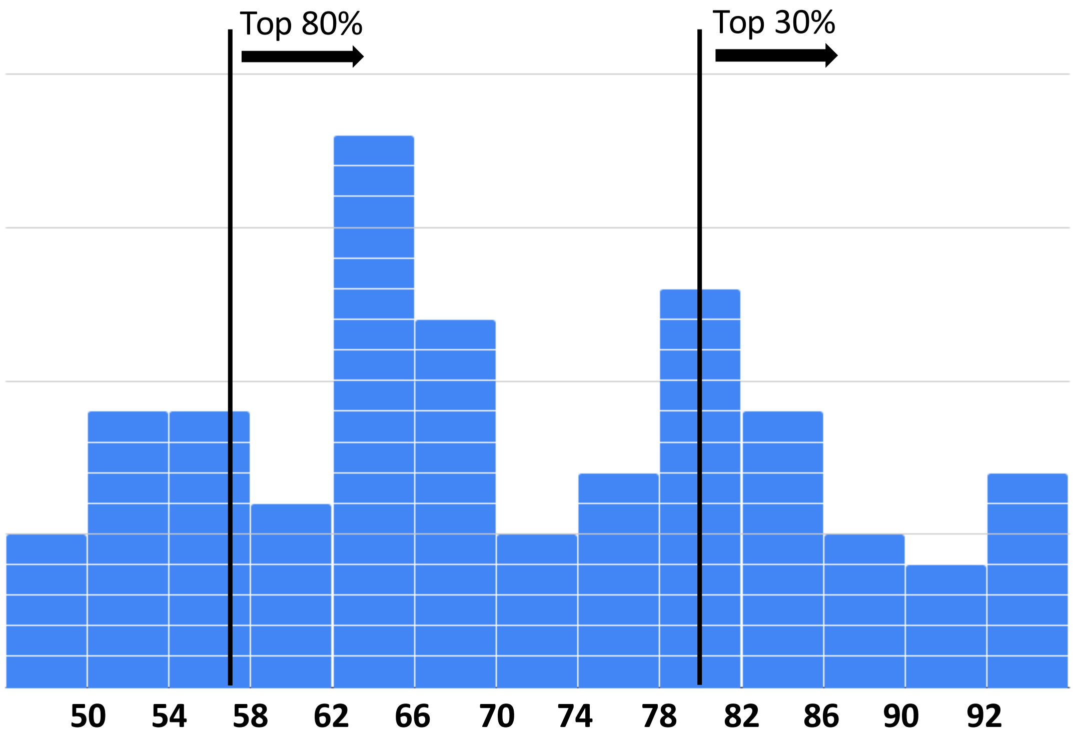
CS3000: Algorithms & Data
Jonathan Ullman

Lecture 13:
- Depth-First Search
- Start Shortest Paths

Oct 23, 2018

Midterm Stats

Midterm Grade Distribution



This midterm is only 15% of your course grade!

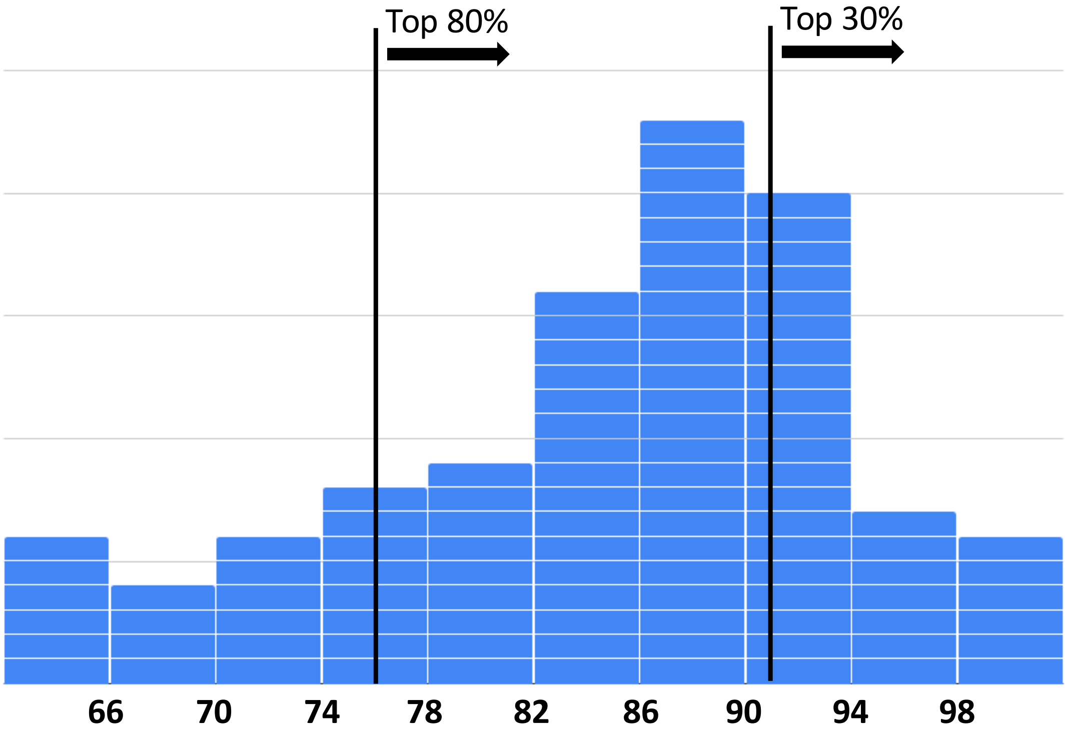
Midterm Grade Distribution

- Lots of points missed for asymptotics, induction:

	Mean
#1 Asymptotics	76%
#2 Recurrences	93%
#3 Induction	65%
#4 Divide-and-Conquer	72%
#5 Stable Matching	90%
#6 Dynamic Programming	63%

- These topics will come up on future exams!

Homework Grade Distribution



Homework is 45% of your course grade!

