Graph Algorithms (continued)
COMP4500/7500
Advanced Algorithms & Data Structures

August 21, 2019

Overview

- Admin/reminders
- Shortest paths:
  - Dijkstra's algorithm
  - Bellman-Ford algorithm
- Priority-first search

Shortest paths

A path p from v_1 to v_n is a sequence of vertices.

$$v_1 \to v_2 \to v_3 \to \ldots \to v_n$$

The total weight of the path, $weight(p)$, is the sum of the edges along the way.

$$weight(p) = \sum_{i=1}^{n-1} weight(v_i, v_{i+1})$$

The **distance** from v_1 to v_n is the minimum weight of all paths from v_1 to v_n.

A **shortest path** from v_1 to v_n is a path from v_1 to v_n of minimum weight.

Shortest paths

Types of shortest paths:

1. **Single-pair.** Given a pair (u, v), find a shortest path between them.
2. **Single-source.** Given a vertex v, find a shortest path to every other vertex. *This is our focus*.
3. **Single-destination.** Given a vertex v, find a shortest path *from* every other vertex.
4. **All-pairs.** Given a graph, find a shortest path between every pair of vertices.

SINGLESOURCESHORTESTPATHS(G, w, s): key ideas

Given graph G with weight-function w and source vertex s, for each vertex $v \in G.V$ we calculate:

$v.d$ it's distance from source vertex s

$v.\pi$ the predecessor to v on a shortest path from s to v

Steps:

- **Initialise** $s.d = 0$ and $v.d = \infty$ for all $v \in G.V - \{s\}$.
  - invariant: $\mathit{distance}(s, v) \leq v.d \leq \infty$
- **Initialise** $v.\pi = \mathit{NIL}$ for all $v \in G.V$.
  - invariant: $v.\pi$ is v's predecessor on the shortest path found so far
- ... we **relax** our estimates until they reach a solution ...

Edge relaxation: relax edge $(u, v) \in G.E$

RELAX(u, v, w):
updates $v.d$ and $v.\pi$ if v can be reached faster via intermediate vertex u.

RELAX(u, v, w)

```
1   if v.d > u.d + w(u, v)
2         v.d = u.d + w(u, v)
3         v.π = u
```

- preserves the distance and predecessor invariants:
  - invariant: $distance(s, v) \leq v.d \leq \infty$
  - invariant: $v.\pi$ is v's predecessor on the shortest path found so far
- does not change $v.d$ and $v.\pi$ if the shortest path to v has already been found

Edge relaxation

Let

$$p_n = v_1 \to v_2 \to v_3 \to \ldots \to v_n$$

be a shortest path from v_1 to v_n.

If edge (v_{i-1}, v_i) is relaxed *after* a shortest path to v_{i-1} is found, we will find a shortest path to v_i.

Graphs without negative weight edges

Let

$$p_n = v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_n$$

be any **shortest path** from v_1 to v_n in a weighted graph with **no negative weight edges**.

For all $v_i \in v_1, v_2, \ldots v_{n-1}$, we must have that the path prefix

$$p_i = v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_i$$

is a **shortest path** from v_1 to v_i, and $\mathit{weight}(p_i) \leq \mathit{weight}(p_n)$.

I.e. $\mathit{distance}(v_1, v_i) \leq \mathit{distance}(v_1, v_n)$.

Relaxation for graphs without negative weight edges

If we:

- **relax** each edge $(u, v) \in G.E$ once
- in order of u's distance from start vertex s,

then we will find all the shortest paths from s in G.

DIJKSTRA(G, w, s)

Solves the **single-source shortest paths problem** for weighted graphs **without negative weight edges**.

[AI in computer games use a sophisticated variant called A^*.]

DIJKSTRA(G, w, s): key idea

For each vertex $v \in G.V$ we calculate:

$v.d$ it's distance from source vertex s

$v.\pi$ the predecessor to v on the shortest path from s to v

Steps:

- **Initialise** $s.d = 0$ and $v.d = \infty$ for all $v \in G.V - \{s\}$.
  - invariant: $\mathit{distance}(s, v) \leq v.d \leq \infty$
- **Initialise** $v.\pi = \mathit{NIL}$ for all $v \in G.V$.
  - invariant: $v.\pi$ is a predecessor on shortest path found so far
- **Visit** each vertex in order of its distance from the source vertex – when we visit a vertex we **relax** its outgoing edges.

Generalises breadth-first search for weighted graphs.

DIJKSTRA(G, w, s)

How do we (efficiently) find the next vertex to visit?

Let

S the set of **visited vertices**

We maintain

Q a **priority queue** containing vertices $V - S$, with **key** $v.d$

At each step we visit the vertex u on Q with the minimum key.

When we visit vertex u, we are guaranteed to have already found a shortest path to u – *assuming that the graph has no negative weight edges!*.

DIJKSTRA(G, w, s): why does the idea work?

When we visit a vertex u, we have already found a shortest path to u.

Let v be the predecessor of u on a shortest path from s to u.

