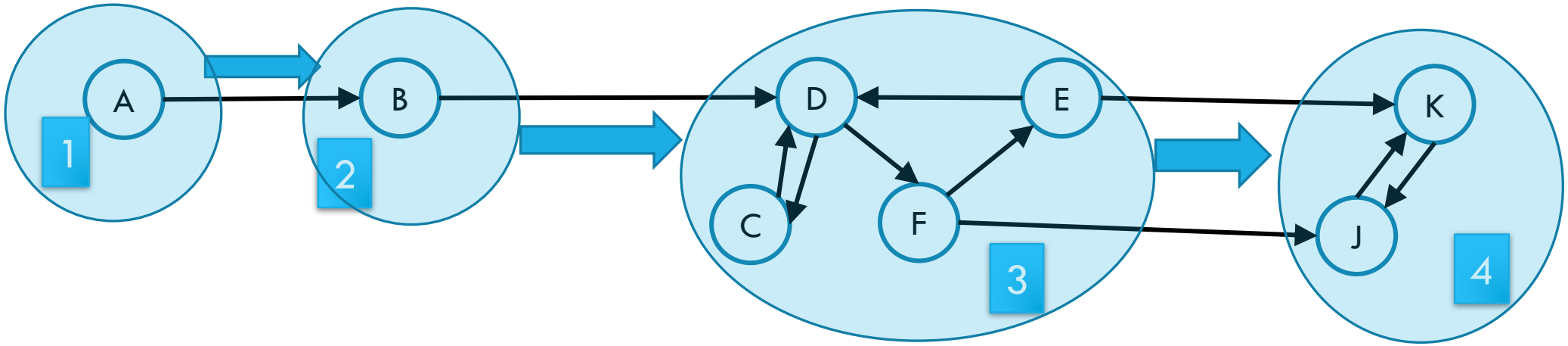
Why Find SCCs?

Graphs are useful because they encode relationships between arbitrary objects.

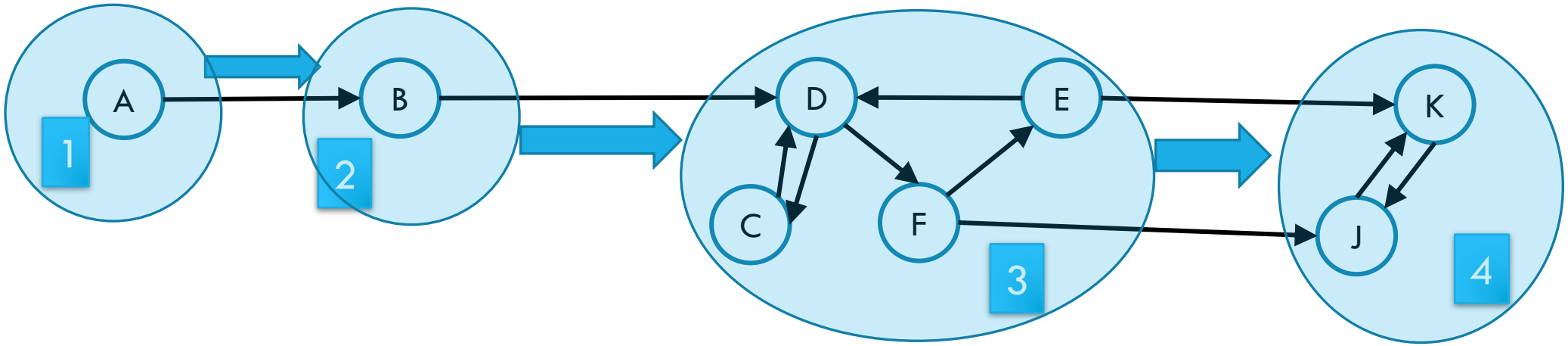
We've found the strongly connected components of G.

Let's build a new graph out of them! Call it H

- Have a vertex for each of the strongly connected components
- Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2.



Why Find SCCs?



That's awful meta. Why?

This new graph summarizes reachability information of the original graph.

- I can get from A (of G) in 1 to F (of G) in 3 if and only if I can get from 1 to 3 in H.

Why Must H Be a DAG?

H is always a DAG (do you see why?).

Takeaways

Finding SCCs lets you **collapse** your graph to the meta-structure.
If (and only if) your graph is a DAG, you can find a topological sort of your graph.

Both of these algorithms run in linear time.

Just about everything you could want to do with your graph will take at least as long.

You should think of these as **"almost free" preprocessing** of your graph.

- Your other graph algorithms only need to work on
  - topologically sorted graphs and
  - strongly connected graphs.

Appendix: Strongly Connected Components Algorithm

Efficient SCC

We'd like to find all the vertices in our strongly connected component in time corresponding to the size of the component, not for the whole graph.

We can do that with a DFS (or BFS) as long as we don't leave our connected component.

If we're a "sink" component, that's guaranteed. I.e. a component whose vertex in the meta-graph has no outgoing edges.

How do we find a sink component? We don't have a meta-graph yet (we need to find the components first)

DFS can find a vertex in a source component, i.e. a component whose vertex in the meta-graph has no incoming edges.

- That vertex is the last one to be popped off the stack.

So if we run DFS in the *reversed* graph (where each edge points the opposite direction) we can find a sink component.

Efficient SCC

So from a DFS in the reversed graph, we can use the order vertices are popped off the stack to find a sink component (in the original graph).

Run a DFS from that vertex to find the vertices in that component *in size of that component time.*

Now we can delete the edges coming into that component.

The last remaining vertex popped off the stack is a sink of the remaining graph, and now a DFS from them won't leave the component.

Iterate this process (grab a sink, start DFS, delete edges entering the component).

In total we've run two DFSs. (since we never leave our component in the second DFS).

More information, and pseudocode:

https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

http://jeffe.cs.illinois.edu/teaching/algorithms/notes/19-dfs.pdf (mathier)