Lecture 07
BFS / DFS / Shortest paths problem

Spring 2023

Zhihua Jiang

Graphs I: Breadth First Search

Lecture Overview

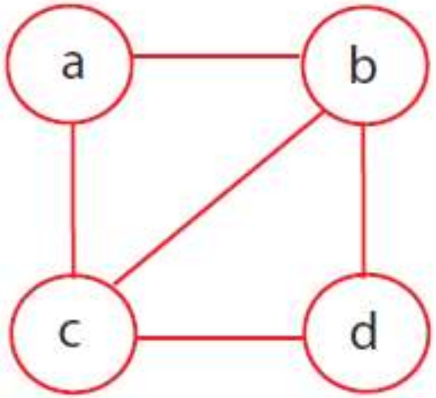
- Applications of Graph Search
- Graph Representations
- Breadth-First Search

Recall:

Graph $G = (V, E)$

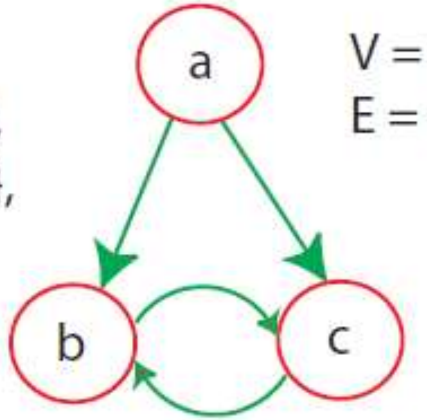
- V = set of vertices (arbitrary labels)
- E = set of edges i.e. vertex pairs (v, w)
  - ordered pair $\Longrightarrow$ directed edge of graph
  - unordered pair $\Longrightarrow$ undirected

e.g.



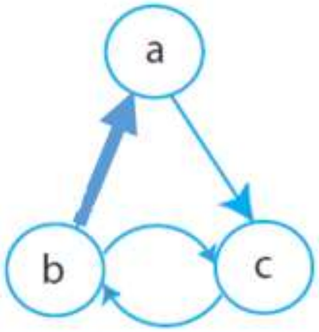
UNDIRECTED

V = {a,b,c,d}
E = {{a,b},{a,c},
{b,c},{b,d},
{c,d}}



DIRECTED

V = {a,b,c}
E = {(a,c),(b,c),
(c,b),(b,a)}



Graph Representations: (data structures)

Adjacency lists:

Array *Adj* of $|V|$ linked lists

- for each vertex $u \in V, Adj[u]$ stores u's neighbors, i.e., $\{v \in V \mid (u, v) \in E\}$. (u, v) are just outgoing edges if directed. (See Fig. 2 for an example.)

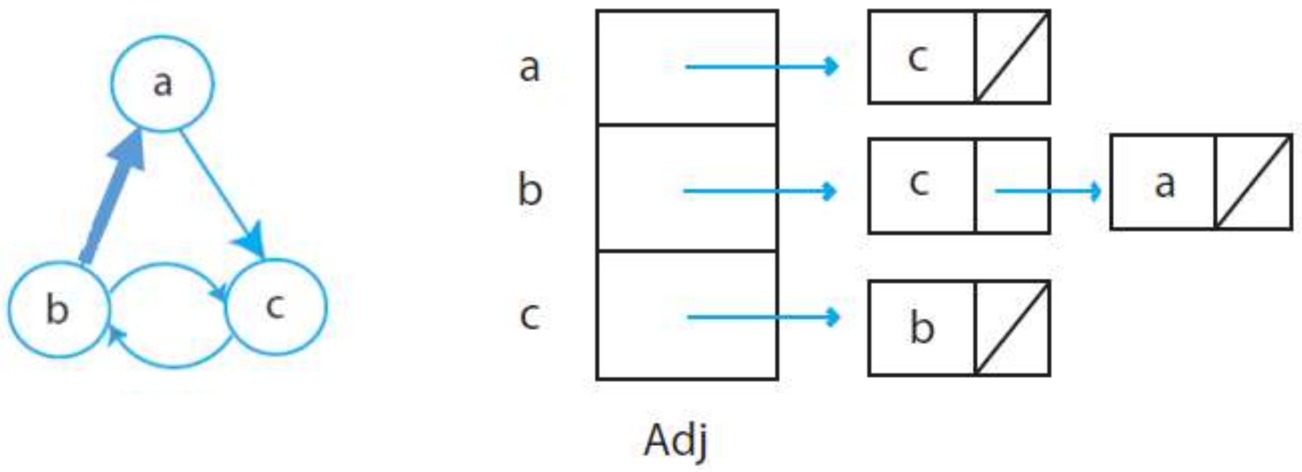
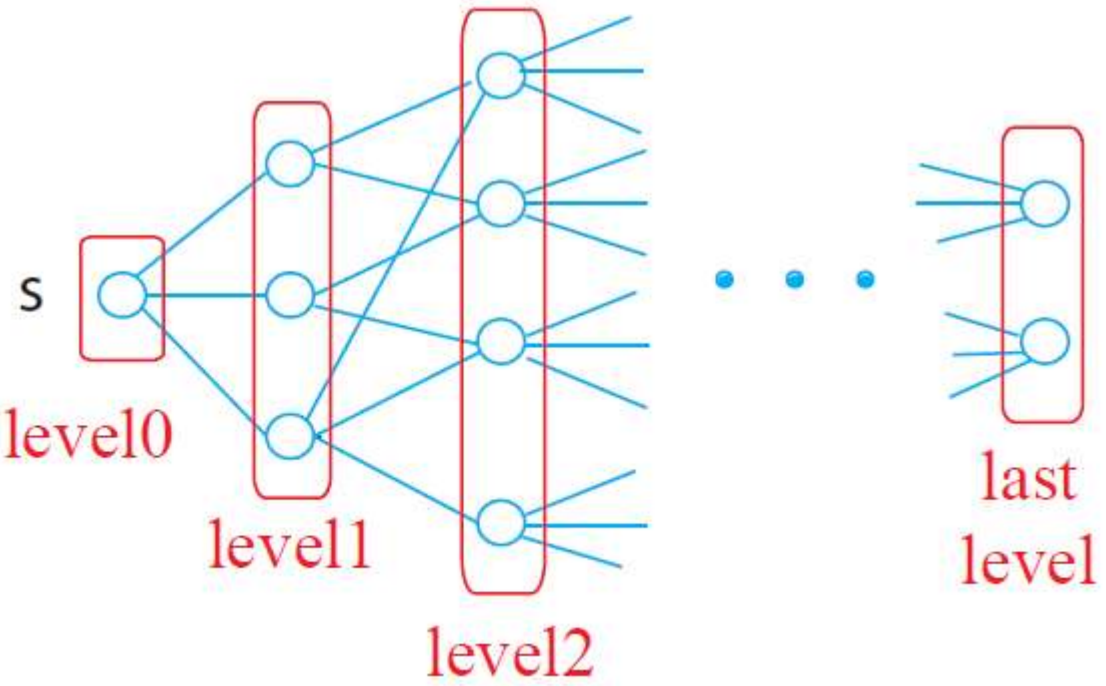


Figure 2: Adjacency List Representation: Space $\Theta(V + E)$

Breadth-First Search

Explore graph level by level from s

- level $0 = \{s\}$
- level $i =$ vertices reachable by path of i edges but not fewer



- build level $i > 0$ from level $i-1$ by trying all outgoing edges, but ignoring vertices from previous levels