$distance(s, v) \leq distance(s, u)$, and so either

- we have already visited v and relaxed its edges, or
- we already found another shortest-path to u of length $distance(s, u) = distance(s, v)$

Dijkstra's algorithm

DIJKSTRA(G, w, s)

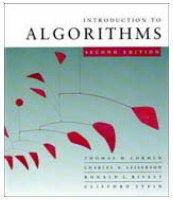
```
1  // G is the graph, w the weight function, s the source vertex
2  INIT-SINGLE-SOURCE(G, s)
3  S = ∅            // S is the set of visited vertices
4  Q = G.V          // Q is a priority queue maintaining G.V − S
5  while Q ≠ ∅
6        u = EXTRACT-MIN(Q)
7        S = S ∪ {u}
8        for each vertex v ∈ G.Adj[u]
9              RELAX(u, v, w)
```

INIT-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2        v.d = ∞
3        v.π = NIL
4  s.d = 0
```

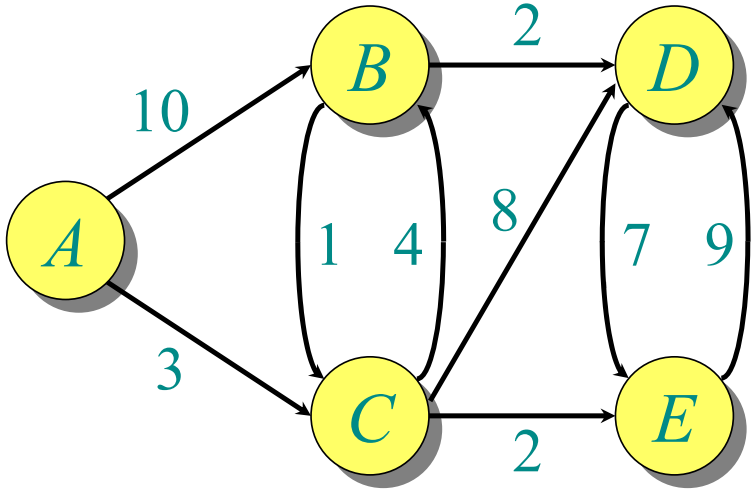
RELAX(u, v, w)

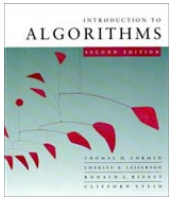
```
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```



Example of Dijkstra's algorithm

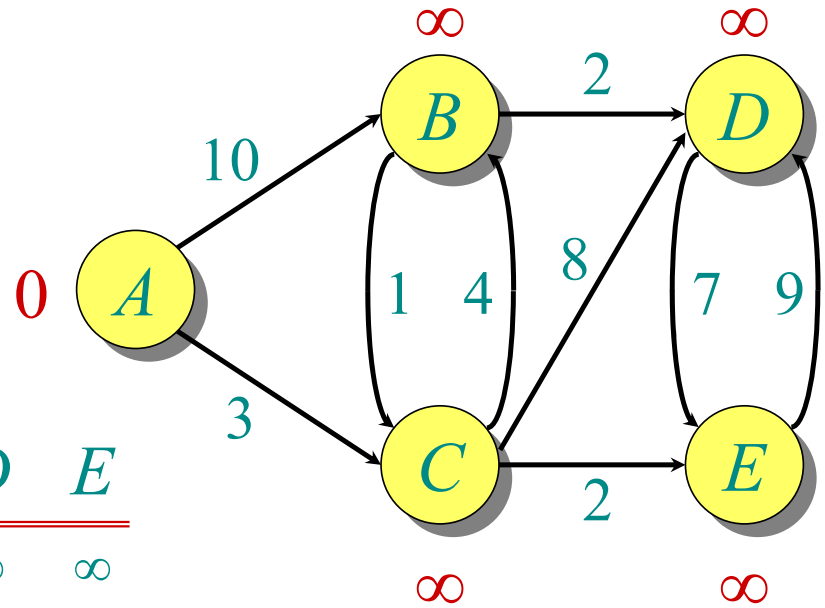
Graph with nonnegative edge weights:





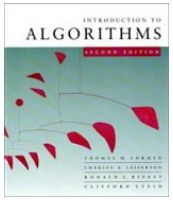
Example of Dijkstra's algorithm

Initialize:



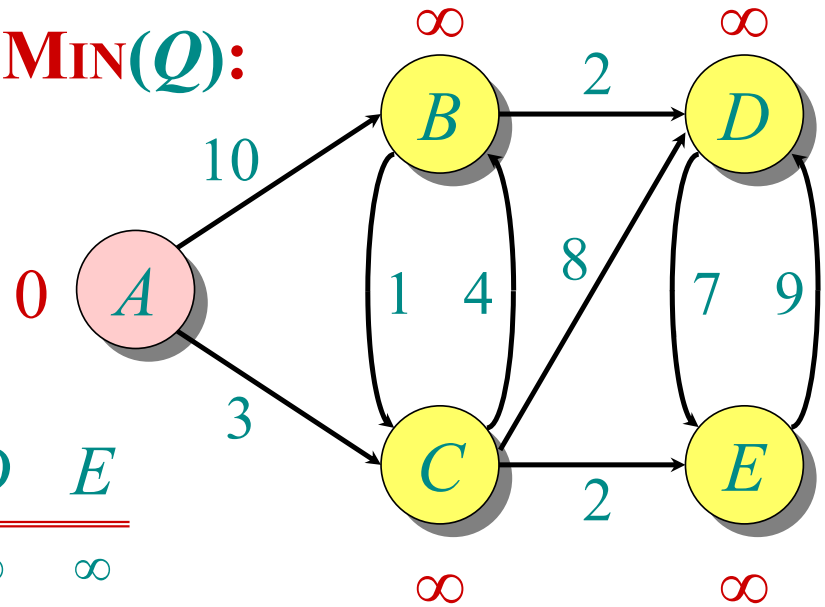
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}



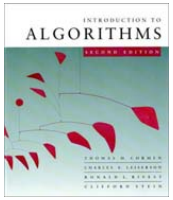
Example of Dijkstra's algorithm

"A" ← EXTRACT-MIN(Q):