I have not dropped your lowest grade yet

Depth-First Search (DFS)

Exploring a Graph

- **Problem:** Is there a path from s to t?
- **Idea:** Explore all nodes reachable from s.

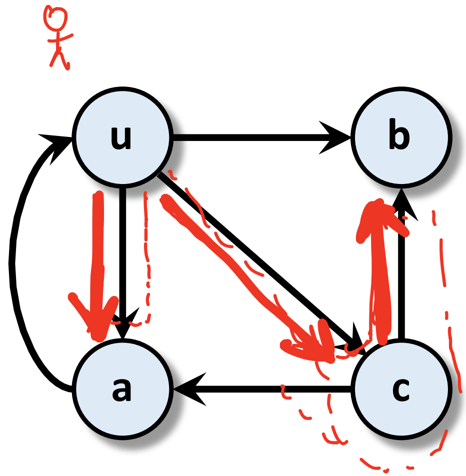
- Two different search techniques:
  - **Breadth-First Search:** explore nearby nodes before moving on to farther away nodes
  - **Depth-First Search:** follow a path until you get stuck, then go back

Depth-First Search

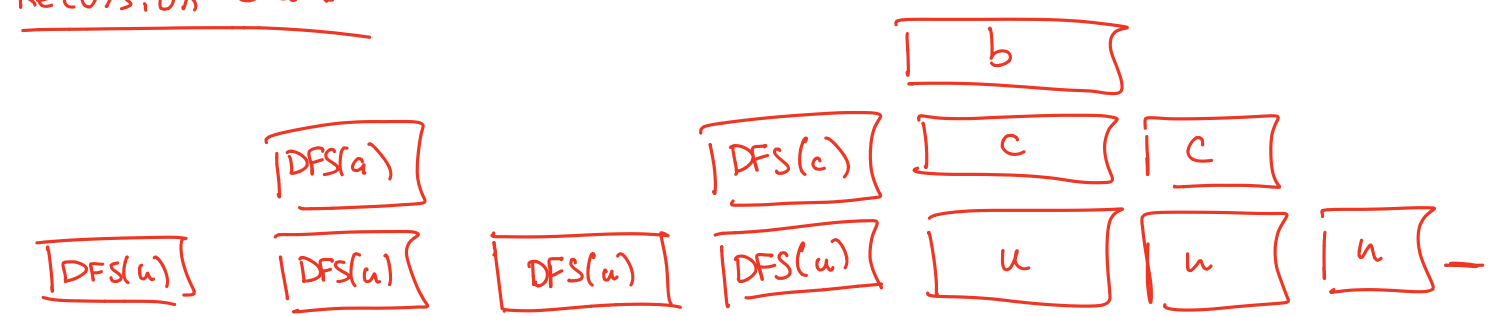
```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)
```

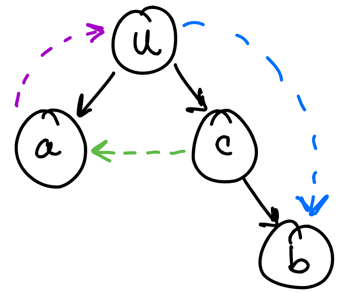


Recursion Stack



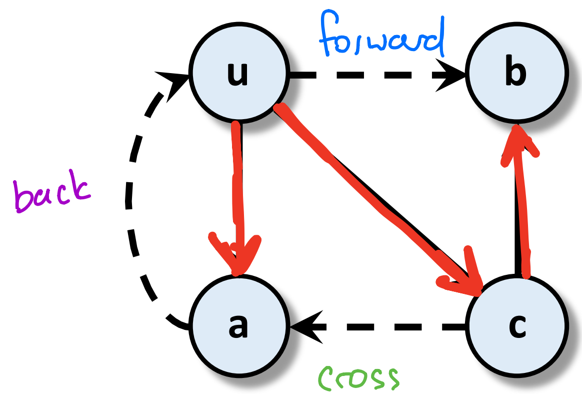
Depth-First Search

- **Fact:** The parent-child edges form a (directed) tree
- **Each edge has a type:**



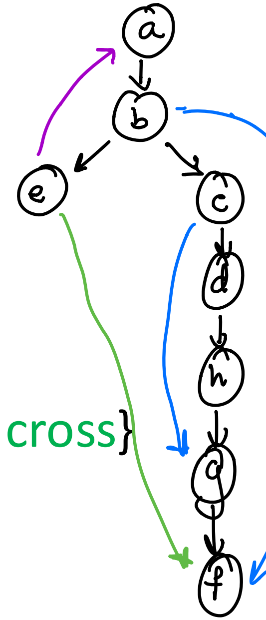
  - **Tree edges:** $(u, a), (u, c), (c, b)$
    - These are the edges that explore new nodes
  - **Forward edges:** (u, b)
    - Ancestor to descendant
  - **Backward edges:** (a, u)
    - Descendant to ancestor
  - **Cross edges:** (c, a)
    - No ancestral relation

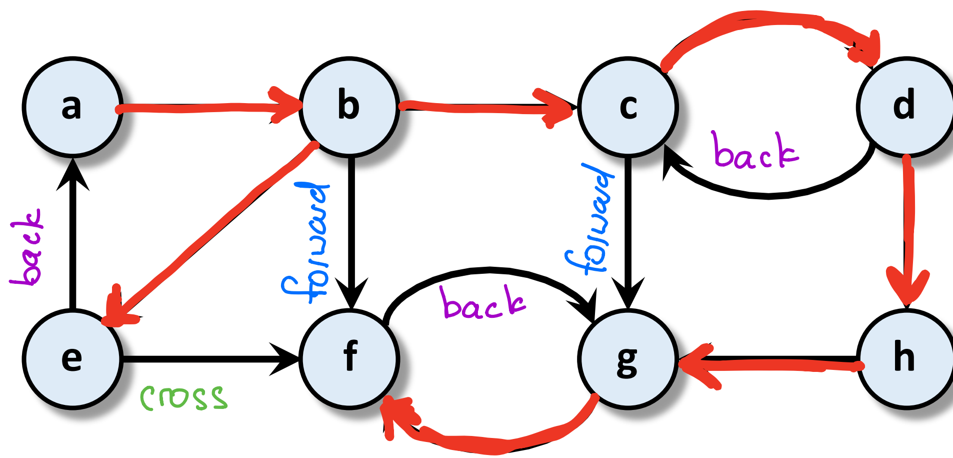
Backward edges give directed cycles



Ask the Audience

- DFS this graph starting from node a
  - Search in alphabetical order
  - Label edges as { tree , forward , backward , cross}