Breadth-First-Search Algorithm

```
BFS (V,Adj,s):
    level = { s: 0 }
    parent = {s : None }
    i = 1
    frontier = [s]                      # previous level, i − 1
    while frontier:
        next = [ ]                      # next level, i
        for u in frontier:
            for v in Adj [u]:
                if v not in level:      # not yet seen
                    level[v] = i        ♯ = level[u] + 1
                    parent[v] = u
                    next.append(v)
        frontier = next
        i + =1
```

Analysis:

- vertex V enters next (& then frontier) only once (because level$[v]$ then set)

  base case: $v = s$

- $\implies$ Adj$[v]$ looped through only once

$$\text{time} = \sum_{v \in V} |Adj[v]| = \begin{cases} |E| \text{ for directed graphs} \\ 2|E| \text{ for undirected graphs} \end{cases}$$

- $\implies O(E)$ time

- $O(V + E)$ ("LINEAR TIME") to also list vertices unreachable from v (those still not assigned level)

Shortest Paths:

- for every vertex v, fewest edges to get from s to v is

$$\begin{cases} \text{level}[v] & \text{if } v \text{ assigned level} \\ \infty & \text{else (no path)} \end{cases}$$

- parent pointers form shortest-path tree = union of such a shortest path for each v
$\implies$ to find shortest path, take v, parent[v], parent[parent[v]], etc., until s (or None)

Graphs II: Depth-First Search

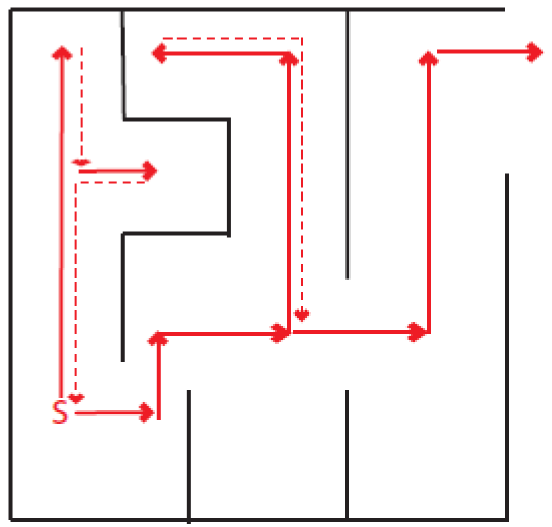
Lecture Overview

- Depth-First Search
- Edge Classification
- Cycle Testing
- Topological Sort

Depth-First Search (DFS)

This is like exploring a maze.

- follow path until you get stuck
- backtrack along breadcrumbs until reach unexplored neighbor
- recursively explore
- careful not to repeat a vertex



```
parent = {s: None}

DFS-visit (V, Adj, s):
        for v in Adj [s]:
            if v not in parent:
                parent [v] = s
                DFS-visit (V, Adj, v)

DFS (V, Adj)
    parent = { }
    for s in V:
        if s not in parent:
            parent [s] = None
            DFS-visit (V, Adj, s)
```

start v

finish v

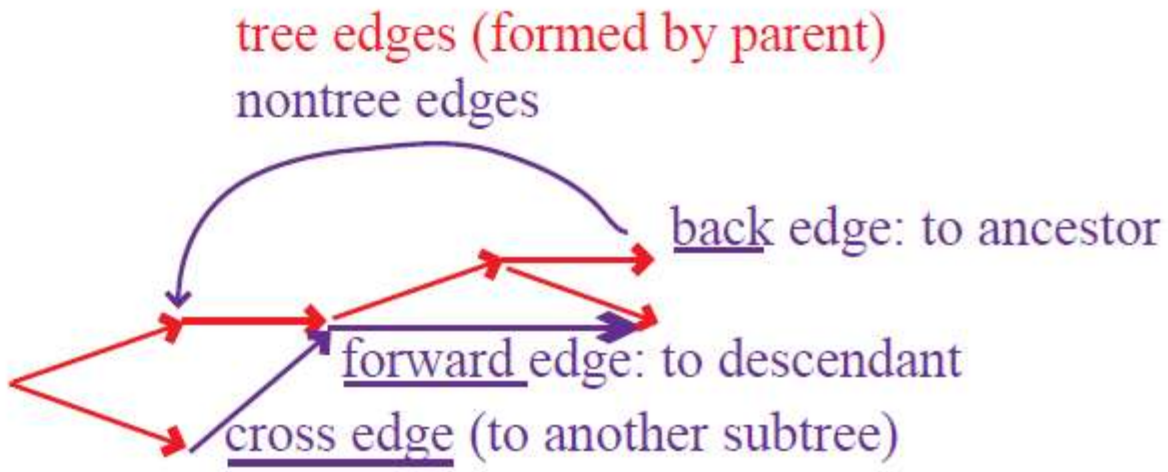
search from start vertex s (only see stuff reachable from s)

explore entire graph

Analysis

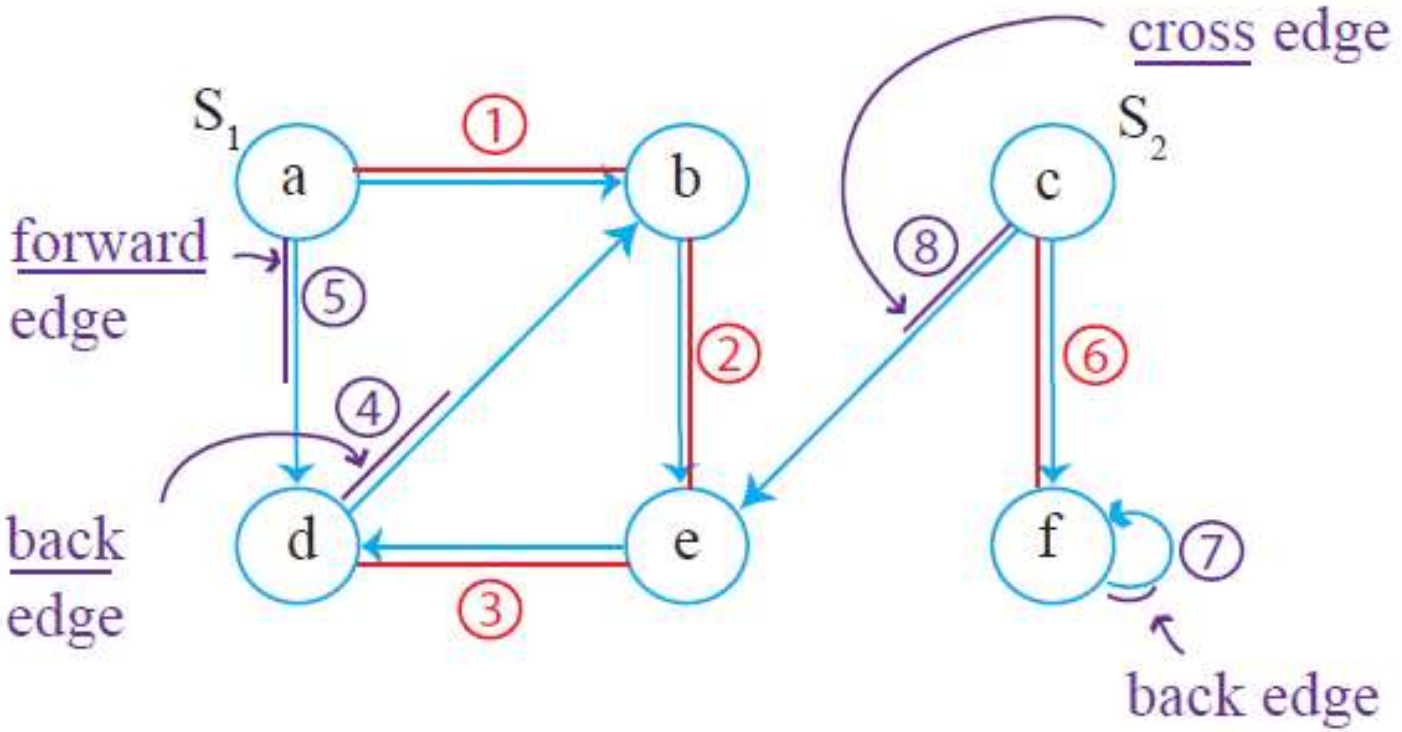
- DFS-visit gets called with a vertex s only once (because then parent$[s]$ set)
  $\implies$ time in DFS-visit $= \sum_{s \in V} |\text{Adj}[s]| = O(E)$
- DFS outer loop adds just $O(V)$
  $\implies O(V + E)$ time (linear time)

