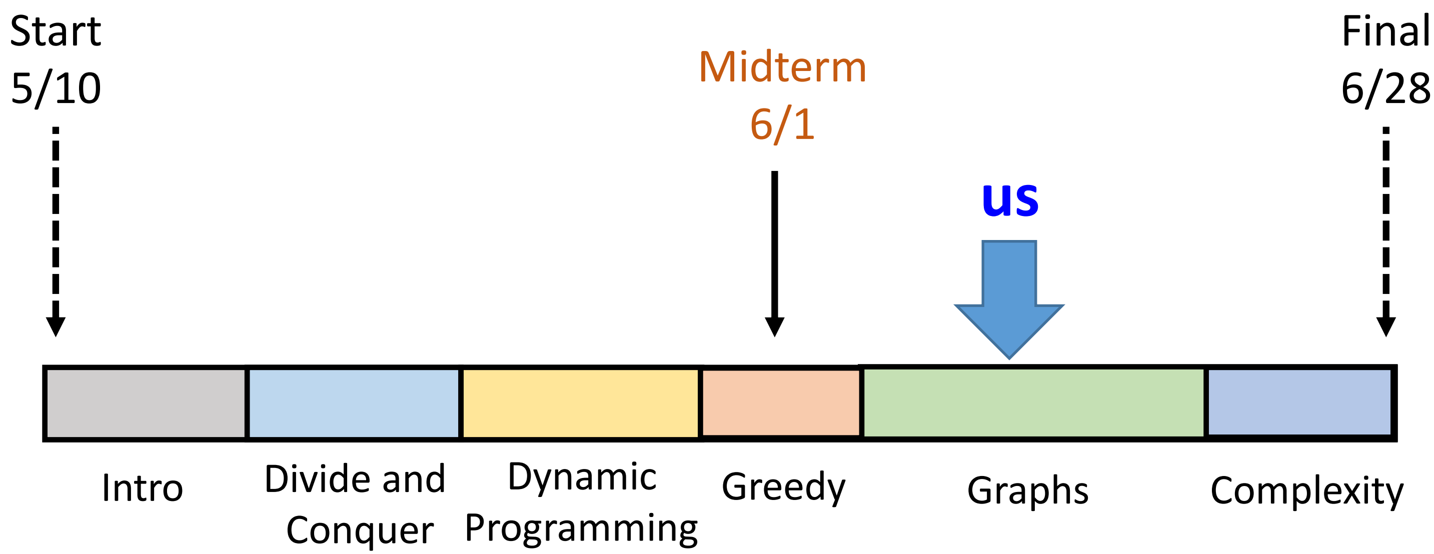
HW4 - due 6/12
Kristen OH change 6/10

CS3000: Algorithms & Data
Drew van der Poel

Lecture 17
- Strongly Connected Components
- Dijkstra's

June 9, 2021

Outline



Last class: Graphs: Topological Orderings, Connected Components

Next class: Graphs: Dijkstra's

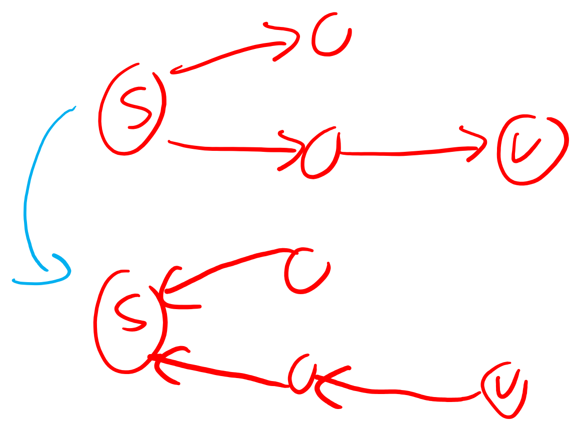
Strongly Connected Components

- **Observation:** $\text{SCC}(s)$ is all nodes $v \in V$ such that v is reachable from s and vice versa
    - Can find all nodes reachable from s using DFS
    - How do we find all nodes that *can reach* s?
        - DFS(s) in reverse of the graph!

DFS in G^R:

all nodes that can reach s in G

G:

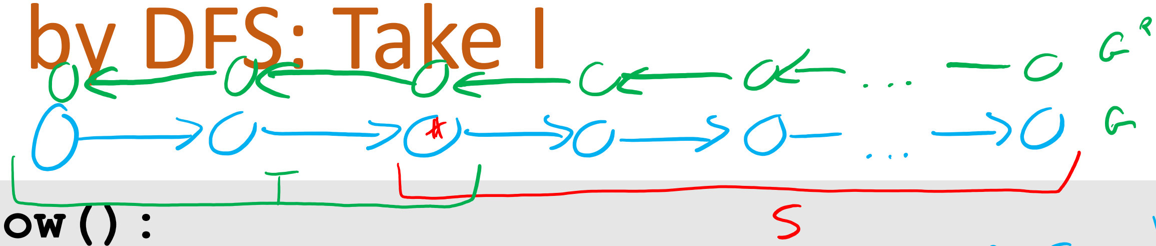


path from s to v

↓

path from v to s

SCCs by DFS: Take I

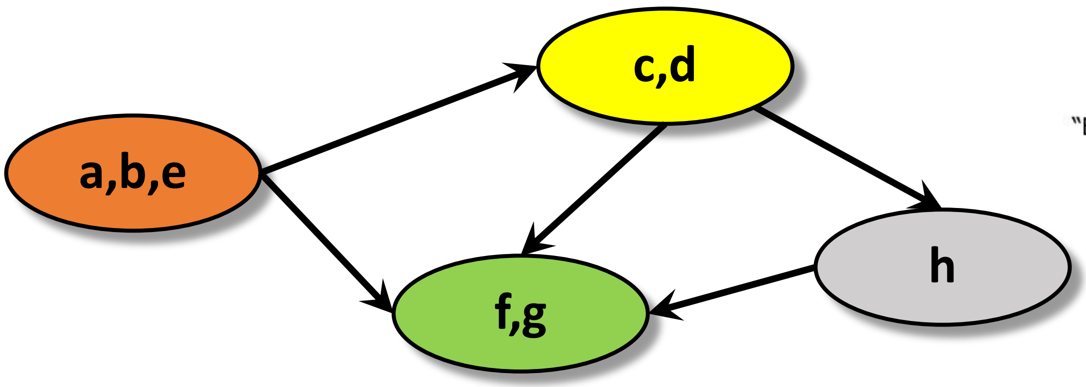


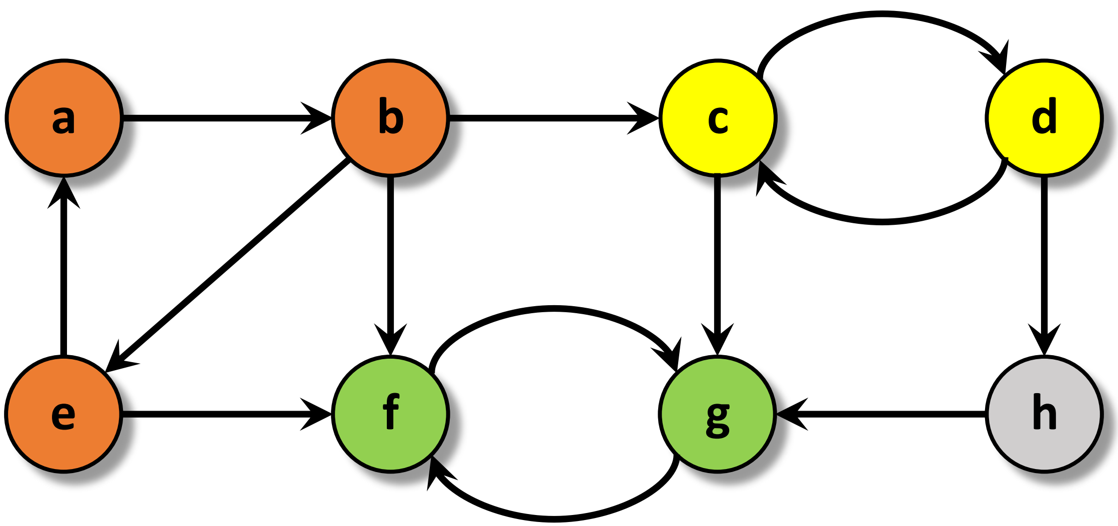

```
SCC-Slow():
  G^R = G with all edges "reversed"                 O(m)

  // Initialize an array and counter
  comp[1:n] = ⊥, c = 1                              O(n)

  for (u = 1,…,n): ← O(n)
    // If u has not been explored
    if (comp[u] == ⊥):
each:
O(m+n)   S = set of nodes found by DFS(G,u)   ← nodes which u can reach
O(m+n)   T = set of nodes found by DFS(G^R,u) ← nodes which can reach u
         // S ∩ T contains SCC(u)
O(n)     label S ∩ T with c
O(1)     c = c + 1

  return comp
```

Total: $O(n(m+n))$

SCCs Form a DAG!