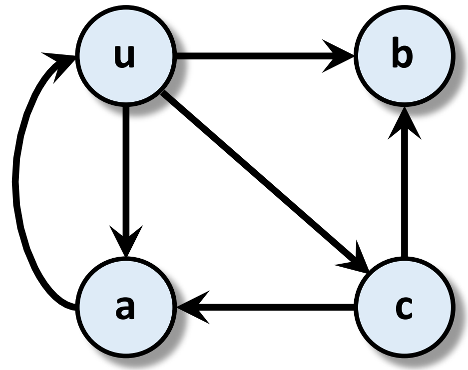
Post-Ordering

```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)

  post-visit(u)
```



Vertex	Post-Order
u	4
a	1
b	2
c	3

- Maintain a counter **clock**, initially set **clock = 1**
- **post-visit(u):**
  set postorder[u]=clock, clock=clock+1

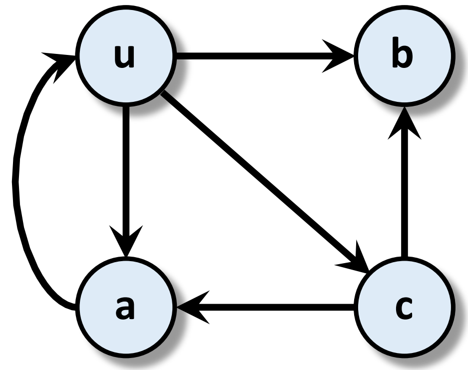
Pre-Ordering

```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  pre-visit(u)

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)
```

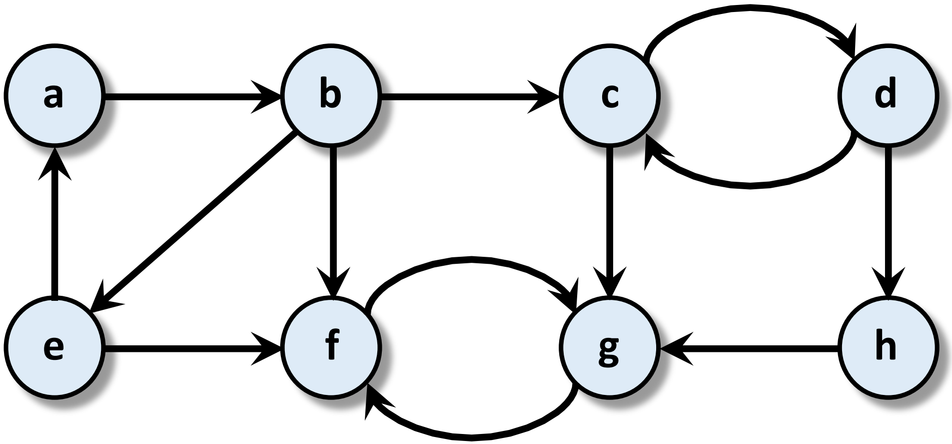


Vertex	Pre-Order
u	1
a	2
b	4
c	3

- Maintain a counter **clock**, initially set **clock = 1**
- **pre-visit(u):**
  set preorder[u]=clock, clock=clock+1

Ask the Audience

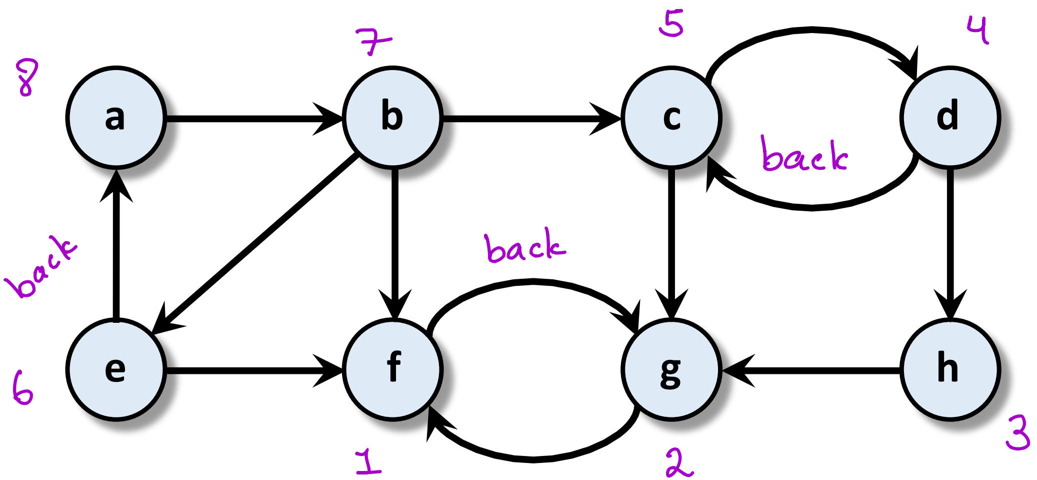
- Compute the **post-order** of this graph
  - DFS from a, search in alphabetical order