Edge Classification



- to compute this classification (back or not), mark nodes for duration they are “on the stack”
- only tree and back edges in undirected graph

Example



cross edge
S_1
S_2
a
b
c
d
e
f
1
2
3
4
5
6
7
8
forward edge
back edge
back edge

Cycle Detection

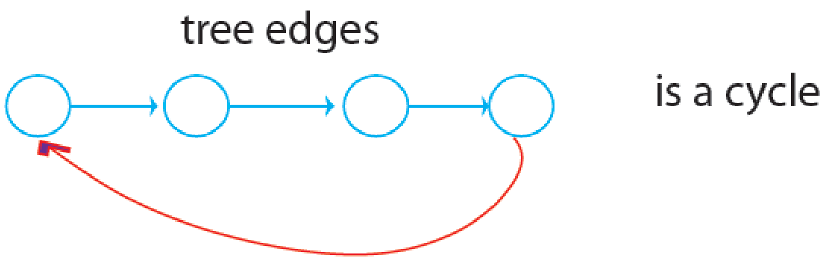
Graph G has a cycle $\Leftrightarrow$ DFS has a back edge

- before visit to v_i finishes,
  will visit v_{i+1} (& finish):
  will consider edge (v_i, v_{i+1})
  $\Longrightarrow$ visit v_{i+1} now or already did

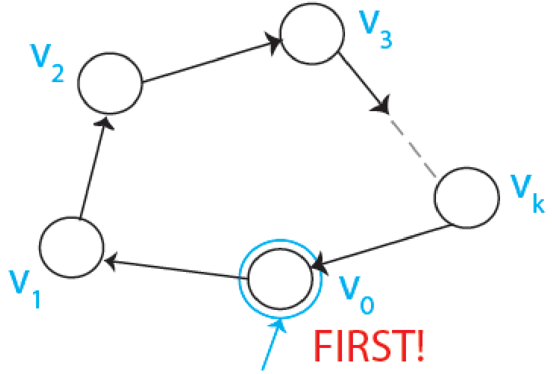
- $\Longrightarrow$ before visit to v_0 finishes,
  will visit v_k (& didn't before)

- $\Longrightarrow$ before visit to v_k (or v_0) finishes,
  will see (v_k, v_0) as back edge

(<=)



(=>) consider first visit to cycle:



Job scheduling

Given Directed Acylic Graph (DAG), where vertices represent tasks & edges represent dependencies, order tasks without violating dependencies

Source = vertex with no incoming edges
= schedulable at beginning

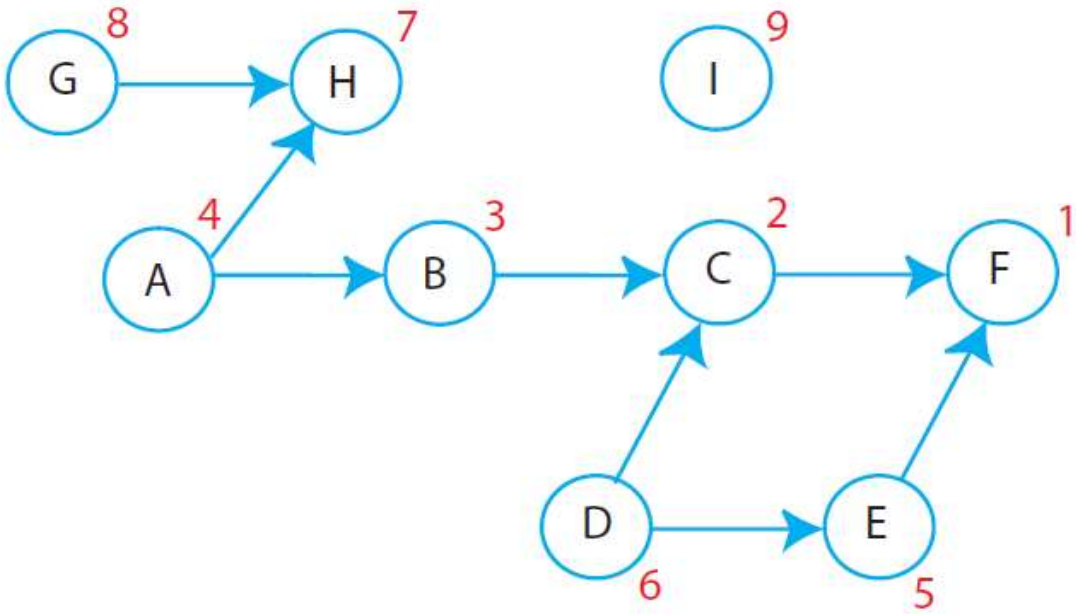


Figure 6: Dependence Graph: DFS Finishing Times

Topological Sort

Reverse of DFS finishing times (time at which DFS-Visit(v) finishes)

```
DFS-Visit(v)
    ...
    order.append(v)

order.reverse()
```

Correctness

For any edge (u, v) — u ordered before v, i.e., v finished before u

Shortest Paths I: Intro

Lecture Overview

- Weighted Graphs
- General Approach
- Negative Edges
- Optimal Substructure

Motivation:

Shortest way to drive from A to B Google maps “get directions”

Formulation: Problem on a weighted graph $G(V, E) \quad W : E \to \Re$

Two algorithms: Dijkstra $O(V \lg V + E)$ assumes non-negative edge weights
Bellman Ford $O(VE)$ is a general algorithm

path $p = < v_0, v_1, \ldots v_k >$

$$(v_i, v_{i+1}) \in E \quad \text{for} \quad 0 \leq i < k$$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Weighted Graphs:

Notation:

$v_0 \xrightarrow{p} v_k$ means p is a path from v_0 to v_k. (v_0) is a path from v_0 to v_0 of weight 0.

Definition:

Shortest path weight from u to v as

$$\delta(u,v) = \begin{cases} \min \left\{ w(p) : u \xrightarrow{p} v \right\} & \text{if } \exists \text{ any such path} \\ \infty & \text{otherwise} \quad (v \text{ unreachable from } u) \end{cases}$$

Single Source Shortest Paths:

Given $G = (V, E), w$ and a source vertex S, find $\delta(S, V)$ [and the best path] from S to each $v \in V$.