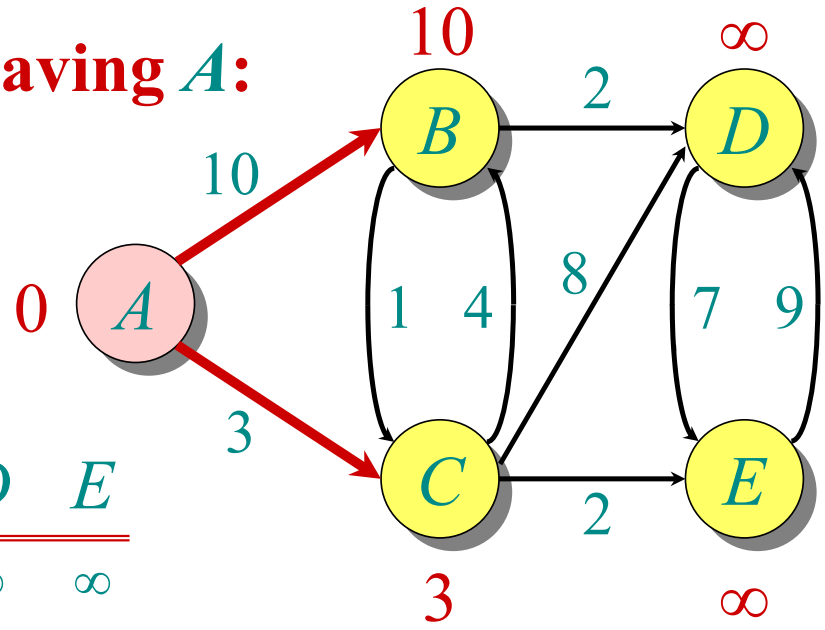
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: { A }



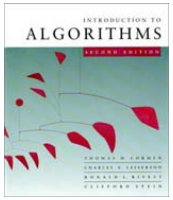
Example of Dijkstra's algorithm

Relax all edges leaving A:



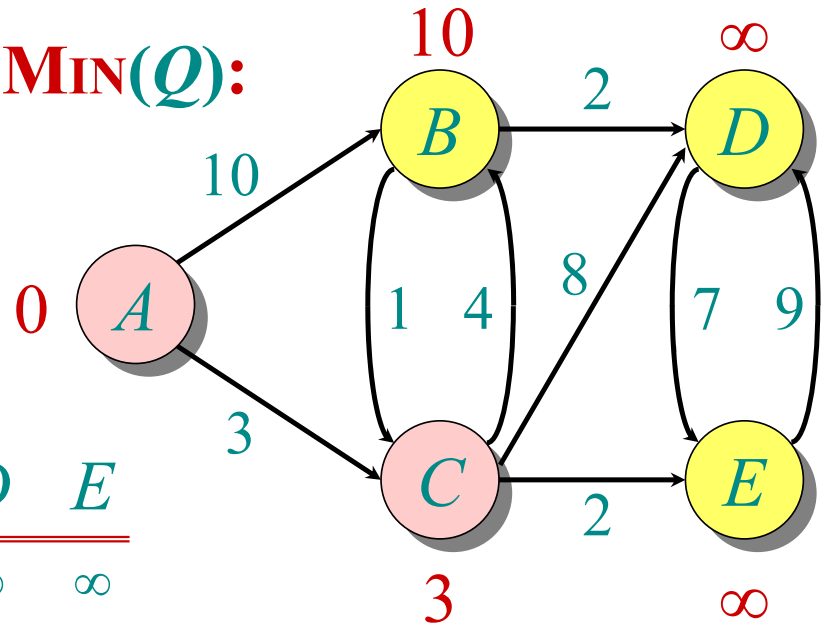
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }



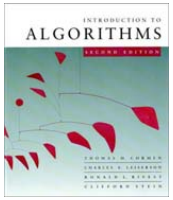
Example of Dijkstra's algorithm

"C" ← EXTRACT-MIN(Q):



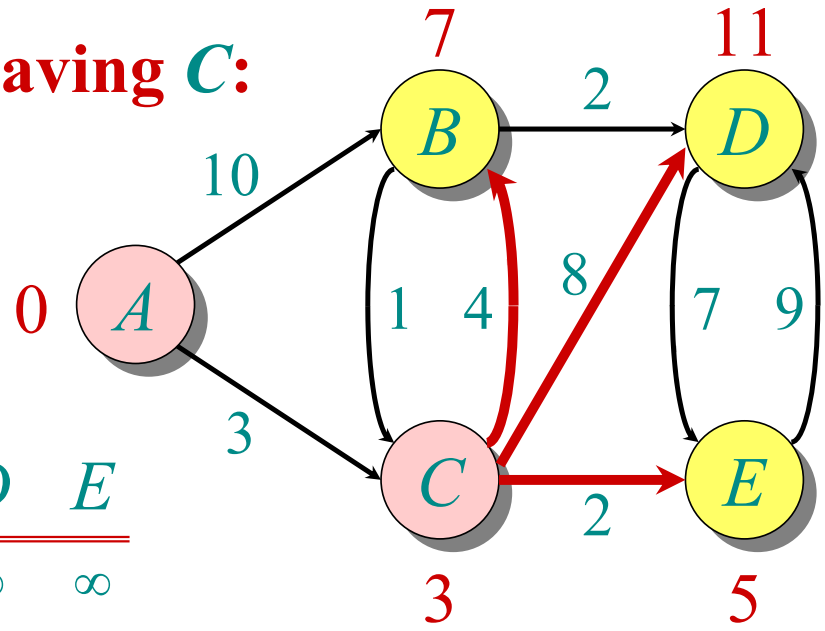
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }



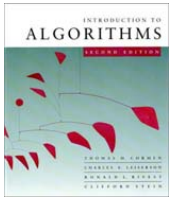
Example of Dijkstra's algorithm

Relax all edges leaving *C*:



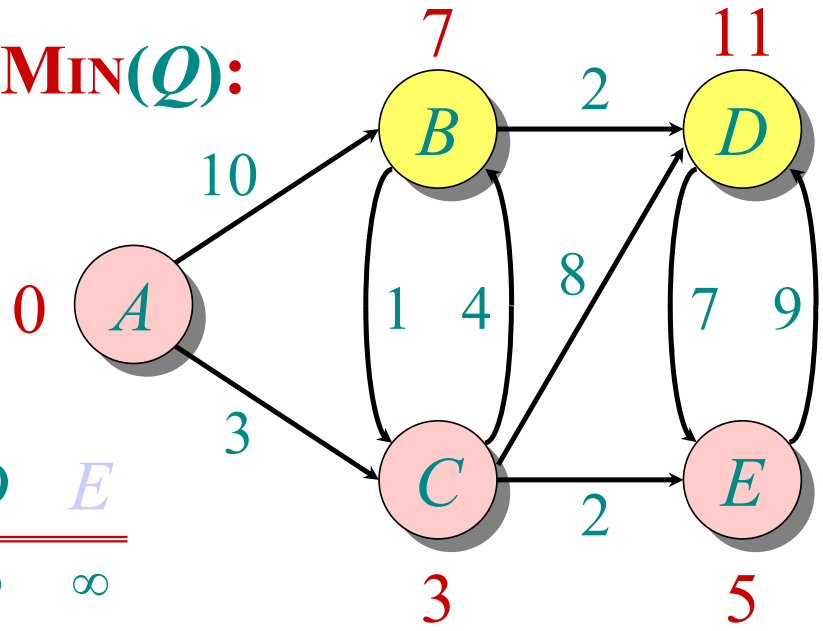
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }



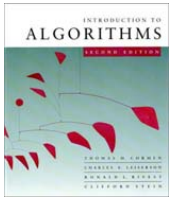
Example of Dijkstra's algorithm

"*E*" ← EXTRACT-MIN(*Q*):



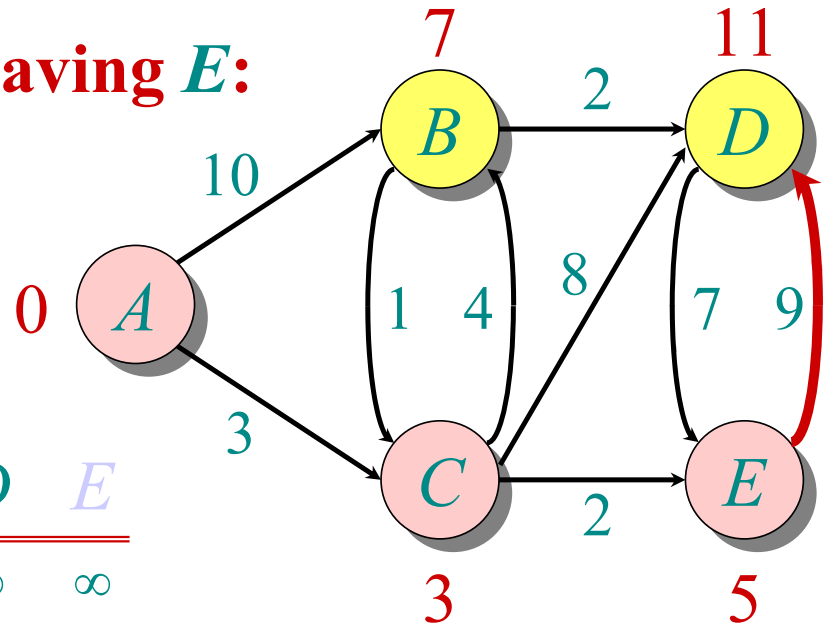
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C*, *E* }



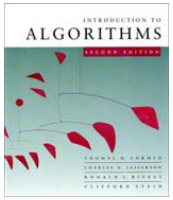
Example of Dijkstra's algorithm

Relax all edges leaving *E*:



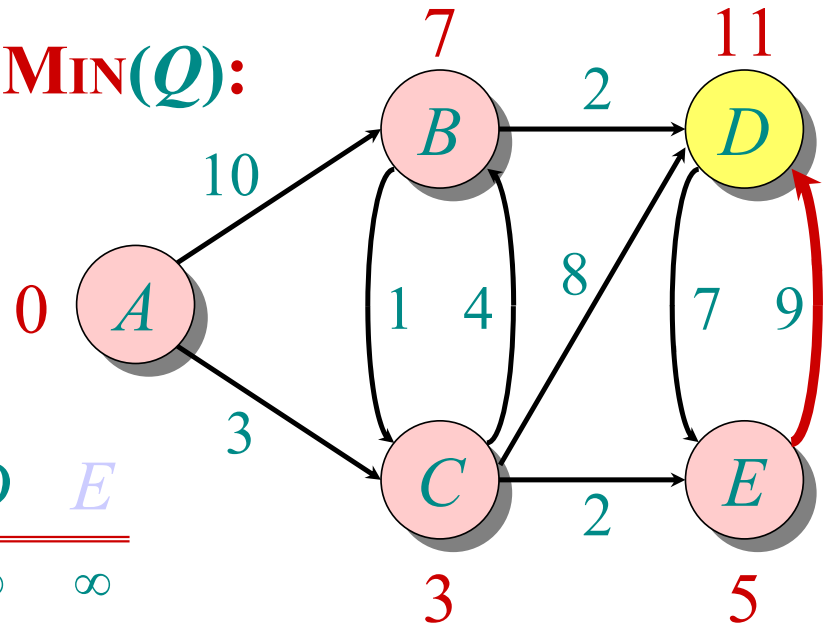
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }



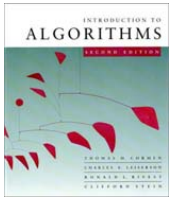
Example of Dijkstra's algorithm

"*B*" ← EXTRACT-MIN(*Q*):



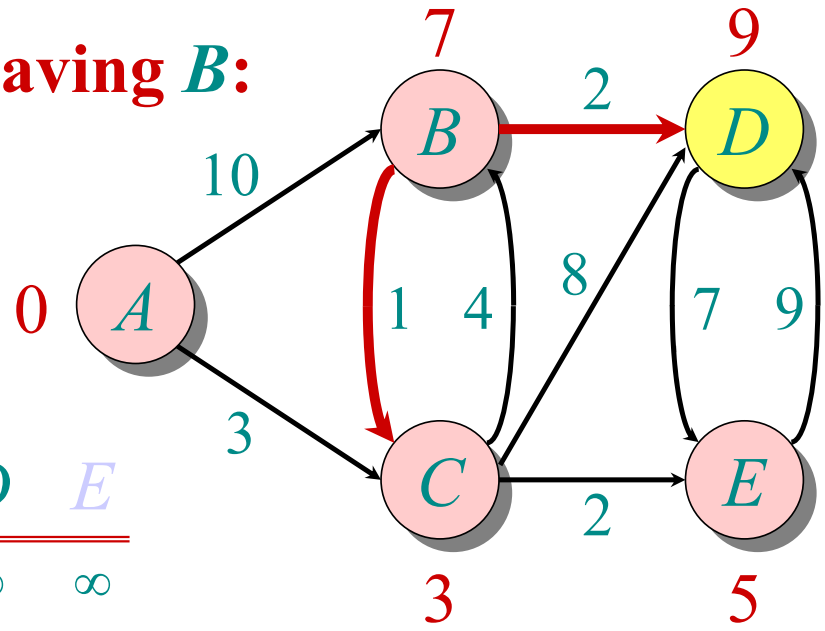
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E*, *B* }



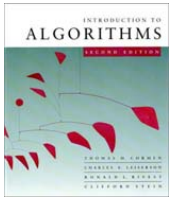
Example of Dijkstra's algorithm

Relax all edges leaving *B*:



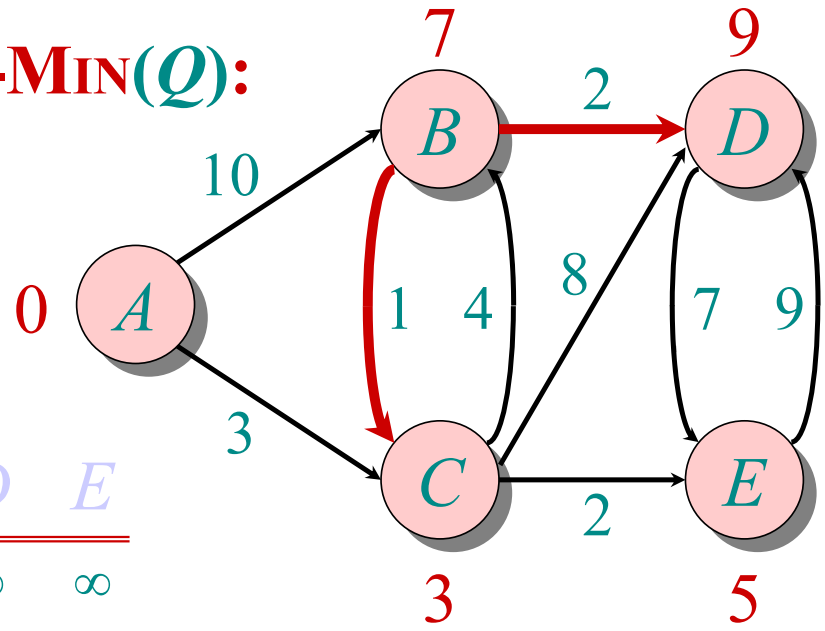
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B* }



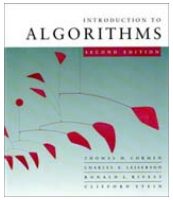
Example of Dijkstra's algorithm

"*D*" ← EXTRACT-MIN(*Q*):



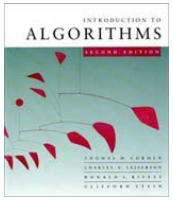
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B*, *D* }



Analysis of Dijkstra

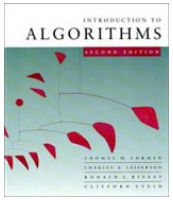
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$



Analysis of Dijkstra

$|V|$ times

while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$



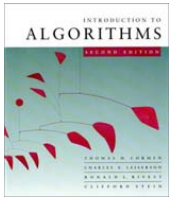
Analysis of Dijkstra

$|V|$ times

while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$

$degree(u)$ times

for each $v \in Adj[u]$
  do if $d[v] > d[u] + w(u, v)$
    then $d[v] \leftarrow d[u] + w(u, v)$



Analysis of Dijkstra

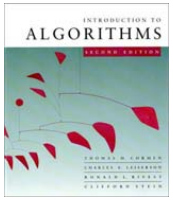
$|V|$ times

while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$

$degree(u)$ times

for each $v \in Adj[u]$
  do if $d[v] > d[u] + w(u, v)$
    then $d[v] \leftarrow d[u] + w(u, v)$

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.