Vertex	a	b	c	d	e	f	g	h
Post-Order	8	7	5	4	6	1	2	3

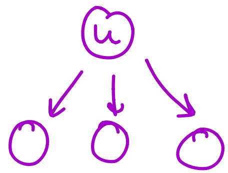
Ask the Audience

- **Observation:** if postorder[u] < postorder[v] then (u,v) is a backward edge



Vertex	a	b	c	d	e	f	g	h
Post-Order	8	7	5	4	6	1	2	3

Ask the Audience

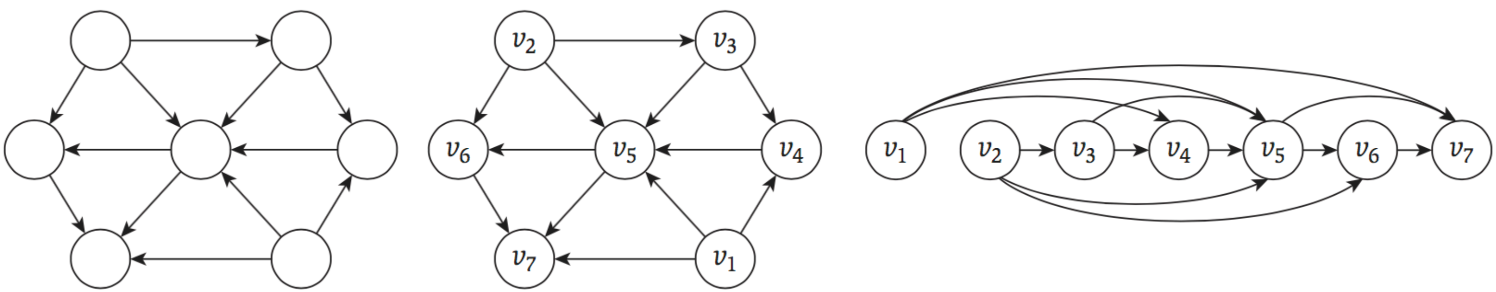


- **Observation:** if postorder[u] < postorder[v] then (u,v) is a backward edge
  - DFS(u) can't finish until its children are finished
    - If (u,v) is a tree edge, then postorder[u] > postorder[v]
    - If (u,v) is a forward edge, then postorder[u] > postorder[v]
  - If postorder[u] < postorder[v], then DFS(u) finishes before DFS(v), thus DFS(v) is not called by DFS(u)
  - When we ran DFS(u), we must have had explored[v]=1
    - Thus, DFS(v) started before DFS(u)
  - DFS(v) started before DFS(u) but finished after
    - Can only happen for a backward edge

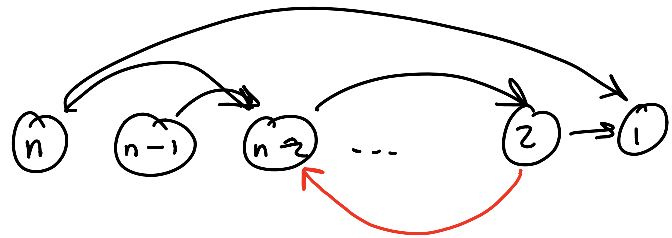
Fast Topological Ordering

Topological Ordering (TO)

- **DAG:** A directed graph with no directed cycles
- Any DAG can be **toplogically ordered**
  - Label nodes $v_1, \ldots, v_n$ so that $(v_i, v_j) \in E \implies j > i$



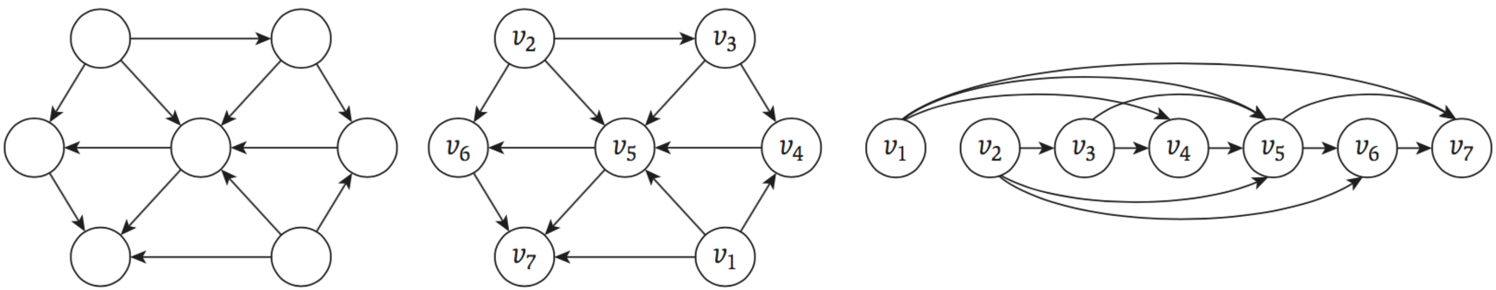
Ask the Audience



- **Claim:** ordering nodes by decreasing postorder gives a topological ordering
- **Proof:**
  - A DAG has no backward edges
  - Suppose this is **not** a topological ordering
    - That means there exists an edge (u,v) such that postorder[u] < postorder[v]
    - We showed that any such (u,v) is a backward edge
    - But there are no backward edges, contradiction!

Topological Ordering (TO)