Data structures:

$$
\begin{aligned}
d[v] &= \text{value inside circle} \\
&= \left\{ \begin{array}{ll} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{array} \right\} \Longleftarrow \quad \text{initially} \\
&= \delta(s, v) \Longleftarrow \quad \text{at end} \\
d[v] &\geq \delta(s, v) \quad \text{at all times}
\end{aligned}
$$

$d[v]$ decreases as we find better paths to v

$\Pi[v]$ = predecessor on best path to v, $\Pi[s] = \text{NIL}$

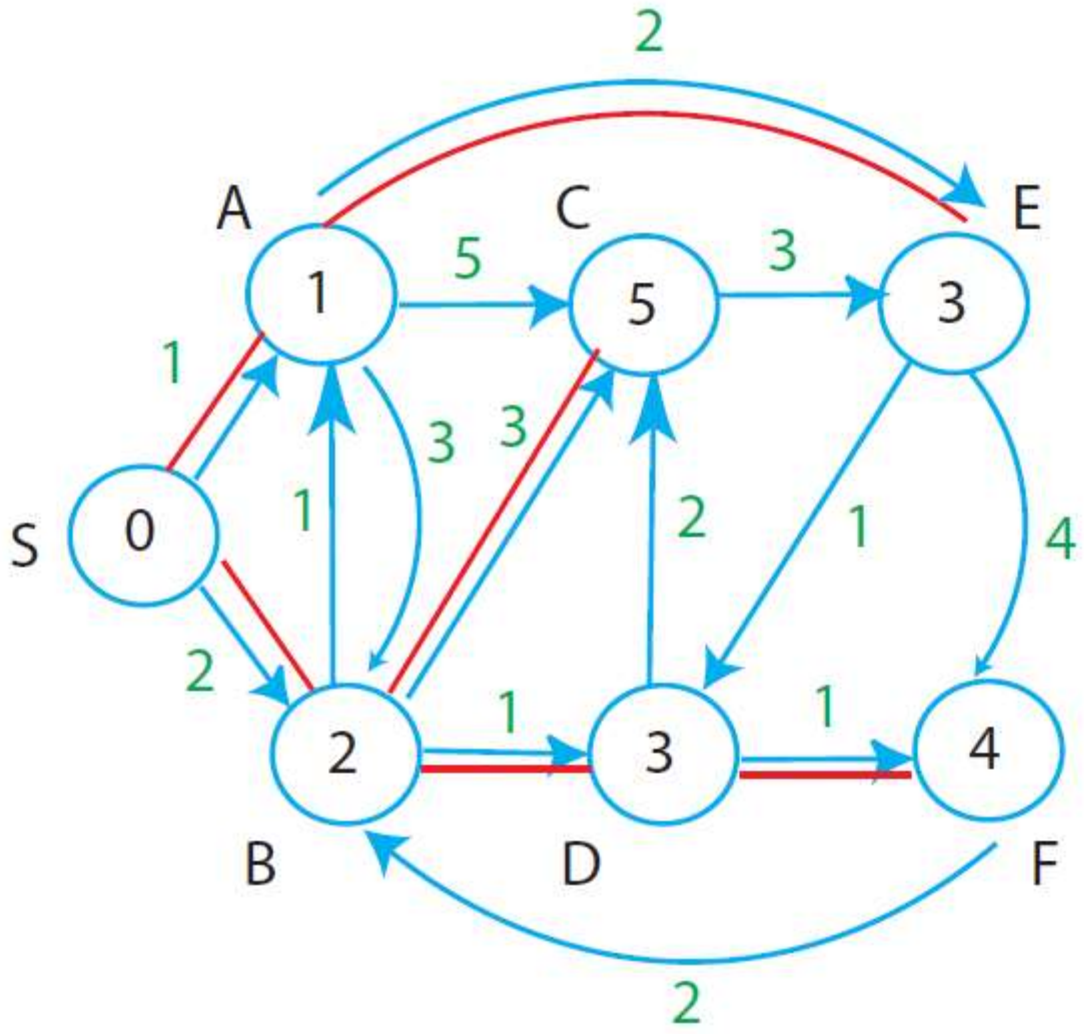
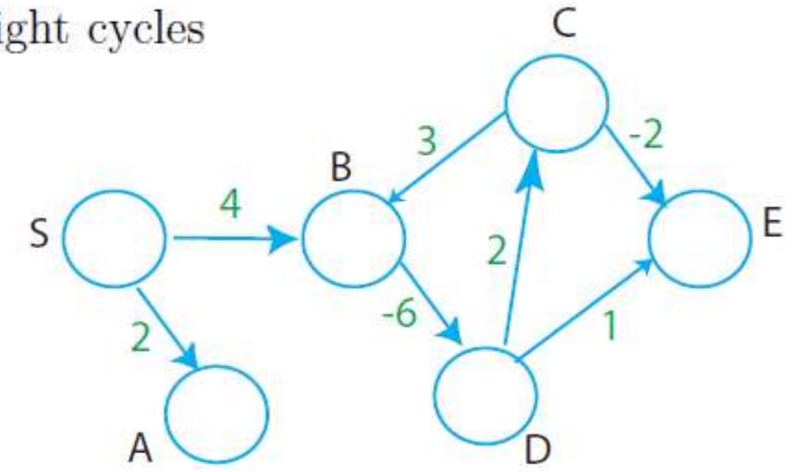


Figure 1: Shortest Path Example: Bold edges give predecessor Π relationships

Negative-Weight Edges:

- Natural in some applications (e.g., logarithms used for weights)
- Some algorithms disallow negative weight edges (e.g., Dijkstra)
- If you have negative weight edges, you might also have negative weight cycles $\Longrightarrow$ may make certain shortest paths undefined!



$B \to D \to C \to B$ (origin) has weight $-6+2+3=-1<0$!

Shortest path $S \longrightarrow C$ (or B, D, E) is undefined. Can go around $B \to D \to C$ as many times as you like

Shortest path $S \longrightarrow A$ is defined and has weight 2

If negative weight edges are present, s.p. algorithm should find negative weight cycles (e.g., Bellman Ford)

General structure of S.P. Algorithms (no negative cycles)

Initialize:	for $v \in V$: $\begin{array}{lcl} d[v] & \leftarrow & \infty \\ \Pi[v] & \leftarrow & \text{NIL} \end{array}$
	$d[S] \leftarrow 0$
Main:	repeat
	select edge (u, v) [somehow]
"Relax" edge (u, v)	$\left[\begin{array}{l} \text{if } d[v] > d[u] + w(u, v) : \\ \quad d[v] \leftarrow d[u] + w(u, v) \\ \quad \pi[v] \leftarrow u \end{array} \right.$
	until all edges have $d[v] \leq d[u] + w(u, v)$

Optimal Substructure:

Theorem: Subpaths of shortest paths are shortest paths

Let $p = < v_0, v_1, \ldots v_k >$ be a shortest path

Let $p_{ij} = < v_i, v_{i+1}, \ldots v_j > \quad 0 \le i \le j \le k$

Then p_{ij} is a shortest path.

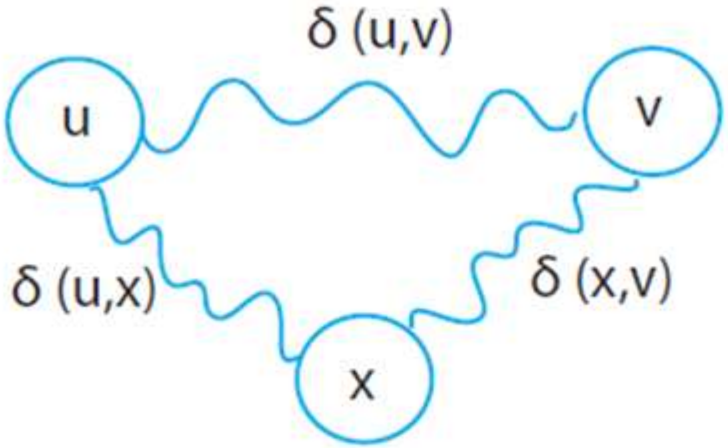
Proof: $p = \begin{array}{ccccccc} & p_{0,i} & & p_{ij} & & p_{jk} & \\ v_0 & \rightarrow & v_i & \rightarrow & v_j & \rightarrow & v_k \\ & & & \rightarrow & & & \\ & & & p'_{ij} & & & \end{array}$

If p'_{ij} is shorter than p_{ij}, cut out p_{ij} and replace with p'_{ij}; result is shorter than p.
Contradiction.