SCC Graph: acyclic

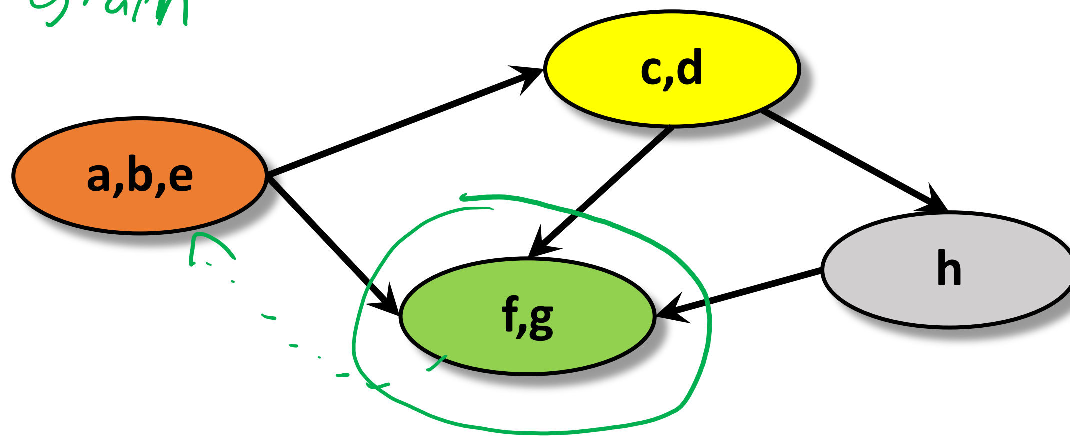




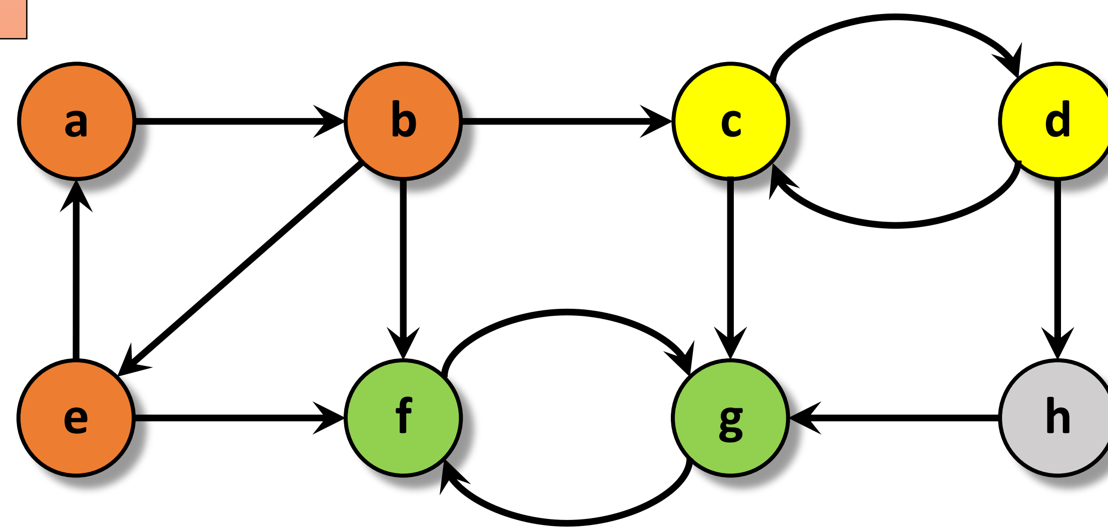
"Before I begin, one of the acronyms I'm going to use is completely made up. See if you can figure out which one."

Clever use of DFS for SCC

sink component: has out-deg = 0
in SCC graph



Observation: DFS from any node in a sink component finds that component



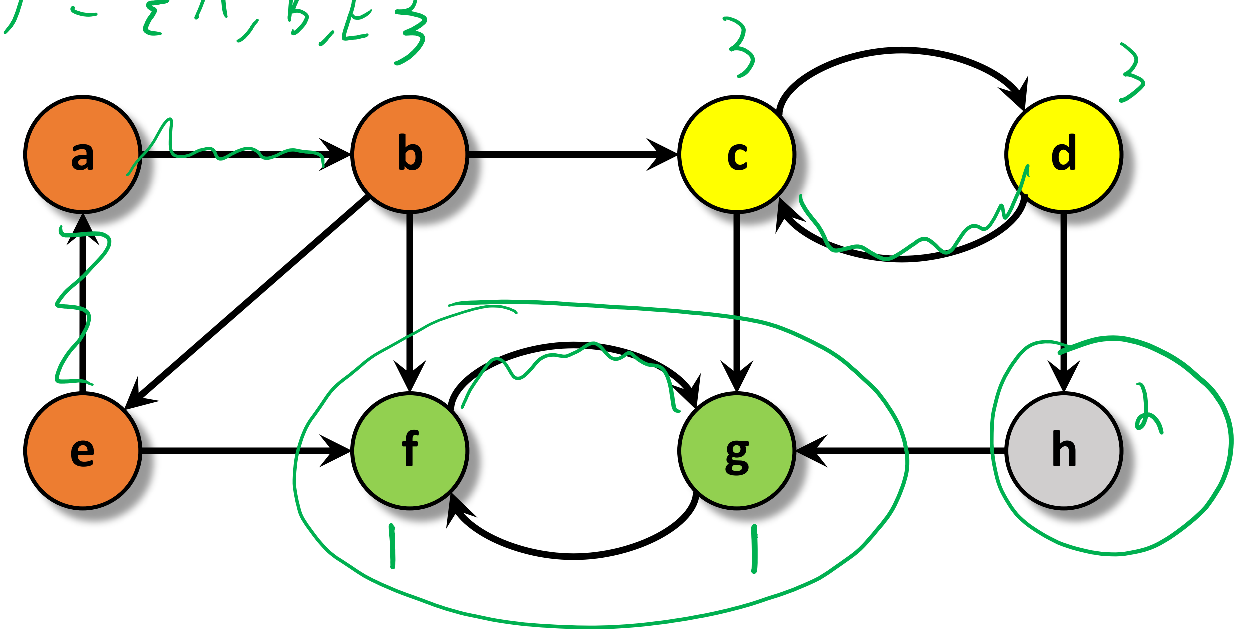
SCC Algorithm Template

DFS(F) = {F, G}

DFS(H) : {H}

DFS(D) : {C, D}

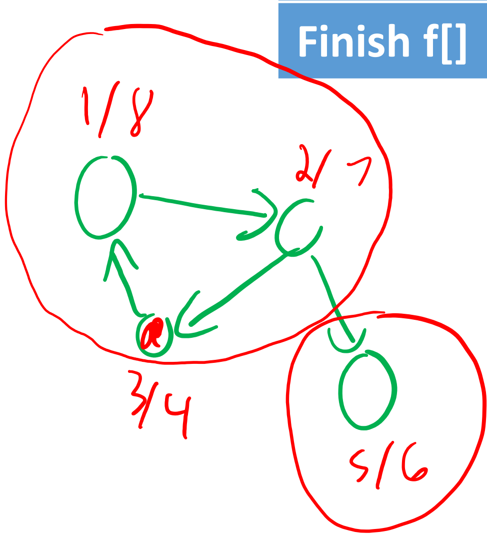
DFS(E) : {A, B, E}

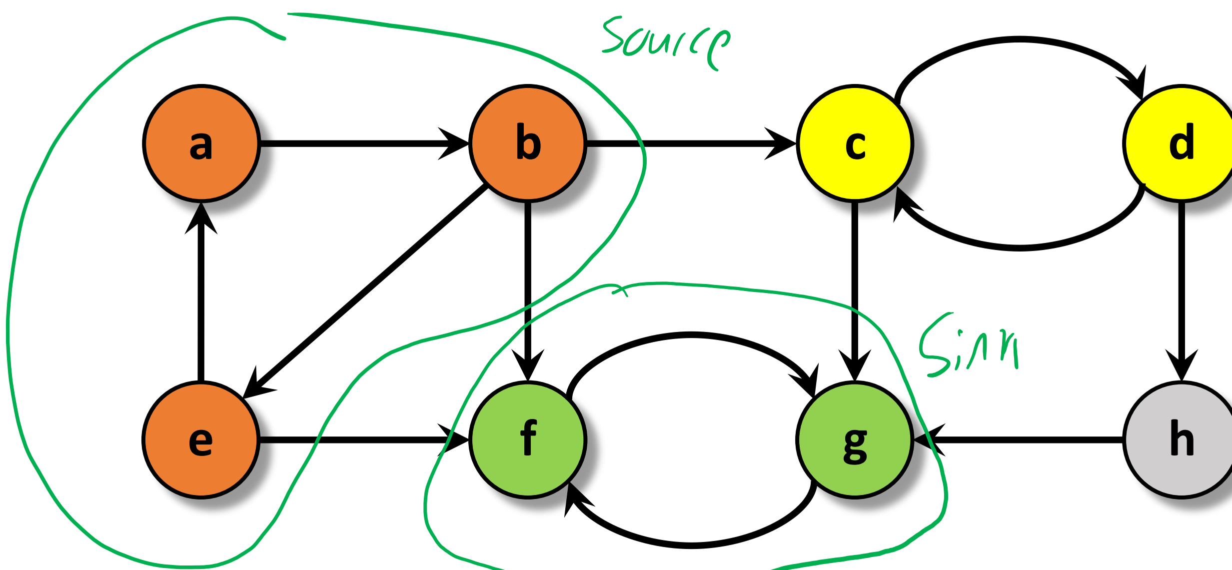


SCC Algorithm Template

- Repeat until all nodes marked:
  - Find a node in a sink component of G
  - Run DFS(u) to find SCC of u
  - Mark the nodes in SCC of u so not visited again
- How to find a node in a sink component?

Vertex	a	b	c	d	e	f	g	h
Finish f[]	16	15	12	11	14	8	9	10



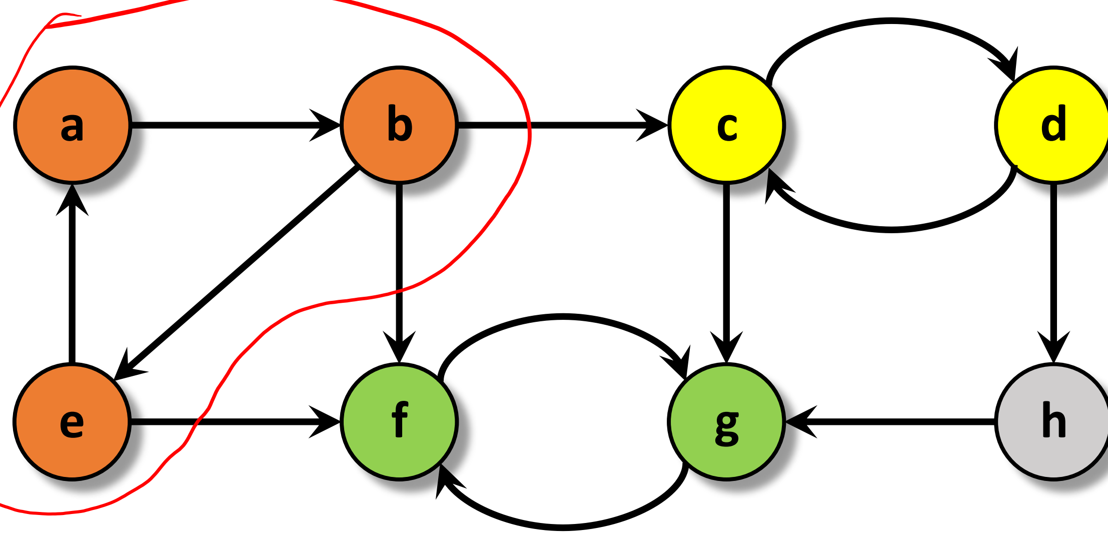


SCC Algorithm Template

- Repeat until all nodes marked:
  - Find a node in a sink component of G
  - Run DFS(u) to find SCC of u
  - Mark the nodes in SCC of u so not visited again
- How to find a node in a sink component?