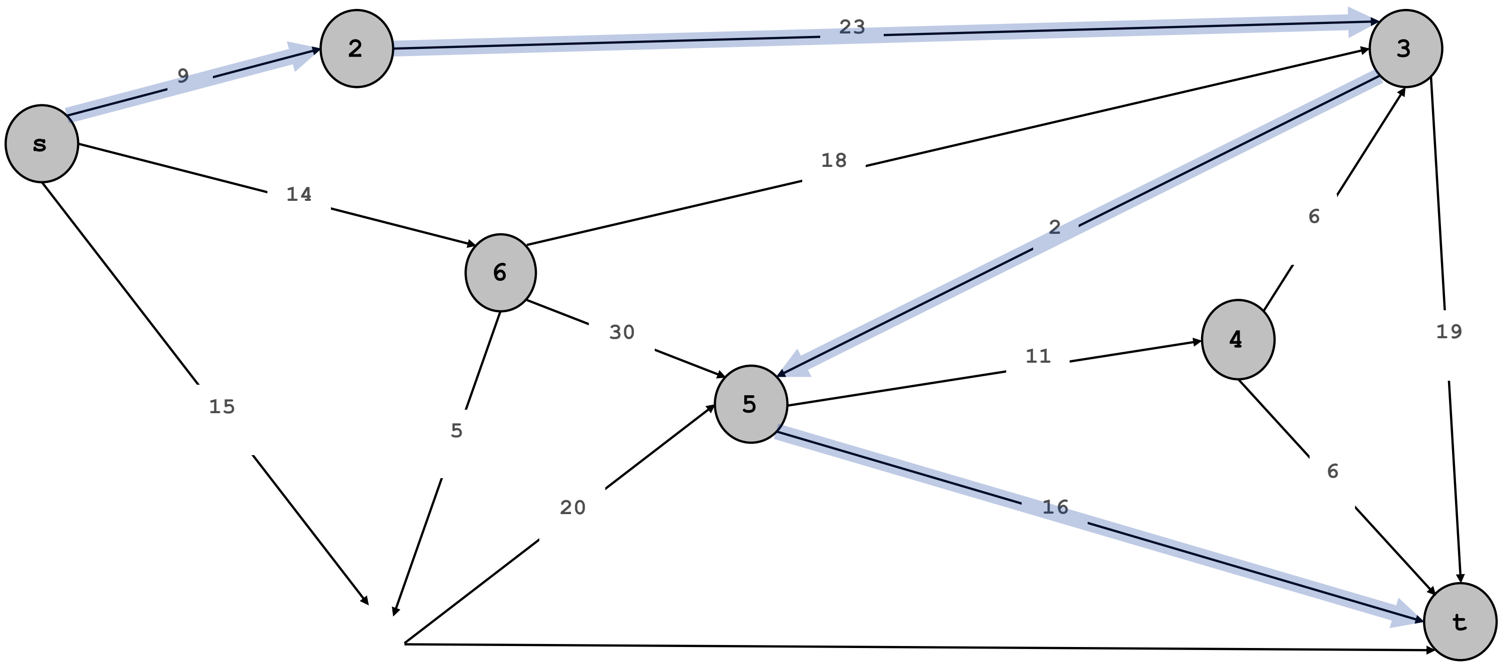
- **DAG:** A directed graph with no directed cycles
- Any DAG can be **toplogically ordered**
  - Label nodes $v_1, \dots, v_n$ so that $(v_i, v_j) \in E \Longrightarrow j > i$



- Can compute a TO in $O(n + m)$ time using DFS
  - Reverse of post-order is a topological order

Shortest Paths

Navigation



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/lengths/capacities
  - Can be directed or undirected

- **Today:**
  - Directed graphs (one-way streets)
  - Strongly connected (there is always some path)
  - Non-negative edge lengths ($\ell(e) \geq 0$)

Shortest Paths

$w(v_1 \to v_2)$ $w(v_2 \to v_3)$

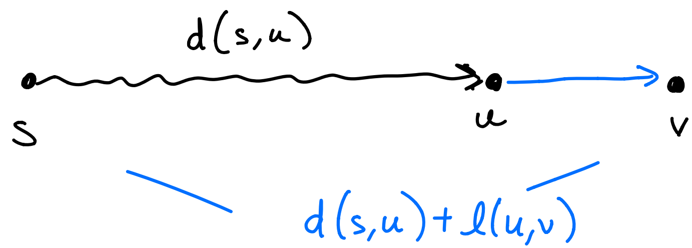
- The length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge lengths

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

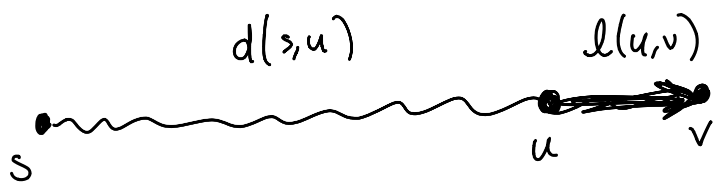
- The distance $d(s, t)$ is the length of the shortest path from s to t
- **Shortest Path:** given nodes $s, t \in V$, find the shortest path from s to t
- **Single-Source Shortest Paths:** given a node $s \in V$, find the shortest paths from s to **every** $t \in V$

Structure of Shortest Paths

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

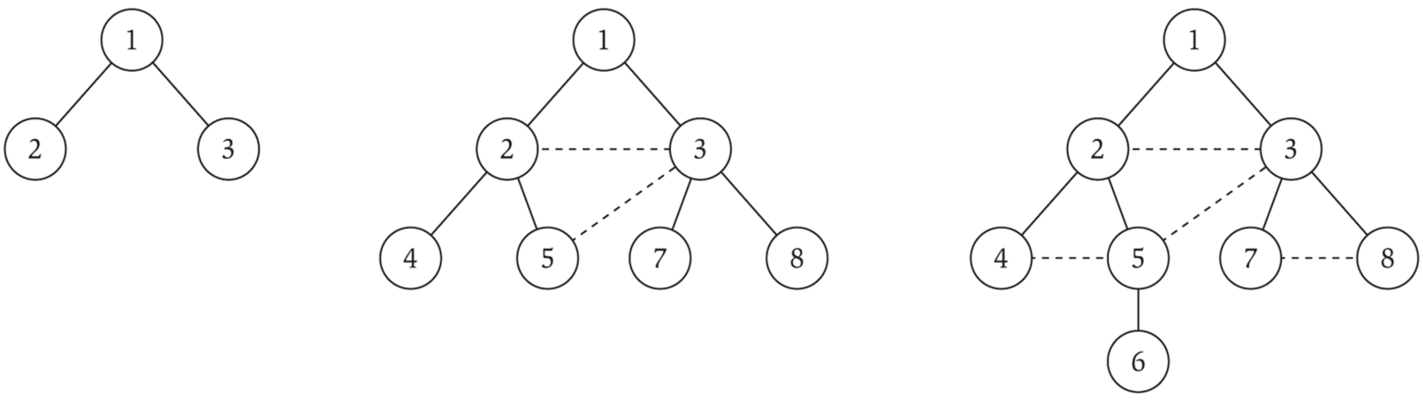


- If $(u, v) \in E$, and $d(s, v) = d(s, u) + \ell(u, v)$ then there is a shortest $s \rightsquigarrow v$-path ending with (u, v)