Triangle Inequality:

Theorem: For all $u, v, x \in X$, we have

$$\delta\left(u,v\right) \leq \delta\left(u,x\right) + \delta\left(x,v\right)$$



Shortest Paths II - Dijkstra

Lecture Overview

- Review
- Shortest paths in DAGs
- Shortest paths in graphs without negative edges
- Dijkstra's Algorithm

Relaxation is Safe

Lemma: The relaxation algorithm maintains the invariant that $d[v] \geq \delta(s, v)$ for all $v \in V$.

Proof: By induction on the number of steps.

Consider $RELAX(u, v, w)$. By induction $d[u] \geq \delta(s, u)$. By the triangle inequality, $\delta(s, v) \leq \delta(s, u) + \delta(u, v)$. This means that $\delta(s, v) \leq d[u] + w(u, v)$, since $d[u] \geq \delta(s, u)$ and $w(u, v) \geq \delta(u, v)$. So setting $d[v] = d[u] + w(u, v)$ is safe. $\square$

DAGs:

Can't have negative cycles because there are no cycles!

1. Topologically sort the DAG. Path from u to v implies that u is before v in the linear ordering.
2. One pass over vertices in topologically sorted order relaxing each edge that leaves each vertex.
   $\Theta(V + E)$ time

   Can deal with negative edges

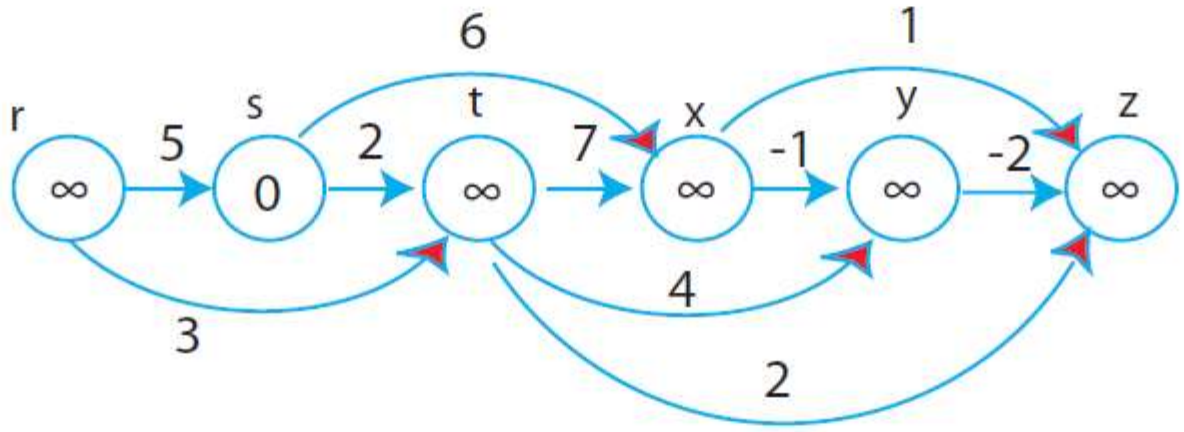
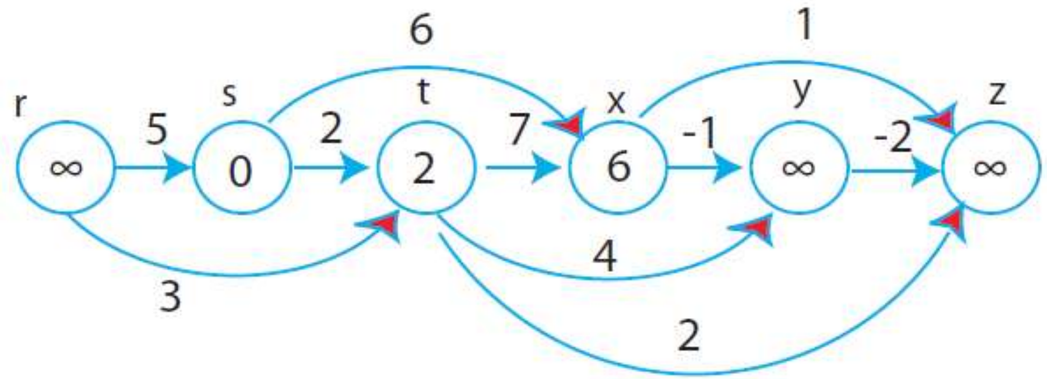


Figure 1: Shortest Path using Topological Sort.

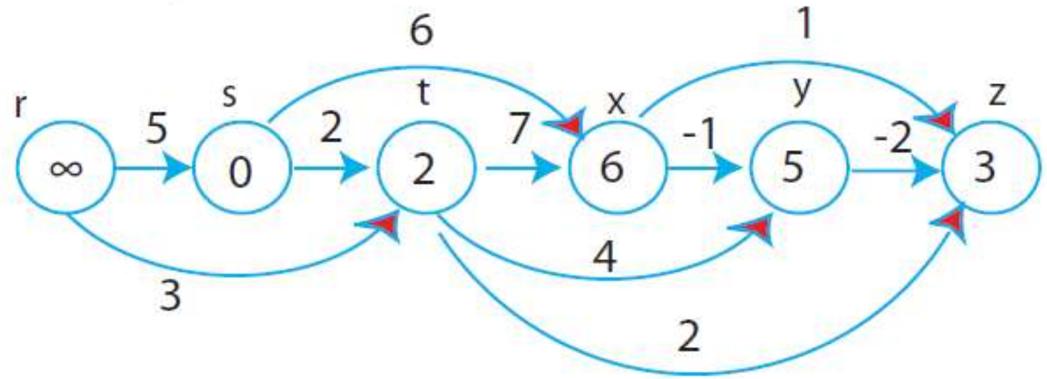
Vertices sorted left to right in topological order

Process r: stays ∞. All vertices to the left of s will be ∞ by definition

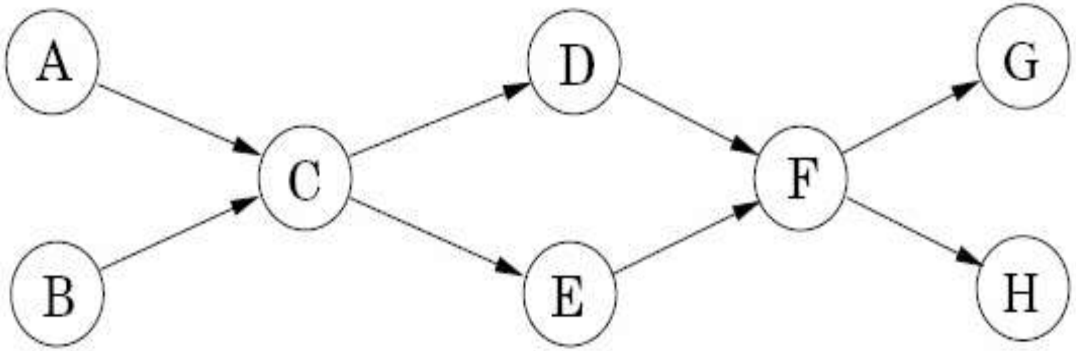
Process s: $t : \infty \rightarrow 2$ $\quad$ $x : \infty \rightarrow 6$



process t, x, y



Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.