Vertex	a	b	c	d	e	f	g	h
Finish f[]	16	15	12	11	14	8	9	10

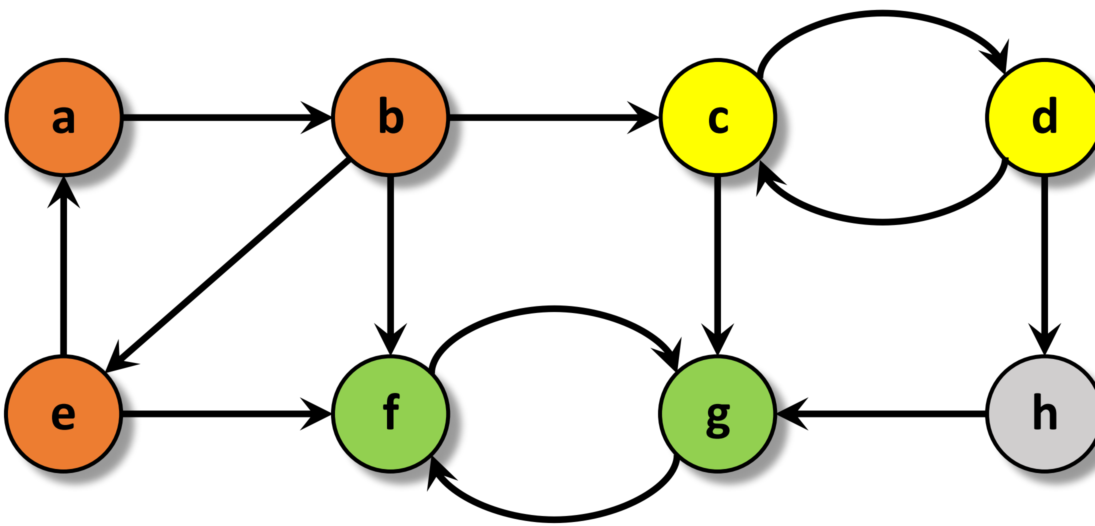
Fact: Node with largest finish time is in a *source* component



SCC Algorithm Template

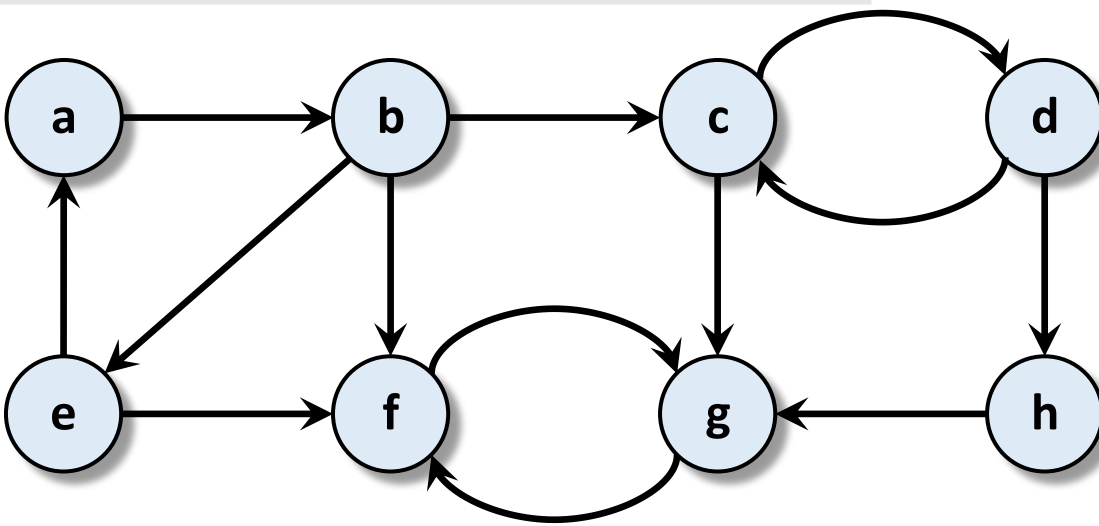
- Repeat until all nodes marked:
  - Find a node in a sink component of G
  - Run DFS(u) to find SCC of u
  - Mark the nodes in SCC of u so not visited again
- How to find a node in a sink component?
  - Node with largest finish time in reverse of G!