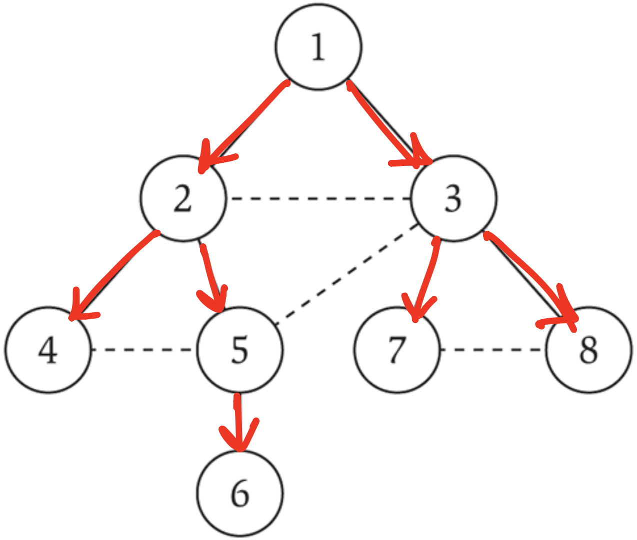


Compare to BFS

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



Compare to BFS



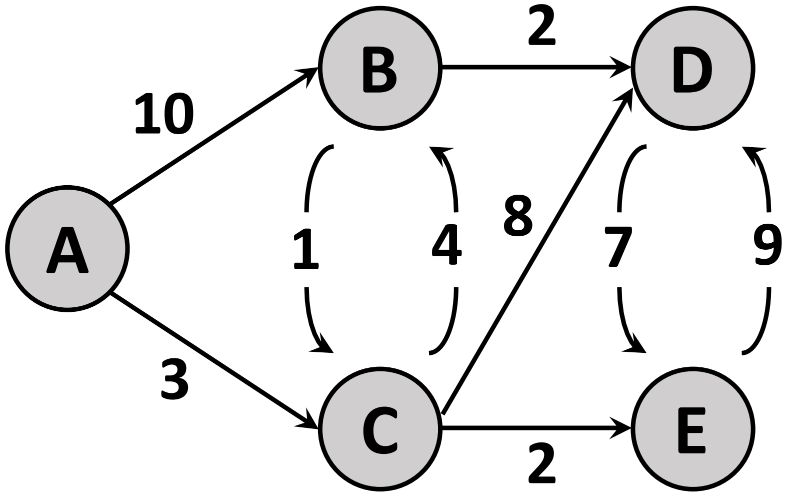
Vertex	1	2	3	4	5	6	7	8
Parent	-	1	1	2	2	5	3	3

Dijkstra's Algorithm

- **Dijkstra's Shortest Path Algorithm** is a modification of BFS for non-negatively weighted graphs
- **Informal Version:**
  - **Maintain a set S of explored nodes**
  - **Maintain an upper bound on distance**

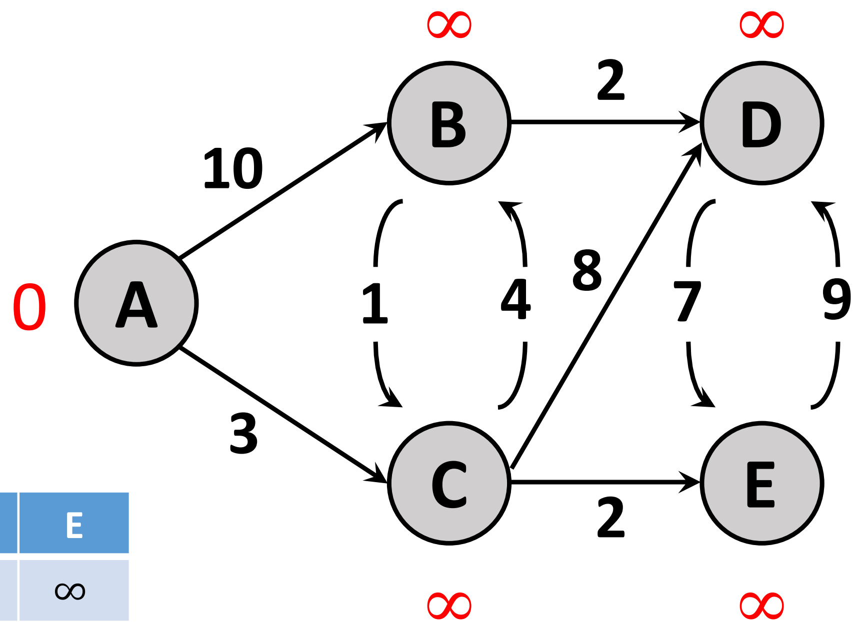
    s is the source
    - If u is explored, then we know $d(u)$ (Key Invariant)
    - If u is explored, and (u, v) is an edge, then we know $d(v) \leq d(u) + \ell(u, v)$
  - **Explore the "closest" unexplored node**
  - **Repeat until we're done**