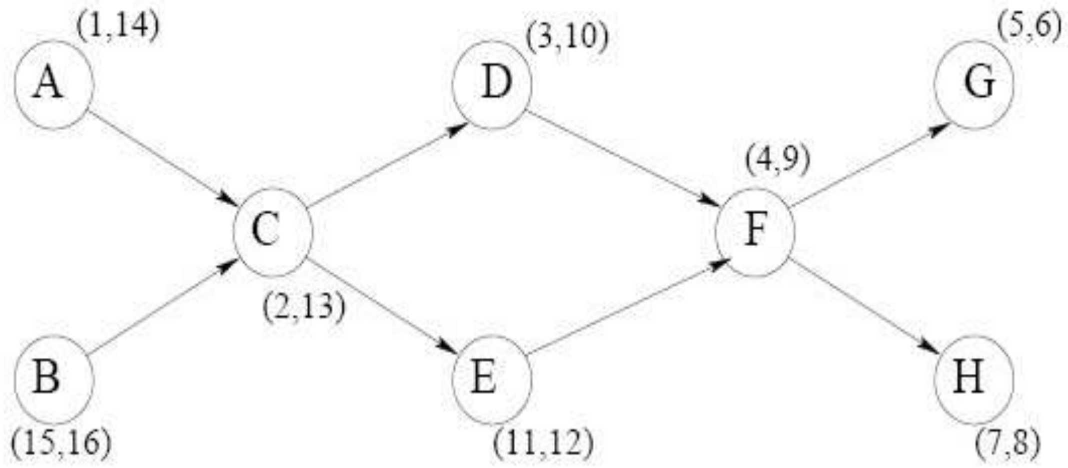
(a) Indicate the `pre` and `post` numbers of the nodes.

(b) What are the sources and sinks of the graph?

(c) What topological ordering is found by the algorithm?

(d) How many topological orderings does this graph have?

(a) The figure below shows the `pre` and `post` times in parentheses.



(b) The vertices A, B are sources and G, H are sinks.

(c) Since the algorithm outputs vertices in decreasing order of `post` numbers, the ordering given is B, A, C, E, D, F, H, G.

(d) Any ordering of the graph must be of the form $\{A, B\}, C, \{D, E\}, F, \{G, H\}$, where $\{A, B\}$ indicates A and B may be in any order within these two places. Hence the total number of orderings is $2^3 = 8$.

```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

Dijkstra's Algorithm

For each edge $(u, v) \epsilon E$, assume $w(u, v) \geq 0$, maintain a set S of vertices whose final shortest path weights have been determined. Repeatedly select $u \epsilon V - S$ with minimum shortest path estimate, add u to S, relax all edges out of u.

Pseudo-code

Dijkstra (G, W, s) //uses priority queue Q

  Initialize (G, s)

  $S \leftarrow \phi$

  $Q \leftarrow V[G]$ //Insert into Q

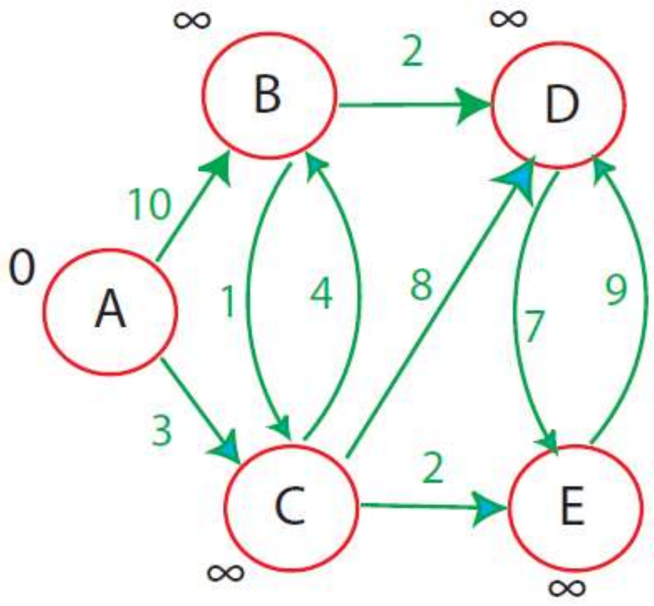
  while $Q \neq \phi$

    do $u \leftarrow$ EXTRACT-MIN(Q) //deletes u from Q

    $S = S \cup \{u\}$

    for each vertex $v \epsilon$ Adj$[u]$

      do RELAX (u, v, w) $\leftarrow$ this is an implicit DECREASE_KEY operation



S = { }	{ A	B	C	D	E }	=	Q
S = { A }	0	∞	∞	∞	∞		
S = { A, C }	0	10	3	∞	∞	←	after relaxing edges from A
S = { A, C }	0	7	3	11	5	←	after relaxing edges from C
S = { A, C, E }	0	7	3	11	5		
S = { A, C , E, B}	0	7	3	9	5	←	after relaxing edges from B

Strategy: Dijkstra is a greedy algorithm: choose closest vertex in $V - S$ to add to set S.

Correctness: We know relaxation is safe. The key observation is that each time a vertex u is added to set S, we have $d[u] = \delta(s, u)$.

Dijkstra Complexity

$\Theta(v)$ inserts into priority queue
$\Theta(v)$ EXTRACT_MIN operations
$\Theta(E)$ DECREASE_KEY operations

Priority queue implementations

- The running time of Dijkstra's algorithm depends heavily on the priority queue implementation.

Implementation	deletemin	insert/ decreasekey	$\lvert V\rvert \times$ deletemin + $(\lvert V\rvert + \lvert E\rvert) \times$ insert
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O((\lvert V\rvert \cdot d + \lvert E\rvert)\frac{\log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

d-ary heap

- Identical to a binary heap, except that nodes have d children.
- The height of the tree: $\Theta(\log_d |V|) = \Theta(\log |V| / \log d)$
- Insert/DecreaseKey: speeded up, $\Theta(\log_d |V|)$
- DeleteMin: take a longer time, $\Theta(d \log_d |V|)$
- Total: $|V|$DeleteMin+$|V|$Insert+$|E|$DecreaseKey

$$= \Theta(|V| \cdot d \log_d |V| + (|V| + |E|) \cdot \log_d |V|)$$

$$= \Theta((|V| d + |E|) \log_d |V|)$$

Shortest Paths III: Bellman-Ford

Lecture Overview

- Review: Notation
- Generic S.P. Algorithm
- Bellman-Ford Algorithm
  - Analysis
  - Correctness

Bellman-Ford(G,W,s)