Fact: Node with largest finish time is in a *source* component



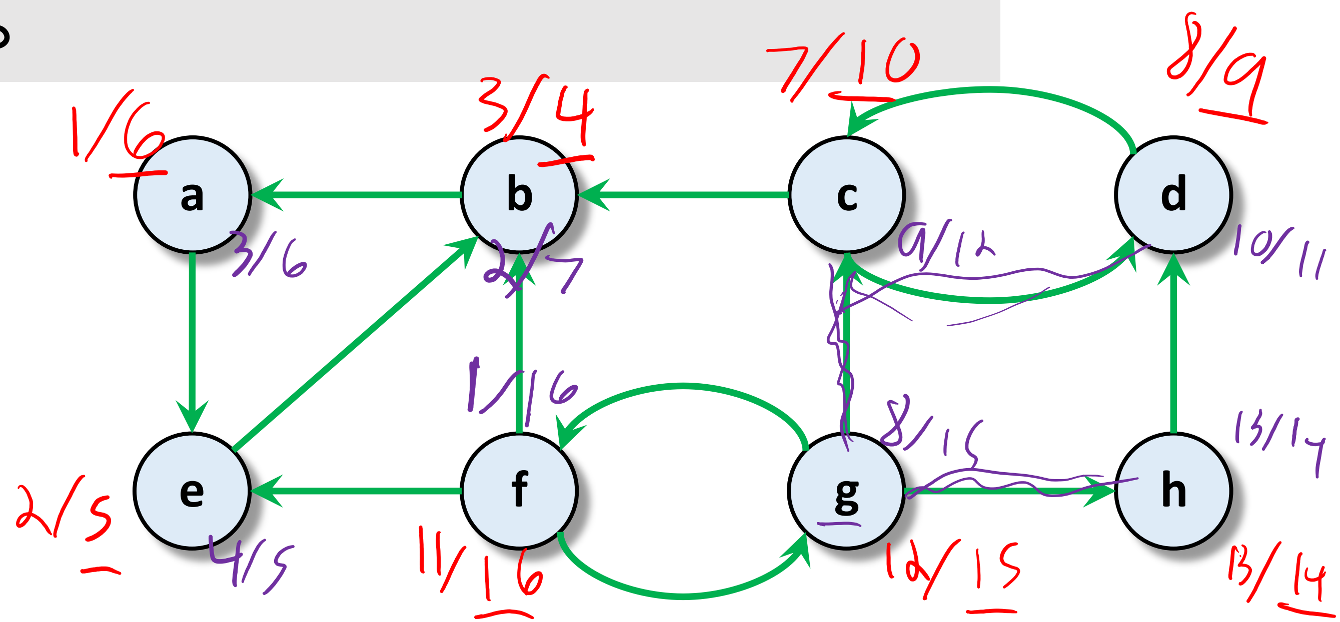
Linear-time algorithm for SCC

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



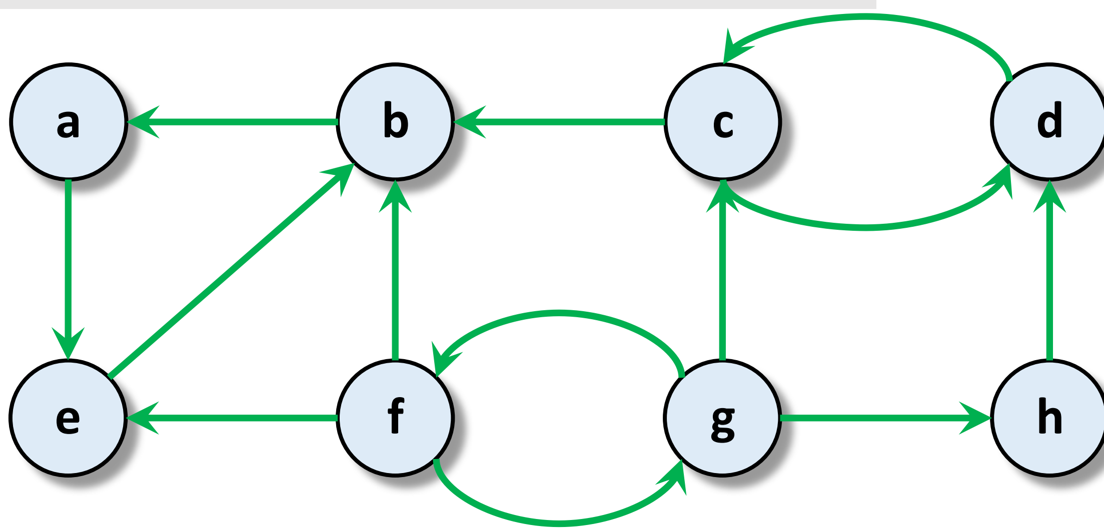
SCC Algorithm in Linear Time

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



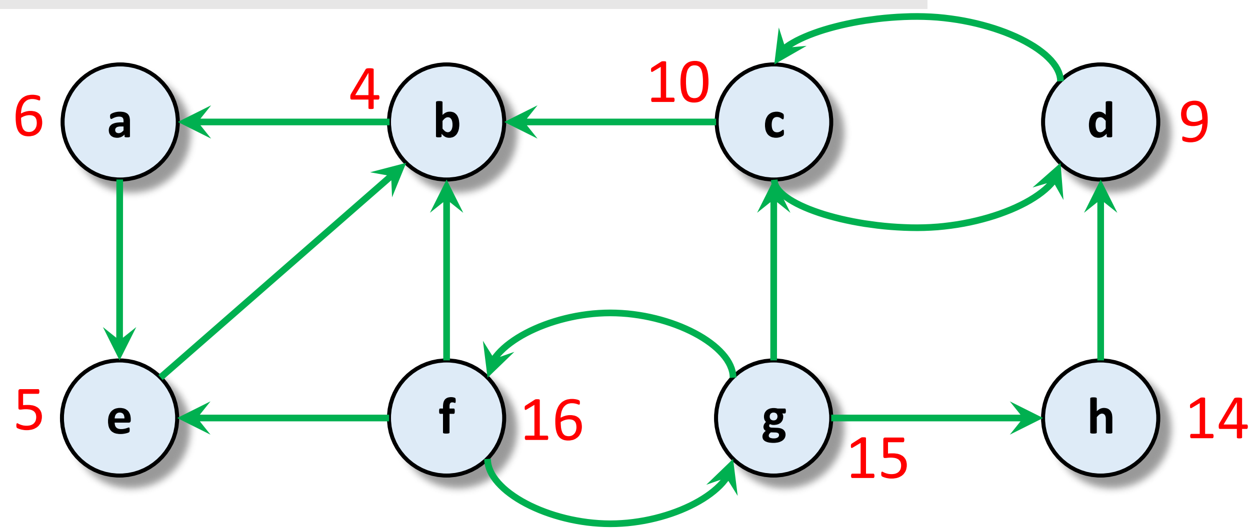
SCC Algorithm in Linear Time

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



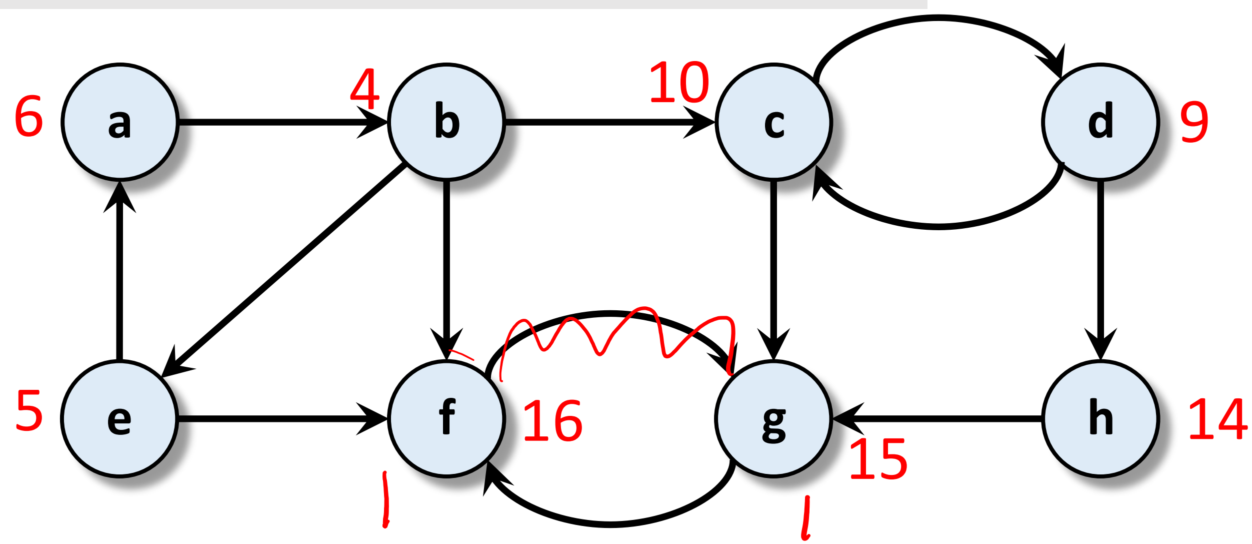
SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



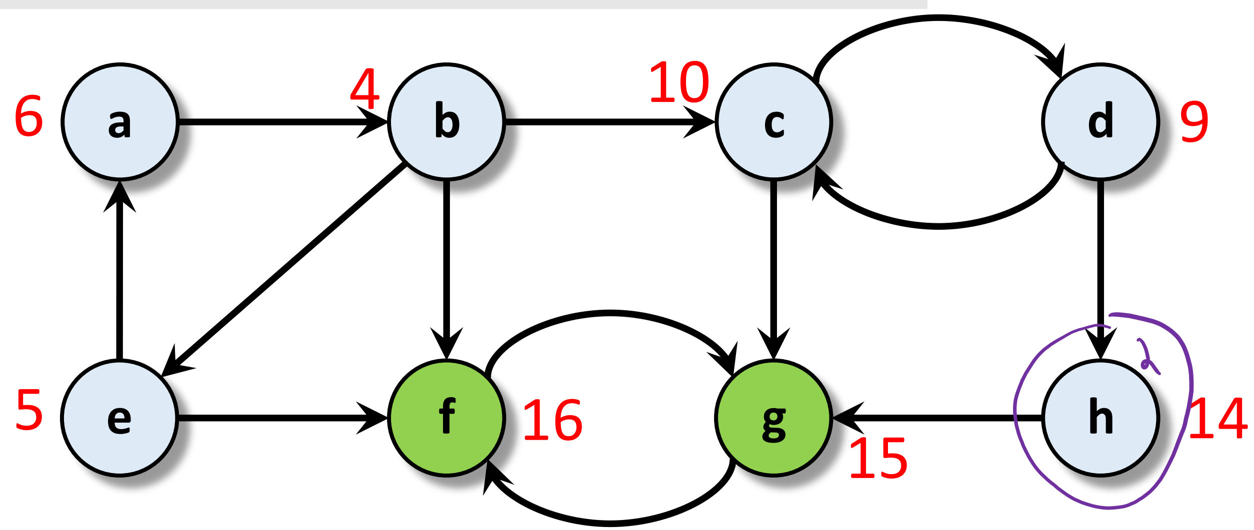
SCC Algorithm in Linear Time

```
SCC(G):
  G^R = G with all edges "reversed"
  DFS of G^R to compute finish times f^R
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of f^R)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



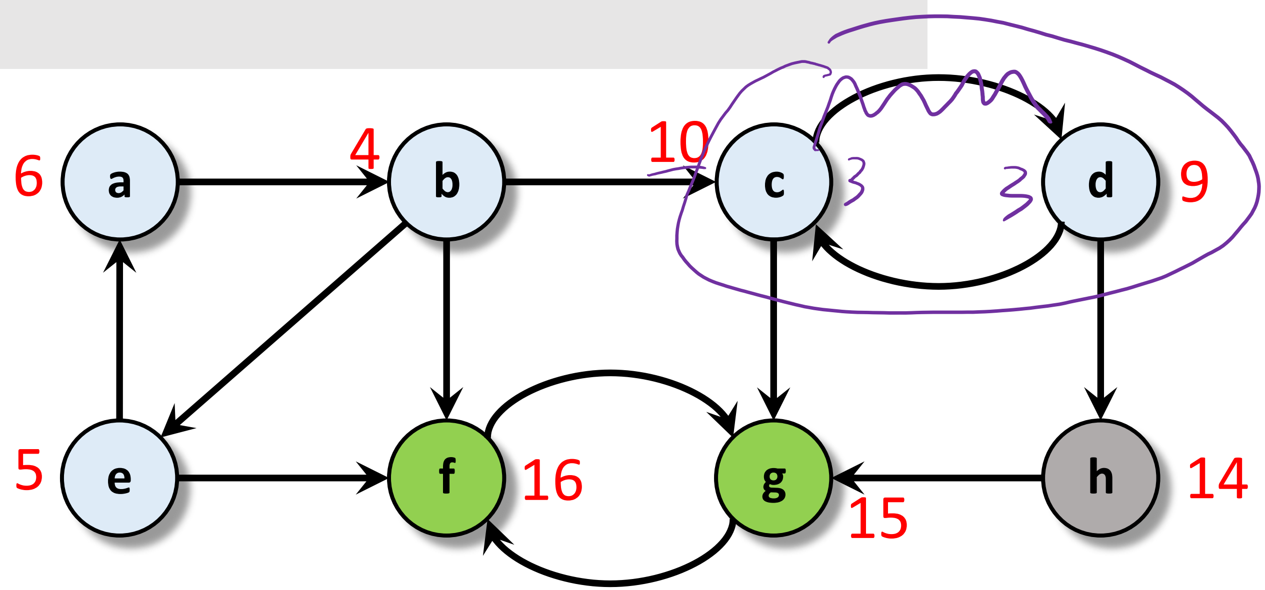
SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



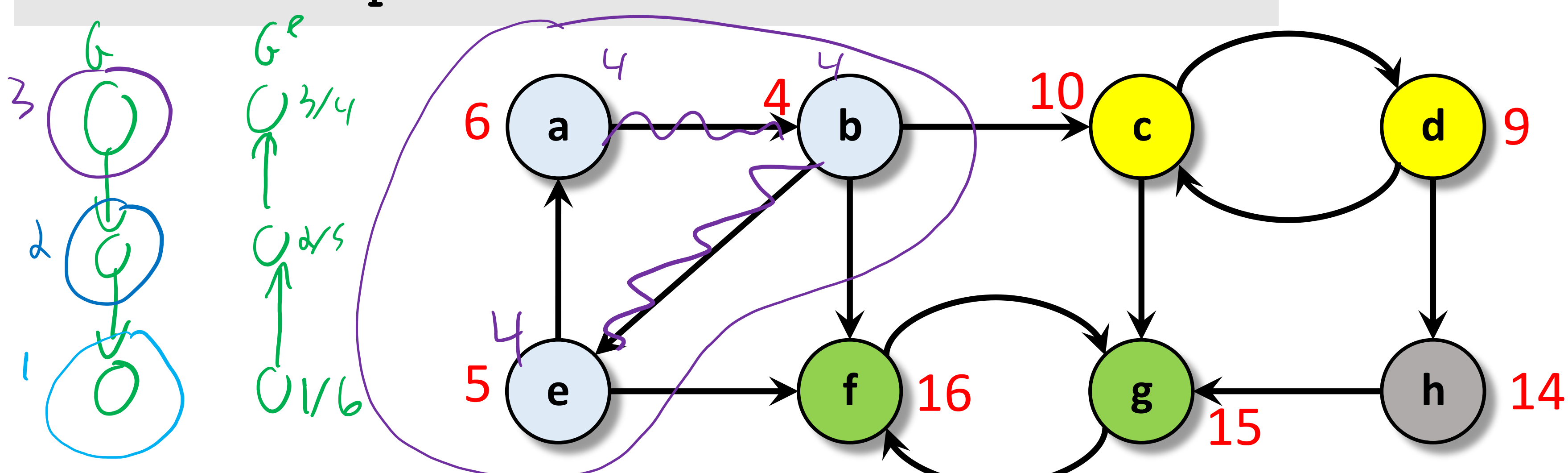
SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



SCC Algorithm in Linear Time