Dijkstra’s Algorithm: Demo



Dijkstra's Algorithm: Demo

Initialize

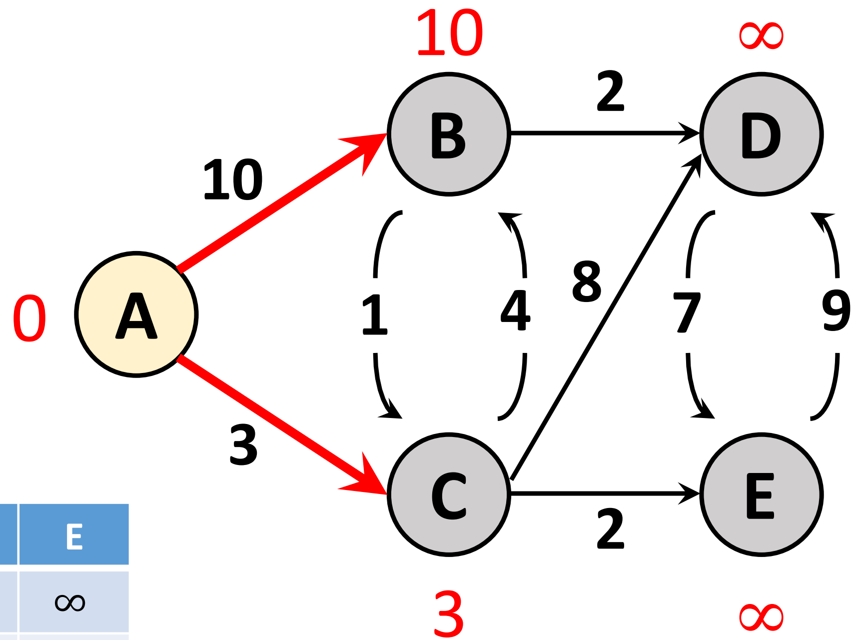


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$$S = \{\}$$

Dijkstra's Algorithm: Demo

Explore A

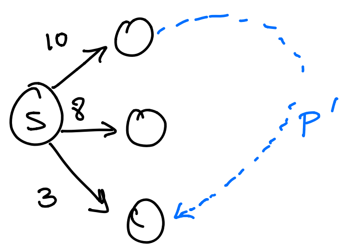


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞

$$S = \{A\}$$

- First node we explore is s

- Second node we explore is just the neighbor of s w/ the smallest distance

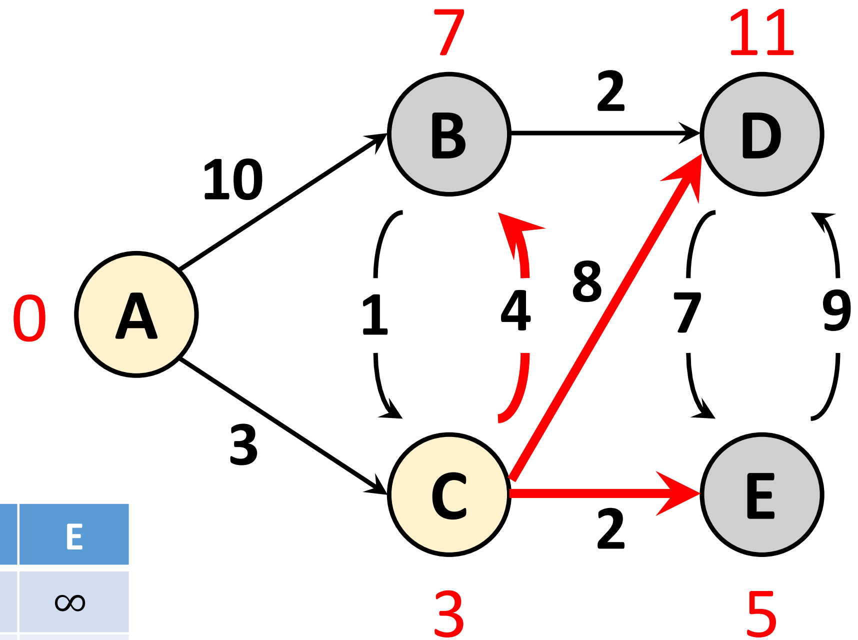


All edge lengths are ≥ 0

$\Rightarrow \ell(p') \geq 10$

Dijkstra's Algorithm: Demo

Explore C

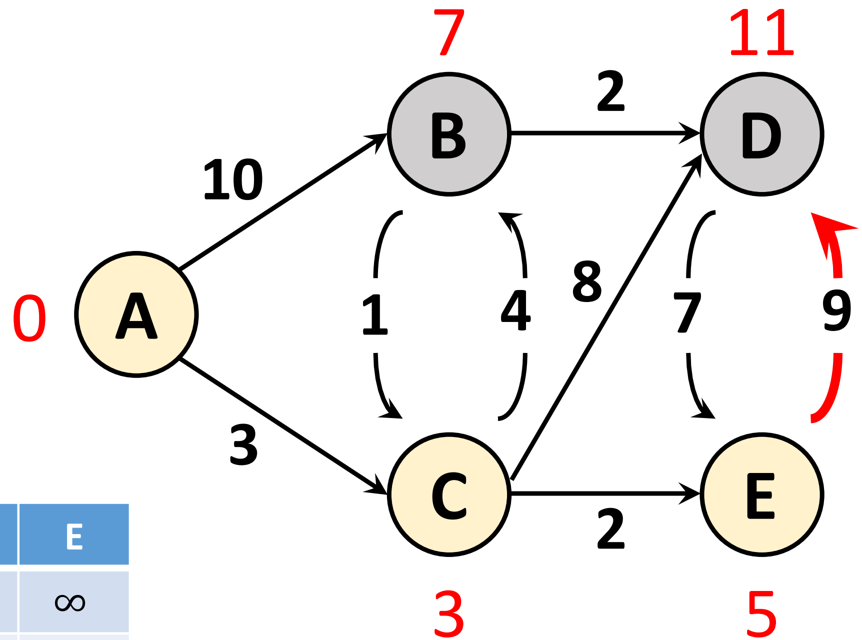


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞
$d_2(u)$	**0**	**7**	**3**	**11**	**5**