```
Initialize ()
for i = 1 to |V| − 1
    for each edge (u, v) ∈ E:
        Relax(u, v)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
        then report a negative-weight cycle exists
```

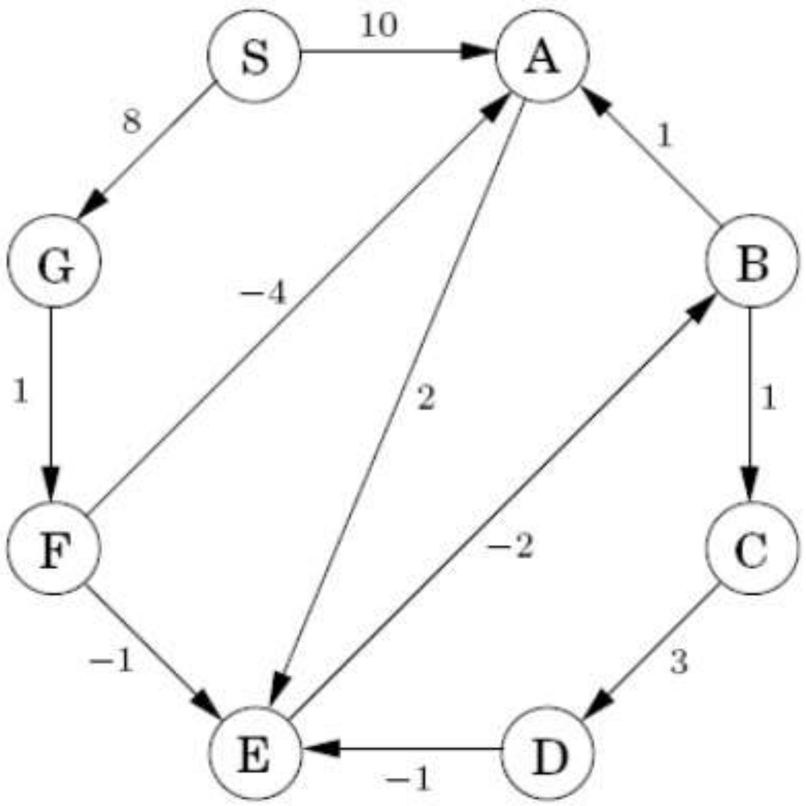
At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.

Corollary

If a value $d[v]$ fails to converge after $|V| - 1$ passes, there exists a negative-weight cycle reachable from s.

Theorem:

If $G = (V, E)$ contains no negative weight cycles, then after Bellman-Ford executes $d[v] = \delta(s, v)$ for all $v \in V$.



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Shortest Paths IV - Speeding up Dijkstra

Lecture Overview

- Single-source single-target Dijkstra
- Bidirectional search
- Goal directed search - potentials and landmarks

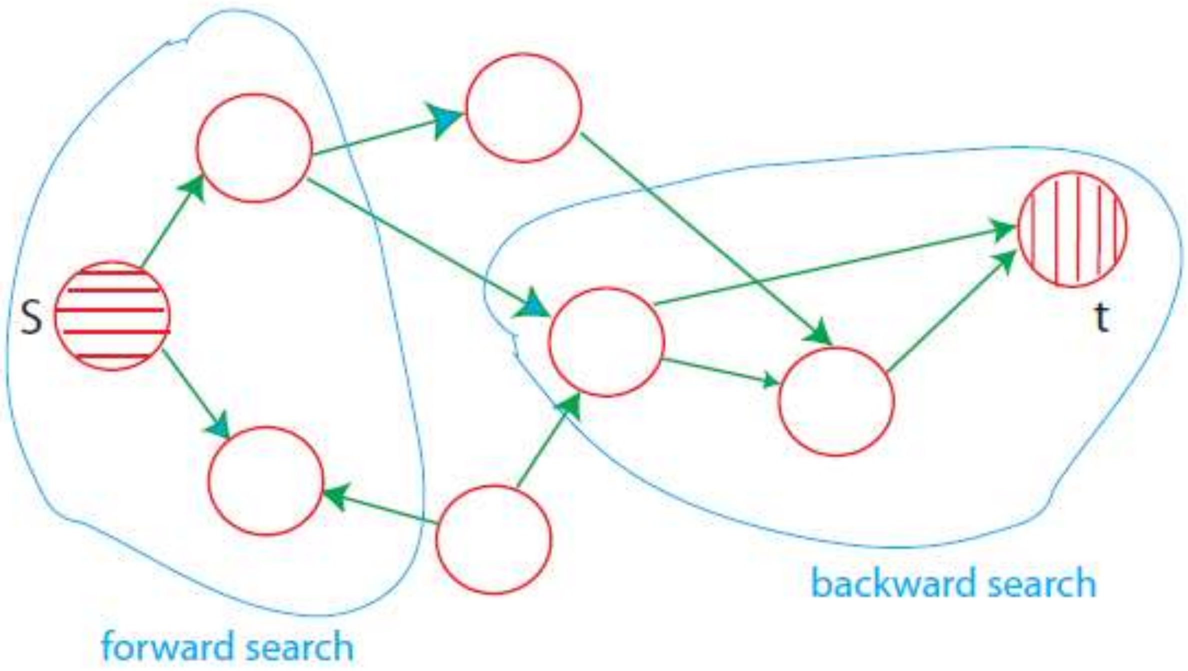
DIJKSTRA single-source, single-target

Initialize()
$Q \leftarrow V[G]$
while $Q \neq \phi$
    do $u \leftarrow$ EXTRACT_MIN(Q) (stop if $u = t$!)
    for each vertex $v \epsilon$ Adj$[u]$
        do RELAX(u, v, w)

Observation: If only shortest path from s to t is required, stop when t is removed from Q, i.e., when $u = t$

Bi-Directional Search

Note: Speedup techniques covered here do not change worst-case behavior, but reduce the number of visited vertices in practice.