```
SCC(G):
  Gᴿ = G with all edges "reversed"
  DFS of Gᴿ to compute finish times fᴿ
  comp[1:n] = ⊥, c = 1
  for (u in reverse order of fᴿ)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G
      for v in S: comp[v]= c
      c = c + 1
  return comp
```



SCC Algorithm in Linear Time

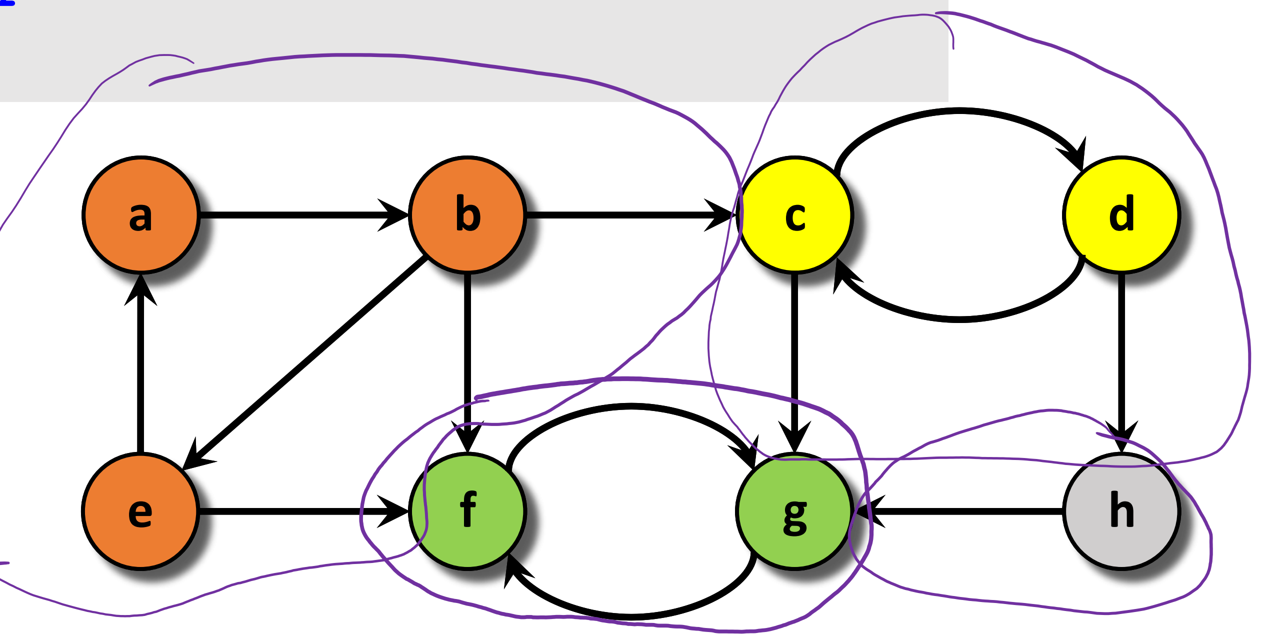
O → O → O → O → ... → ●