Analysis of Dijkstra

$|V|$ times

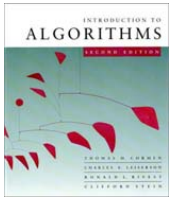
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$

$degree(u)$ times

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

Time $= \Theta(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{DECREASE-KEY}})$

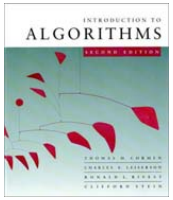
Note: Same formula as in the analysis of Prim's minimum spanning tree algorithm.



Analysis of Dijkstra (continued)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

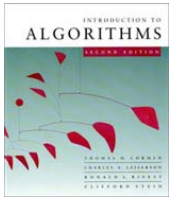
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total



Analysis of Dijkstra (continued)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

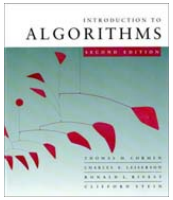
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$



Analysis of Dijkstra (continued)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$



Analysis of Dijkstra (continued)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

Negative-weight problems

What if our graph has negative weights?

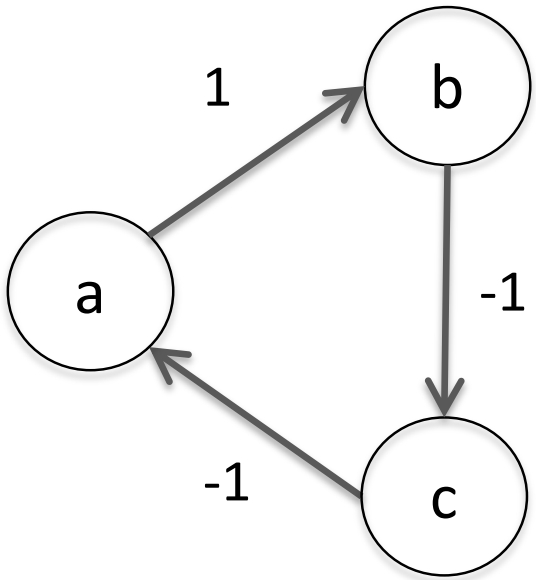
Dijkstra's algorithm might not relax edges in the right order.

I.e. (u, v) might be relaxed before the shortest path to u has been found.

Negative-weight problems

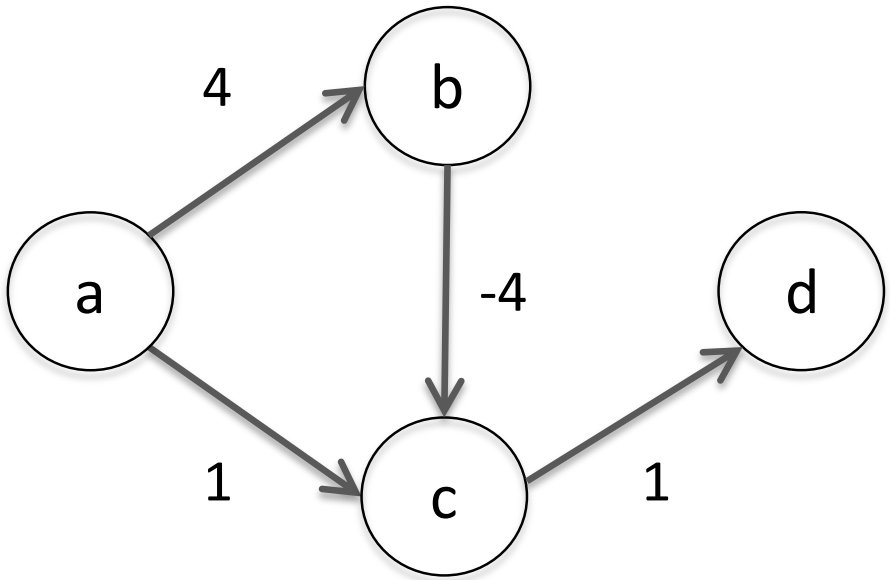
Are shortest paths well-defined for graphs with negative weight cycles?

What is the shortest path from vertex *a* to *a*?



Quick question

Give a directed, weighted graph **with negative weights**, but no negative weight cycles, for which Dijkstra's algorithm fails.



Negative weight problems

- Graphs with negative weights might have **negative weight cycles**.
- Even if a graph with negative weights has no negative-weight cycles, we need to work out how to relax edges so that we are guaranteed to find shortest paths.

Negative weight problems: the Bellman-Ford solution

What do we know?

After initialization:
we have found all shortest paths that contain ≤ 0 edges.

After relaxing each edge in the graph 1 time:
we have found all shortest paths that contain ≤ 1 edges.

After relaxing each edge in the graph 2 times:
we have found all shortest paths that contain ≤ 2 edges.

...

After relaxing each edge in the graph $V-1$ times:
we have found all shortest paths that contain $\leq V-1$ edges.

Bellman-Ford algorithm

The Bellman-Ford algorithm uses this idea to find:

- single-source shortest paths on directed graphs
- that may have negative-weight edges.

Moreover, it

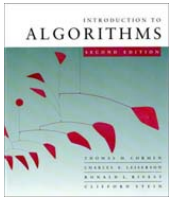
- detects negative weight cycles, returning *false* if one is found.