Bi-D Search

Alternate forward search from s
backward search from t
(follow edges backward)
$d_f(u)$ distances for forward search
$d_b(u)$ distances for backward search

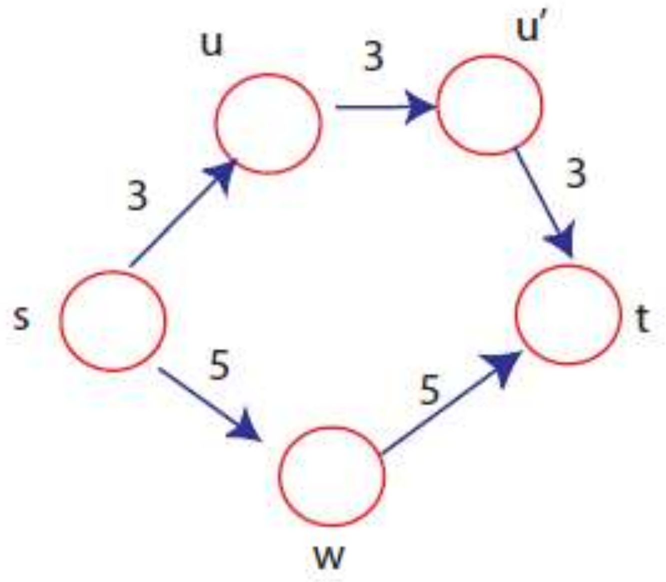
Algorithm terminates when some vertex w has been processed, i.e., deleted from the queue of both searches, Q_f and Q_b
Subtlety: After search terminates, find node x with $\boxed{\text{minimum value of } d_f(x) + d_b(x)}$.
x may not be the vertex w that caused termination as in example to the left!
Find shortest path from s to x using Π_f and shortest path backwards from t to x using Π_b. *Note:* x will have been deleted from either Q_f or Q_b or both.

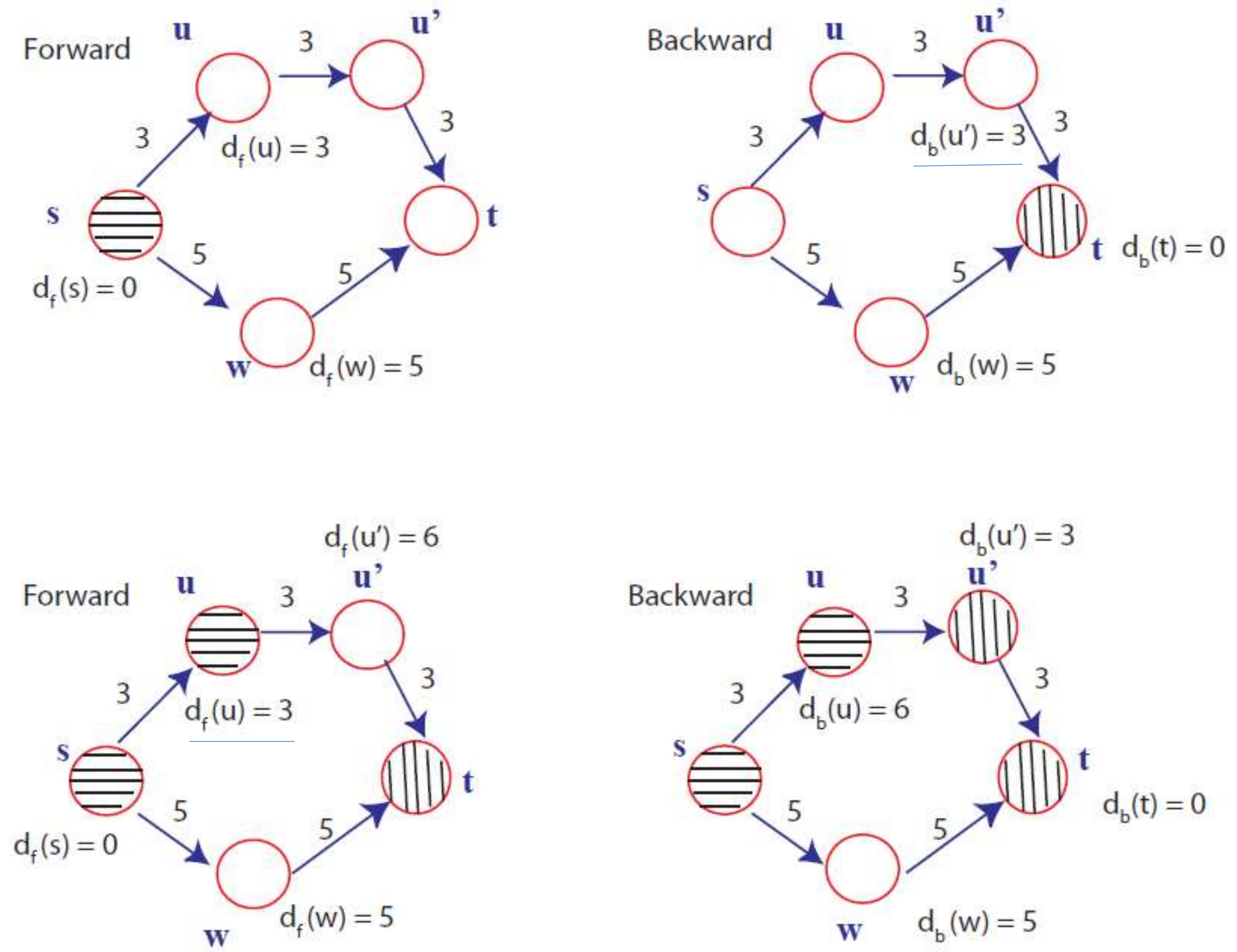
Minimum value for $d_f(x) + d_b(x)$ over all vertices that have been processed in at least one search

$$d_f(u) + d_b(u) = 3 + 6 = 9$$

$$d_f(u') + d_b(u') = 6 + 3 = 9$$

$$d_f(w) + d_b(w) = 5 + 5 = 10$$




Forward
u
u'
s
t
w
3
3
3
5
5
$d_f(u) = 3$
$d_f(s) = 0$
$d_f(w) = 5$
Backward
u
u'
s
t
w
3
3
3
5
5
$d_b(u') = 3$
$d_b(t) = 0$
$d_b(w) = 5$
Forward
$d_f(u') = 6$
u
u'
s
t
w
3
3
3
5
5
$d_f(u) = 3$
$d_f(s) = 0$
$d_f(w) = 5$
Backward
$d_b(u') = 3$
u
u'
s
t
w
3
3
3
5
5
$d_b(u) = 6$
$d_b(t) = 0$
$d_b(w) = 5$

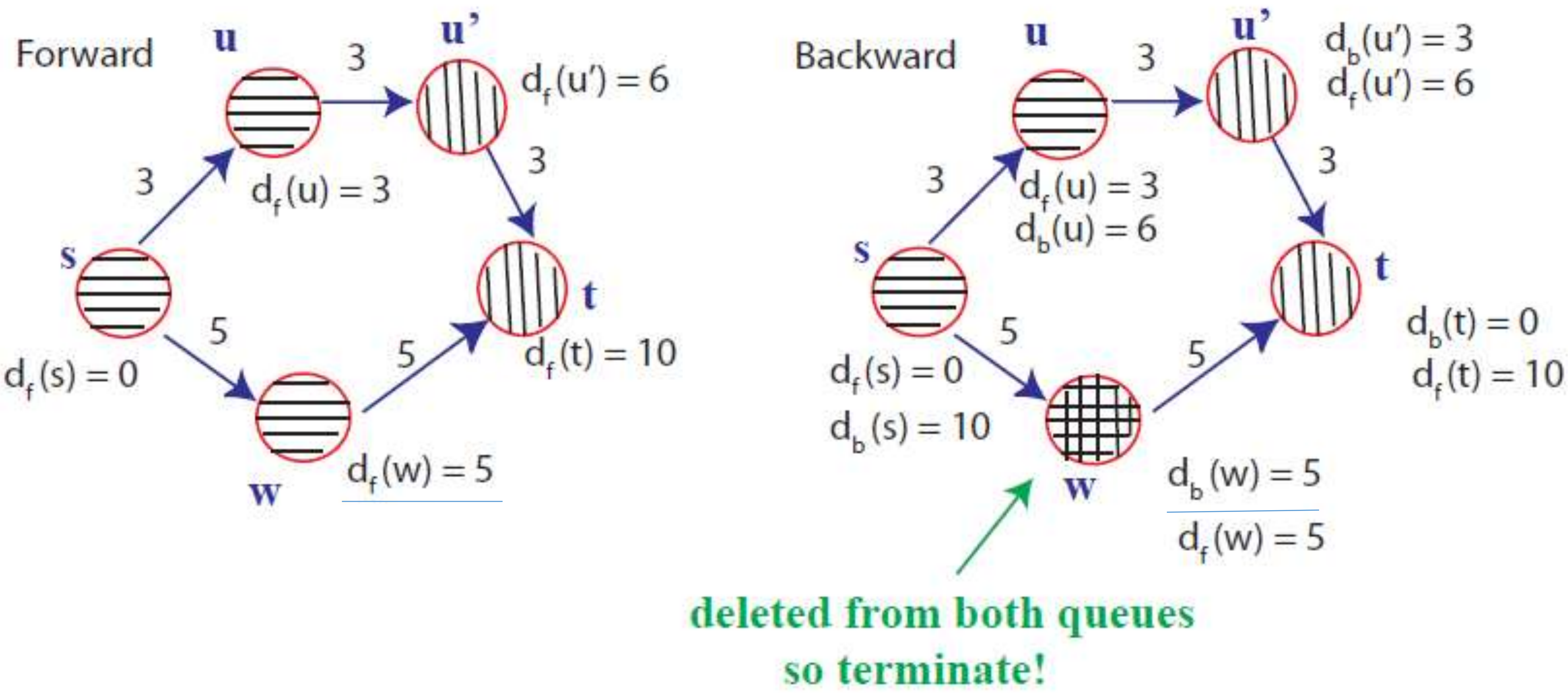


Figure 3: Forward and Backward Search and Termination.