```
SCC(G):
  G^R = G with all edges "reversed"                    O(m)
  DFS of G^R to compute finish times f^R               O(n+m)
  comp[1:n] = ⊥, c = 1                                 O(n)
  for (u in reverse order of f^R)
    if (comp[u] == ⊥):
      S = set of nodes found by DFS(u) of G            TOTAL: O(n+m)
      for v in S: comp[v]= c
      c = c + 1
  return comp
```

Total r/t: $O(n+m)$

visit each else at most once

$\Theta(n+m)$ time



- Problems: counting students, stable matching, sorting, n-digit multiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, prefix-free encoding, graph exploration, bipartiteness, topological sorting, **(strongly) connected components**

- Alg. techniques: divide & conquer, dynamic programming, greedy

- Analysis: asymptotic analysis, recursion trees, Master Thm., Graph Terminology/representations

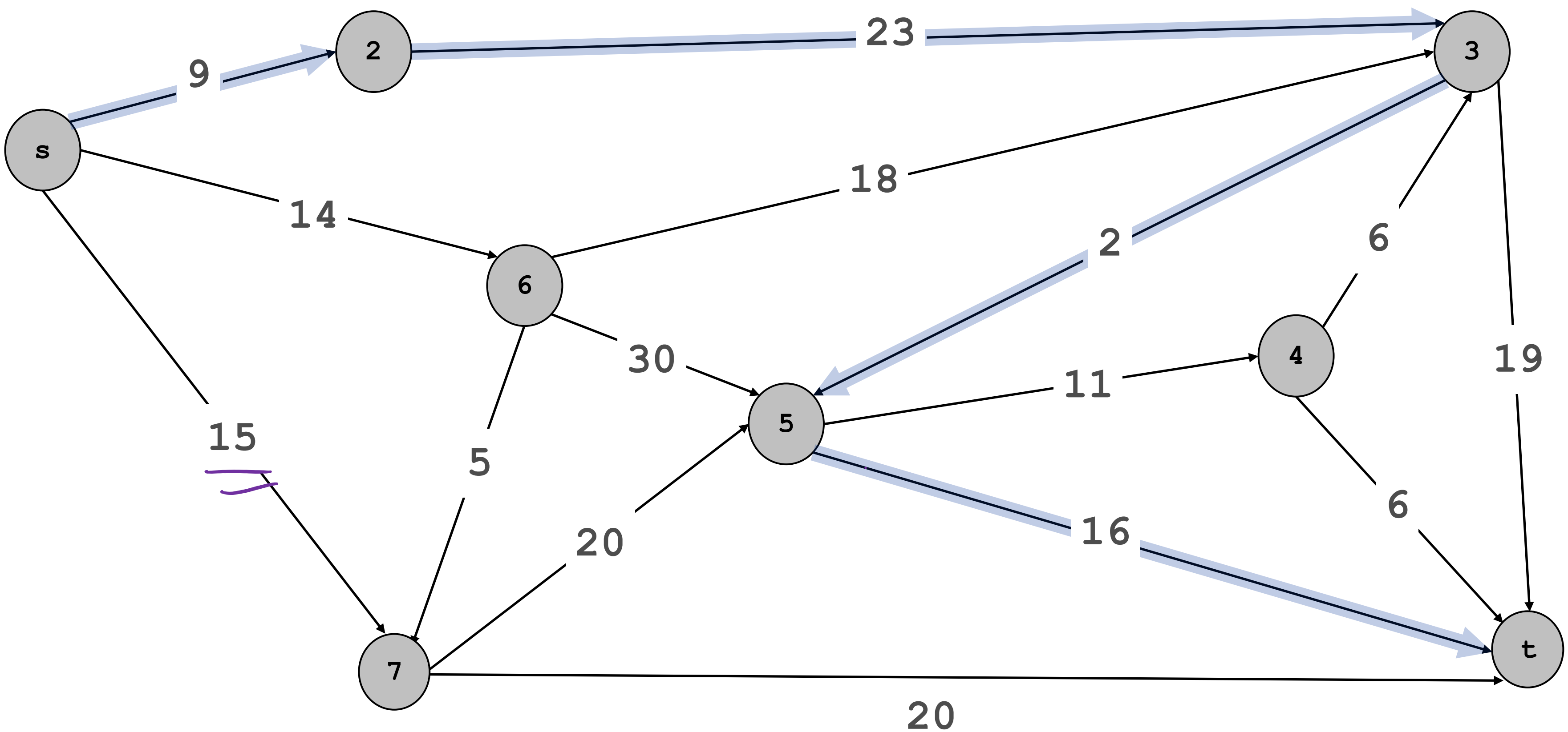
- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

Strongly Connected Components Recap

- **Problem:** Given a directed graph G, split it into strongly connected components
- **Input:** Directed graph $G = (V, E)$
- **Output:** A labeling of the vertices by their strongly connected component
- **Punchline: $O(n+m)$ time** algorithm for SCCs
  - Clever use of DFS on G and reverse of G
  - Can also compute the meta-graph DAG of SCCs
- Can be directly invoked in other algorithms

Shortest Paths

Weighted Graphs



Weighted Graphs

- **Definition:** A weighted graph $G = (V, E, \{w(e)\})$
  - V is the set of vertices
  - $E \subseteq V \times V$ is the set of edges
  - $w(e) \in \mathbb{R}$ are edge weights
  - Can be directed or undirected

- **Today:**
  - Directed graphs (one-way streets)
  - Non-negative edge weights ($w(e) \geq 0$)

Shortest Paths

- In weighted graphs, the length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of its edge weights:

$$\ell(P) = \sum_{e \in P} w(e)$$

- The distance $d(s,t)$ is the length of the shortest path from s to t
- **Shortest Path:** given nodes $s, t \in V$, find the shortest path from s to t
- ***Single-Source Shortest Paths:*** *given a node $s \in V$, find the shortest paths from s to* ***every*** *$t \in V$*
- **All-Pairs Shortest Paths:** find the shortest path between every $(s,t) \in V$

- Problems: counting students, stable matching, sorting, n-digit multiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, prefix-free encoding, graph exploration, bipartiteness, topological sorting, (strongly) connected components, **shortest paths**