Bellman-Ford algorithm

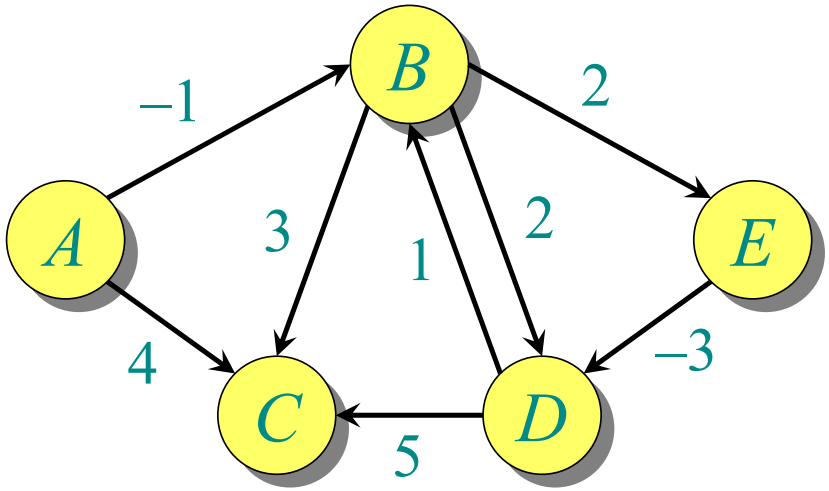
BELLMAN-FORD(G, w, s)

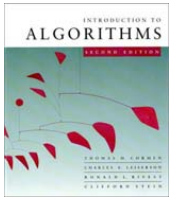
```
1   // G is the graph, w the weight function, s the source vertex
2   INIT-SINGLE-SOURCE(G, s)
3   // Relax each edge V−1 times to find shortest paths
4   for i = 1 to |G.V| − 1
5         for each edge (u, v) ∈ G.E
6               RELAX(u, v, w)
7   // Check for negative-weight cycles
8   for each edge (u, v) ∈ G.E
9         if v.d > u.d + w(u, v)
10              return FALSE
11  return TRUE
```

What is the time complexity? $O(VE)$ (i.e., could be $O(V^3)$).

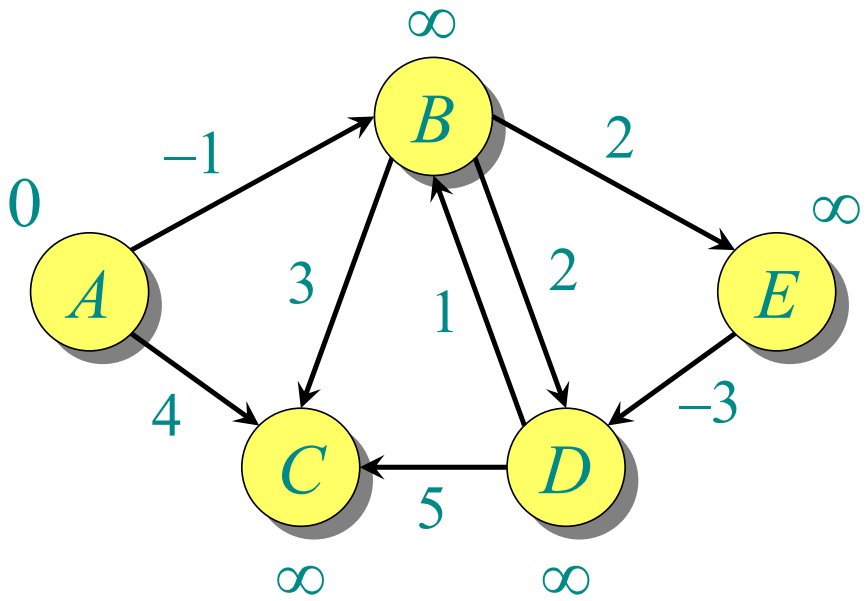


Example of Bellman-Ford

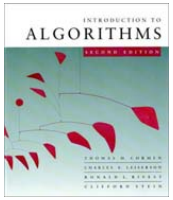




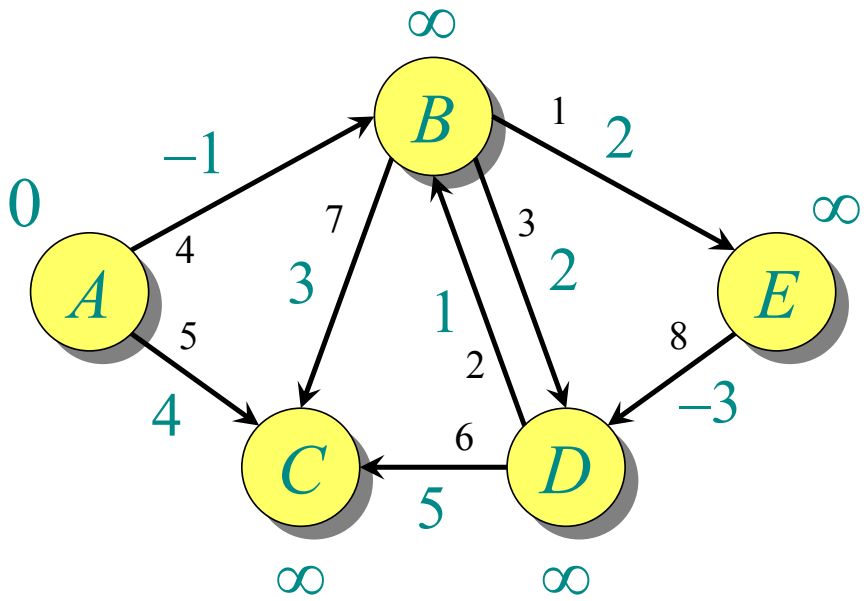
Example of Bellman-Ford



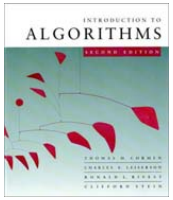
Initialization.