$$S = \{A, C\}$$

Dijkstra’s Algorithm: Demo

Explore E

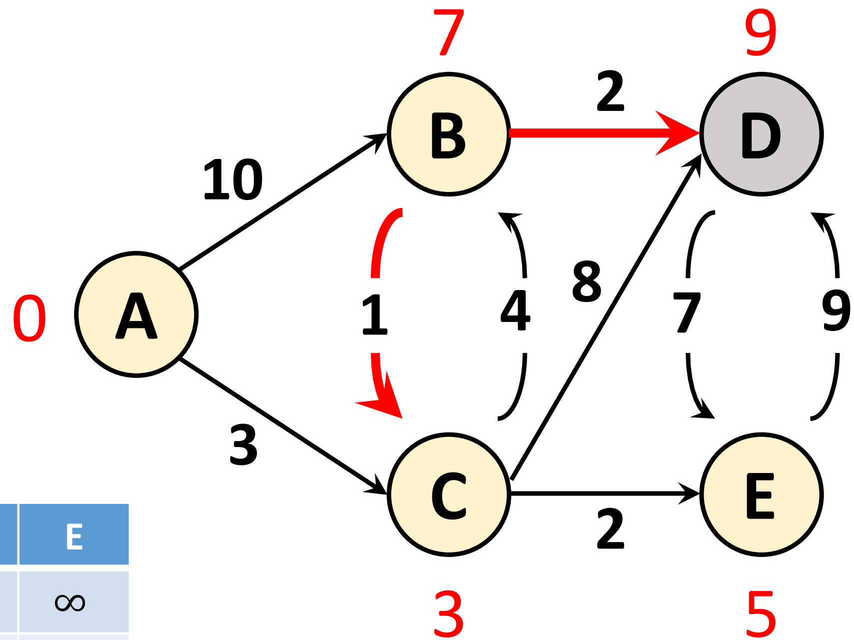


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞
$d_2(u)$	**0**	**7**	**3**	**11**	**5**
$d_3(u)$	**0**	**7**	**3**	**11**	**5**

$$S = \{A, C, E\}$$

Dijkstra's Algorithm: Demo

Explore B

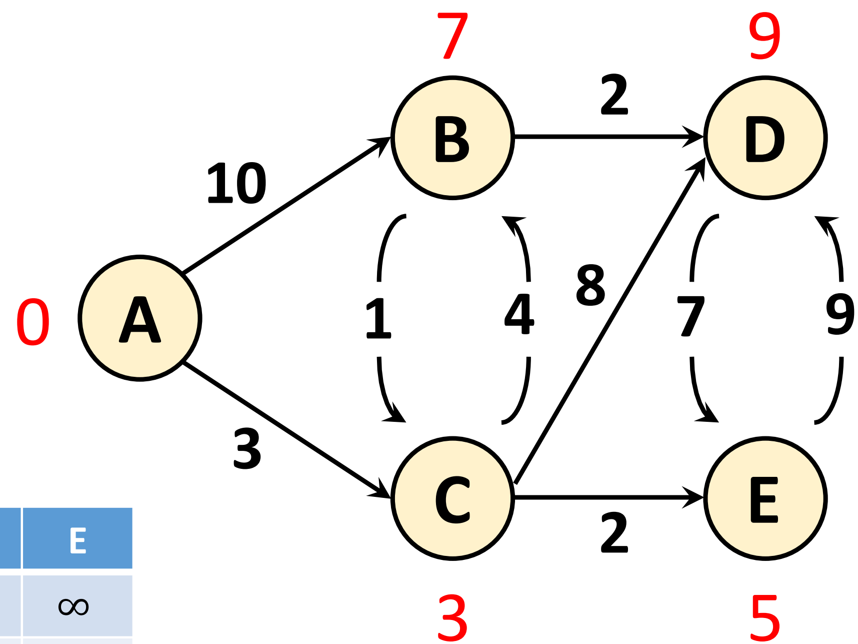


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$S = \{A, C, E, B\}$$

Dijkstra's Algorithm: Demo

Don't need to explore D

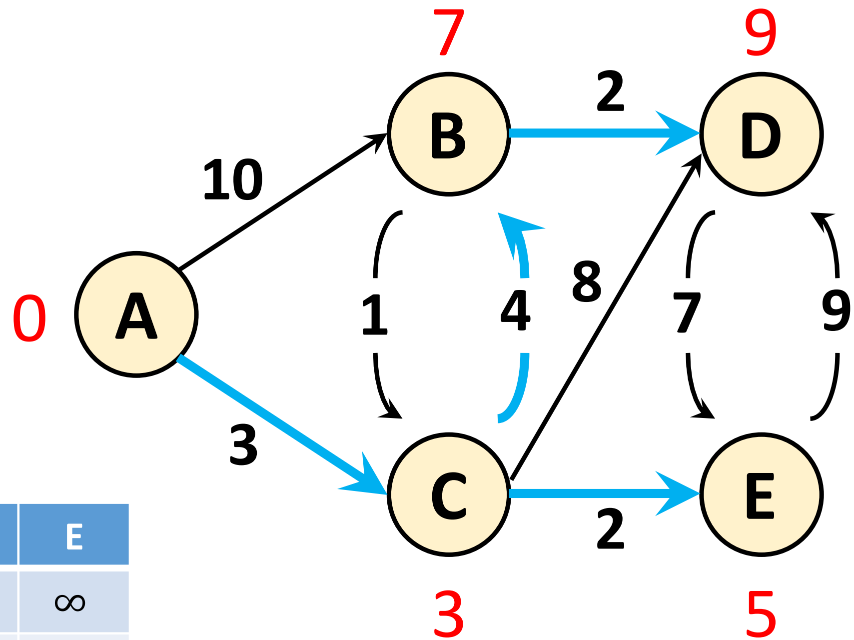


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$S = \{A, C, E, B, D\}$$

Dijkstra's Algorithm: Demo

Maintain parent pointers so we can find the shortest paths



	A	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