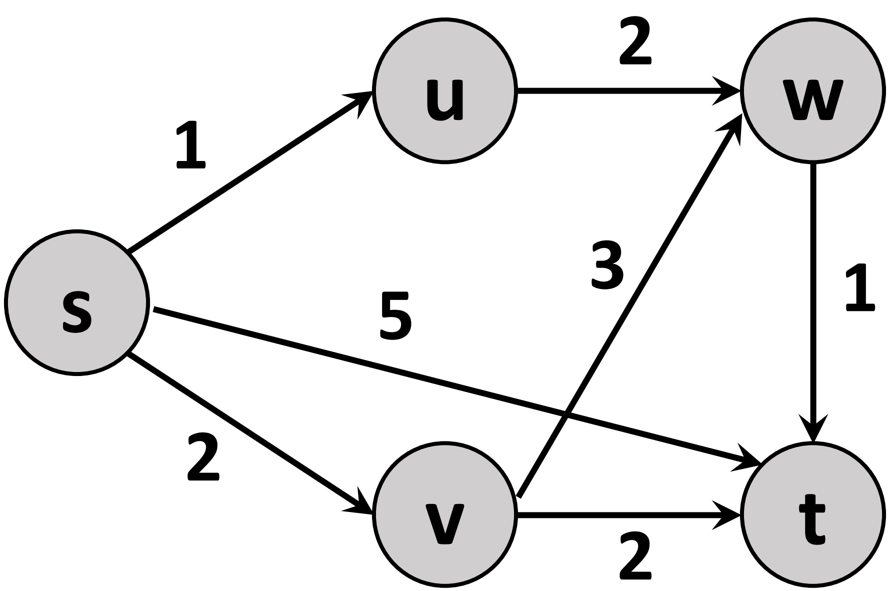
- Alg. techniques: divide & conquer, dynamic programming, greedy

- Analysis: asymptotic analysis, recursion trees, Master Thm., Graph Terminology/representations

- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

Distance

- In weighted graphs, the length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge weights:
- The distance $d(s,t)$ is the length of the shortest path from s to t



$d(s,t) = 4$

$\ell(s-u-w-t) = 4 = 1+2+1$

$\ell(s-t) = 5$

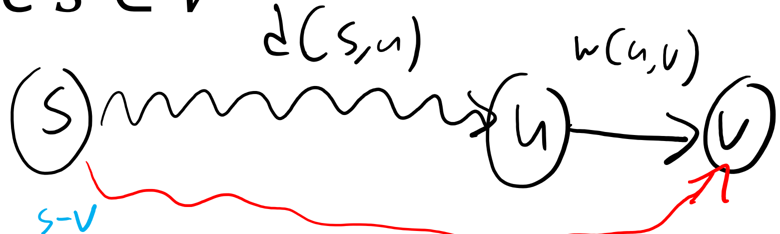
$\ell(s-v-t) = 4$

$\ell(s-v-w-t) = 6$

Structure of Shortest Paths

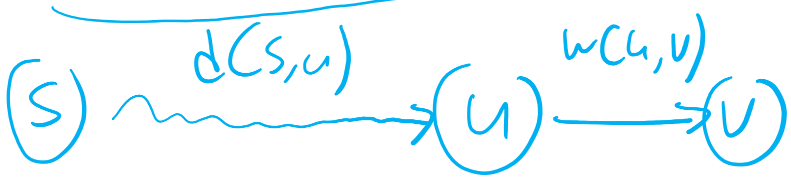
- If $(u, v) \in E$, then $d(s, v) \leq d(s, u) + w(u, v)$ for every node $s \in V$

∃ a path from s to v w/ (u,v) as the final edge of length d(s,u) + w(u,v)



The shortest s-v path can't be any longer than this ↑

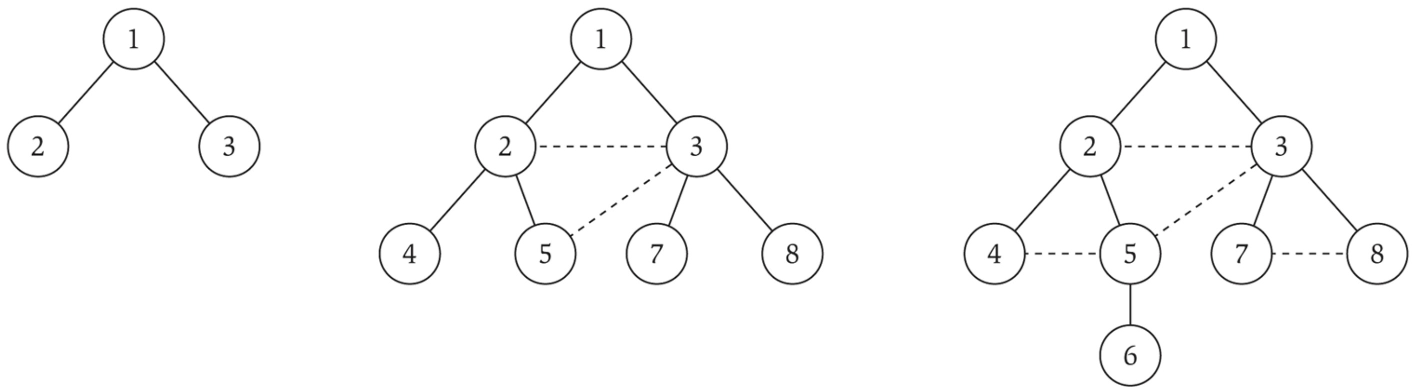
- If $(u, v) \in E$, and $d(s, v) = d(s, u) + w(u, v)$ then there is a shortest $s \rightsquigarrow v$-path ending with (u, v)



No s-v path is shorter than this one

Compare to BFS

- **Thm.:** BFS finds distances from s to other nodes in unweighted graphs
  - L_i contains all nodes at distance i from s
  - Nodes not in any layer are not reachable from s

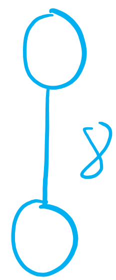


- **Question:** Does running a BFS from s **ever** solve the SSSP problem on weighted graphs?
- **Question:** Does running a BFS from s **always** solve the SSSP problem on weighted graphs?

Compare to BFS

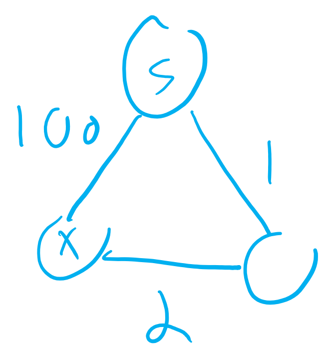
Question: Does running a BFS from *s* **ever** solve the SSSP problem on weighted graphs?

Yes



Question: Does running a BFS from *s* **always** solve the SSSP problem on weighted graphs?

NO

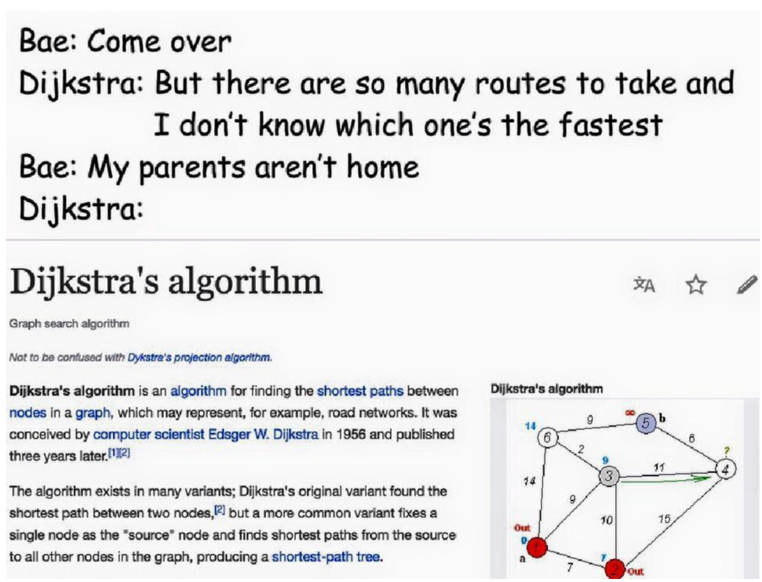


Dijkstra's Algorithm

- **Dijkstra's Shortest Path Algorithm** is a modification of BFS for non-negatively weighted graphs
- **Informal Version:**
  - **Maintain a set X of explored nodes**
  - **Maintain an upper bound on distance for all unexplored nodes**
    - If u is explored, then we know $d(s,u)$ (from the source *s)* (Key Invariant)
    - If u is explored, and (u,v) is an edge, then we know $d(s,v) \leq d(s,u) + w(u,v)$
  - **Explore the "closest" unexplored node**
  - **Repeat until we're done**

Dijkstra's Algorithm

- **Explore the "closest" unexplored node**
  - The unexplored node with the smallest upper bound on its distance
  - Tighten (lower) its out-neighbors' upper bounds (when possible)



- Problems: counting students, stable matching, sorting, n-digit multiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, prefix-free encoding, graph exploration, bipartiteness, topological sorting, (strongly) connected components, shortest paths

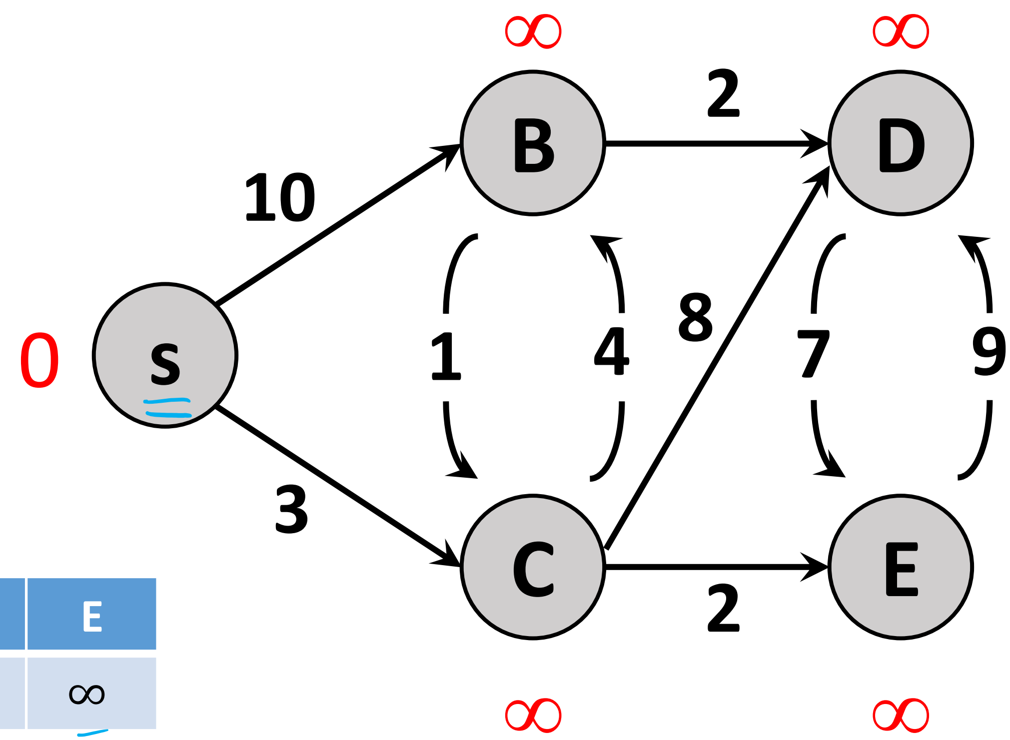
- Alg. techniques: divide & conquer, dynamic programming, greedy, **Dijkstra's**

- Analysis: asymptotic analysis, recursion trees, Master Thm., Graph Terminology/representations

- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

Dijkstra's Algorithm: Demo

Initialize



	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$X = \{ \quad \}$

Dijkstra's Algorithm

- **Explore the "closest" unexplored node**
  - The unexplored node with the smallest upper bound on its distance
  - Tighten its out-neighbors' upper bounds (if we can)

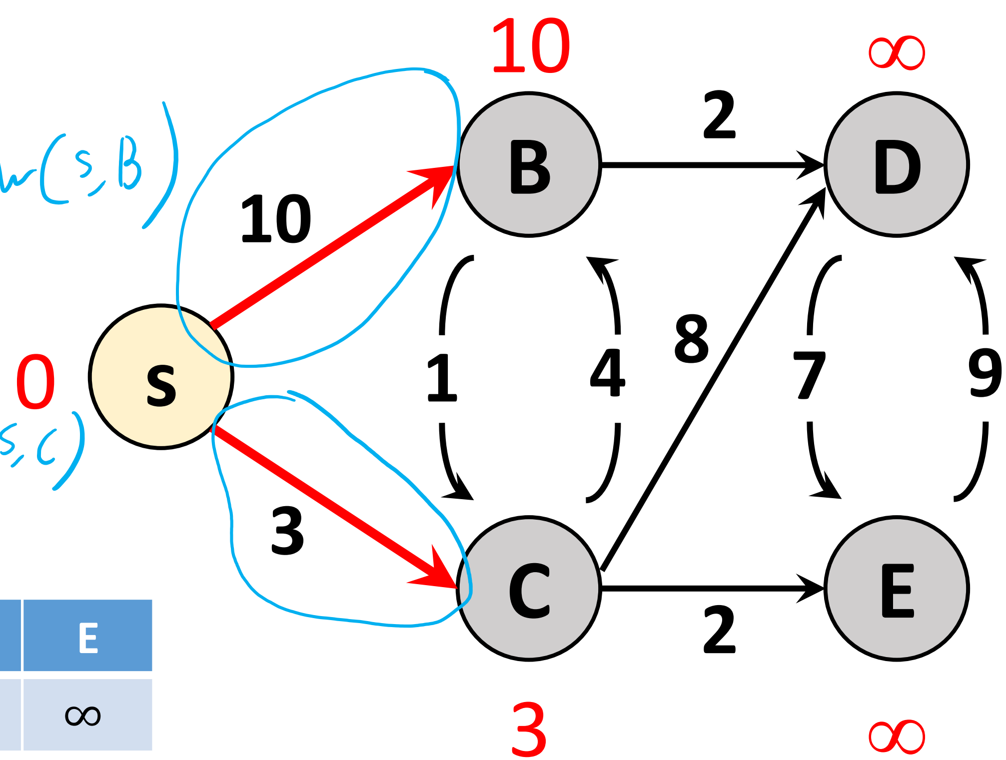
	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

Dijkstra's Algorithm: Demo

Explore *s*

$d(s,B) \le d(s,s) + w(s,B)$
$\le 0 + 10$

$d(s,C) \le d(s,s) + w(s,C)$
$\le 0 + 3$



	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
		10	3	∞	∞

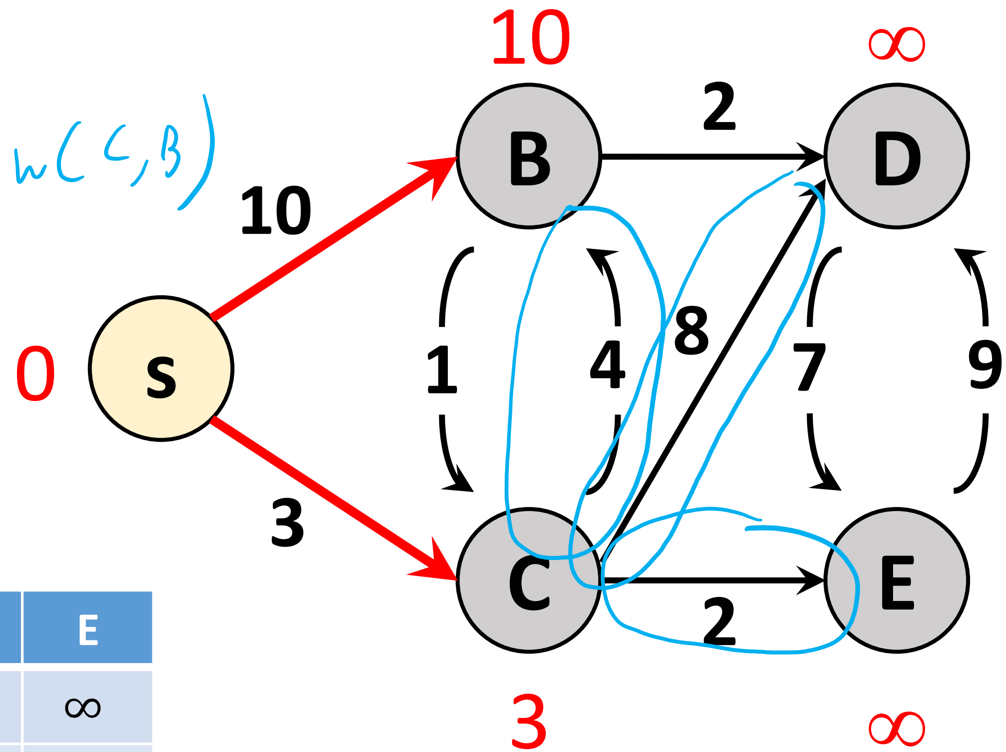
$X = \{ s, C \quad \}$

Dijkstra's Algorithm: Demo

Explore *s*

$d(s,B) \leq d(s,C) + w(C,B)$

$3 + 4 = 7$

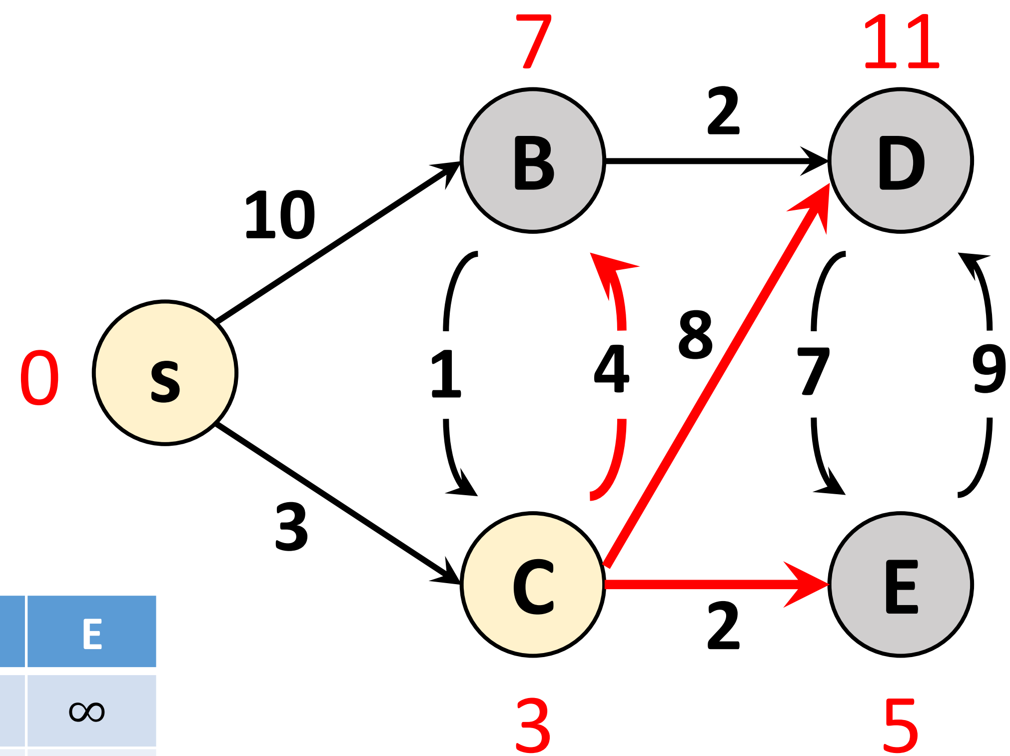


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
		7	–	11	5

$X = \{ s, C \quad \}$

Dijkstra's Algorithm: Demo

Explore C

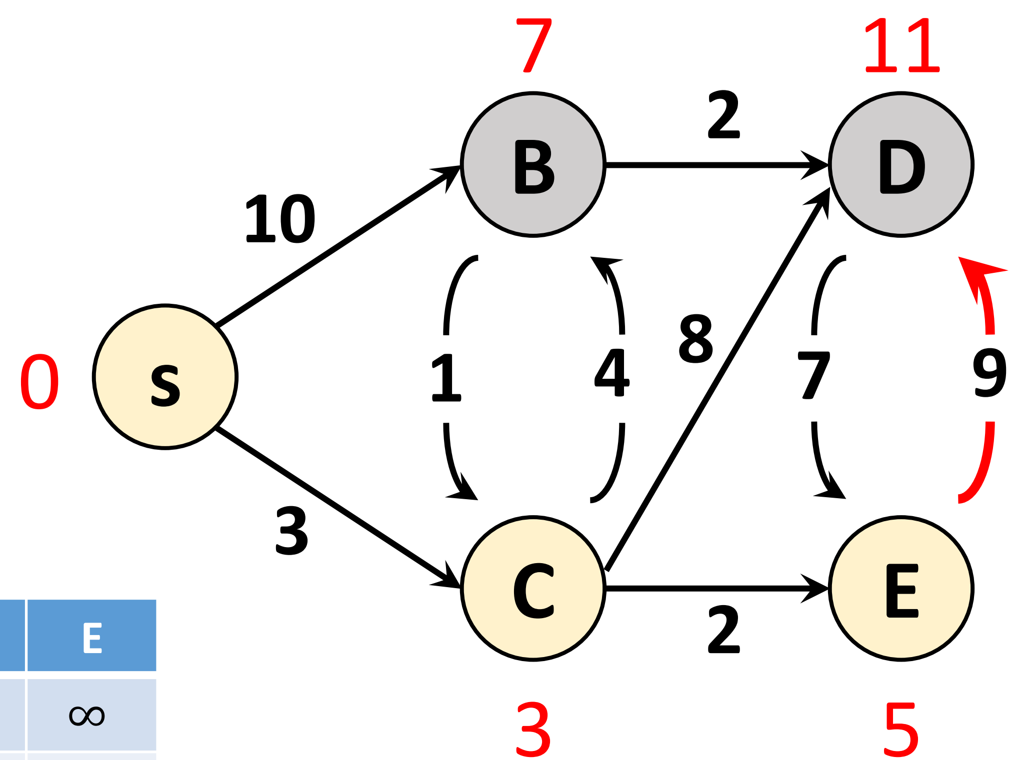


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5

$$X = \{s, C \qquad\qquad \}$$

Dijkstra's Algorithm: Demo

Explore E

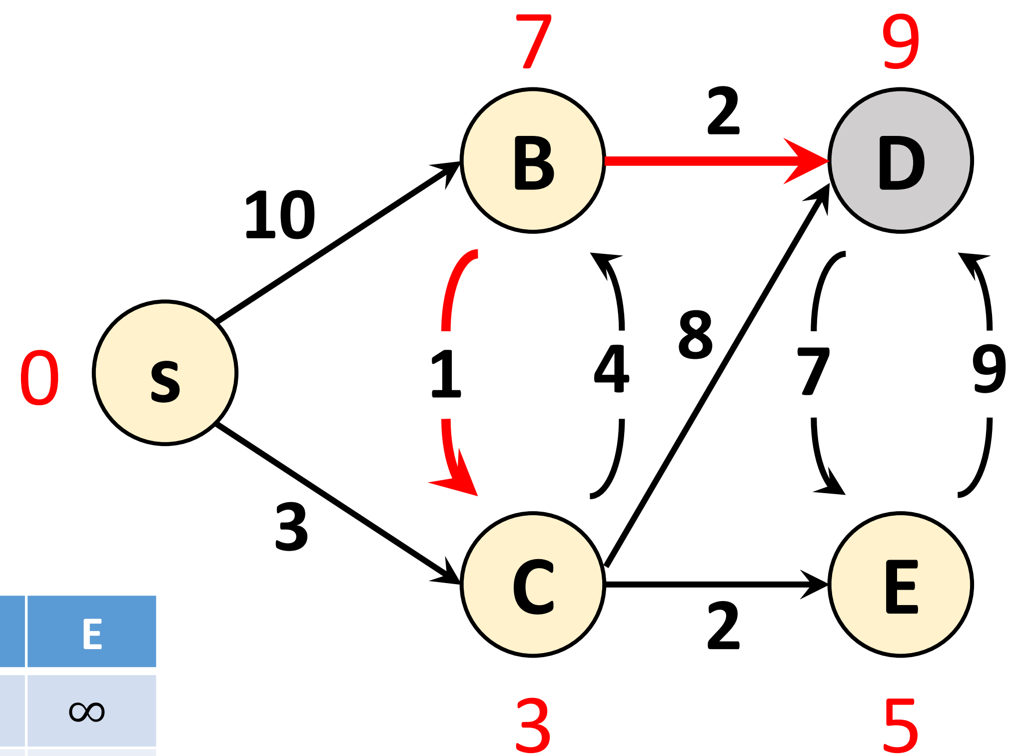


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5

$$X = \{s, C, E \qquad\qquad \}$$

Dijkstra's Algorithm: Demo

Explore B

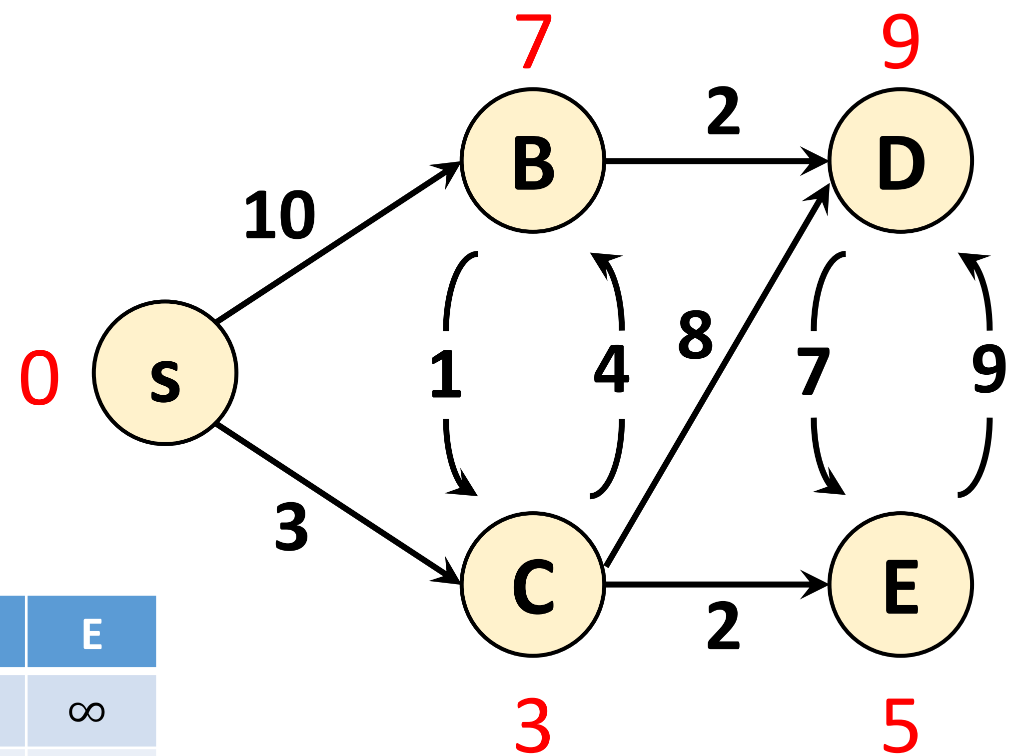


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$X = \{s, C, E, B \qquad \}$$

Dijkstra's Algorithm: Demo

Don't need to explore D

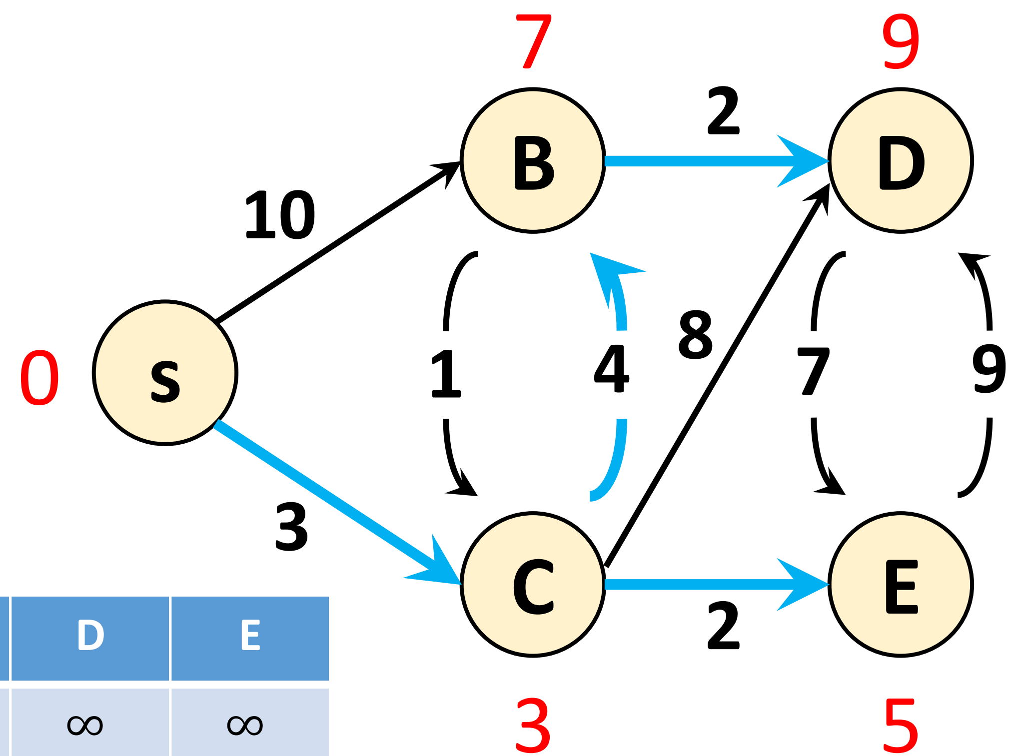


	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$X = \{s, C, E, B, D\}$$

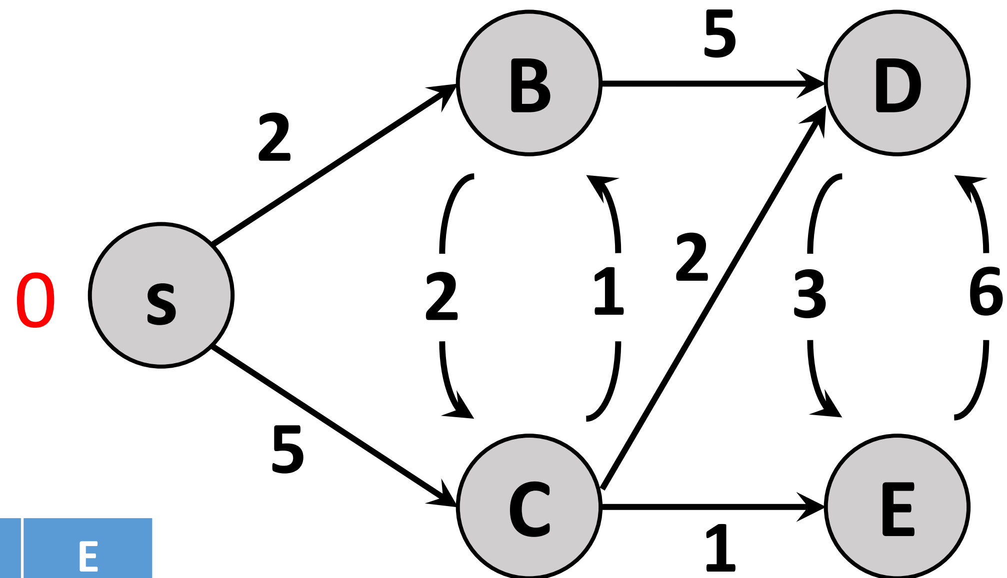
Dijkstra's Algorithm: Demo

Maintain parent pointers so we can find the shortest paths



	s	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

Dijkstra's Algorithm: Practice



	s	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$$X = \{ \qquad \}$$