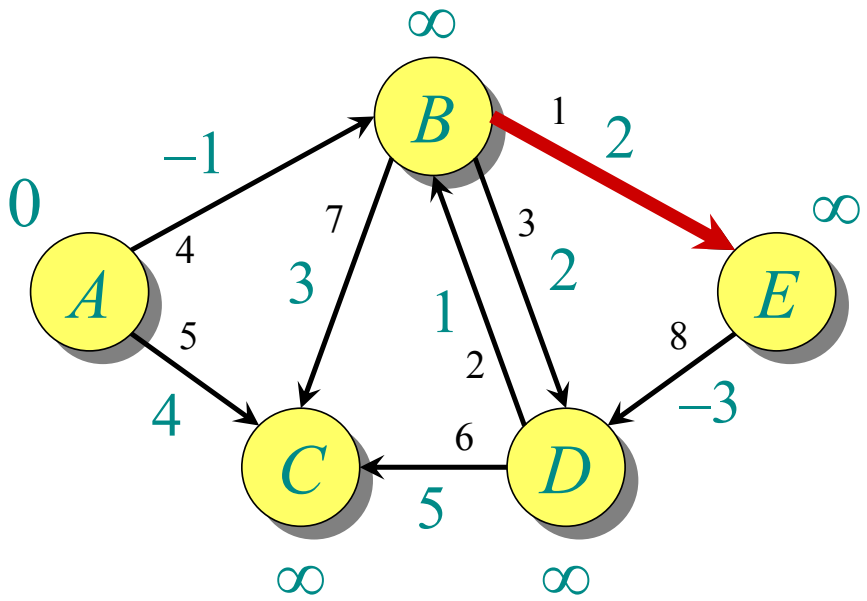
Example of Bellman-Ford

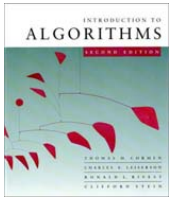


Order of edge relaxation.

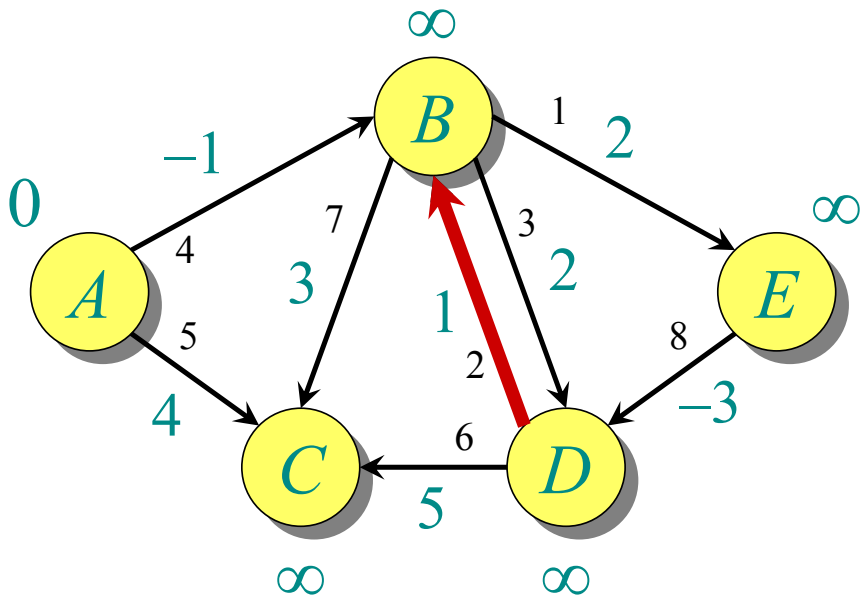


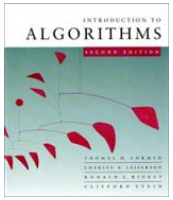
Example of Bellman-Ford



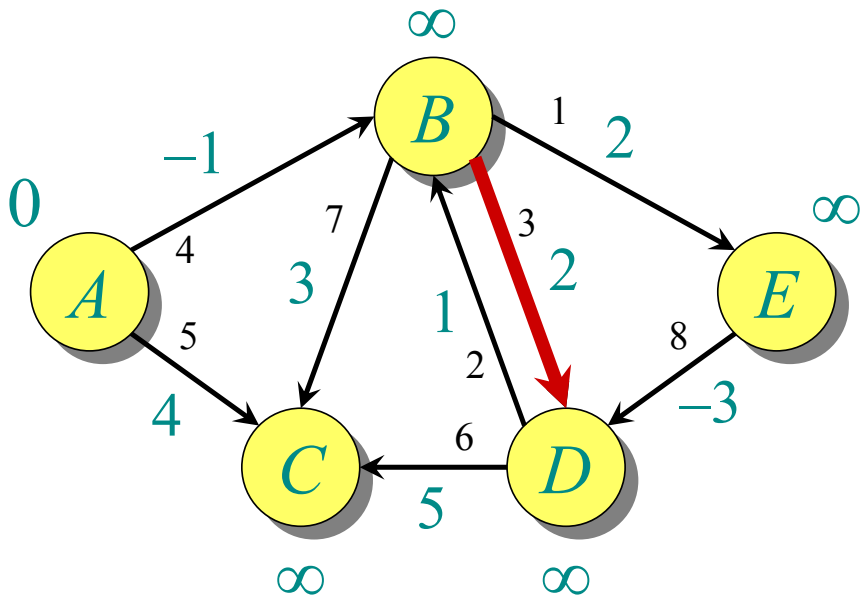


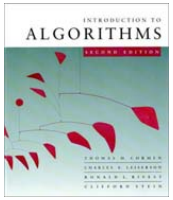
Example of Bellman-Ford



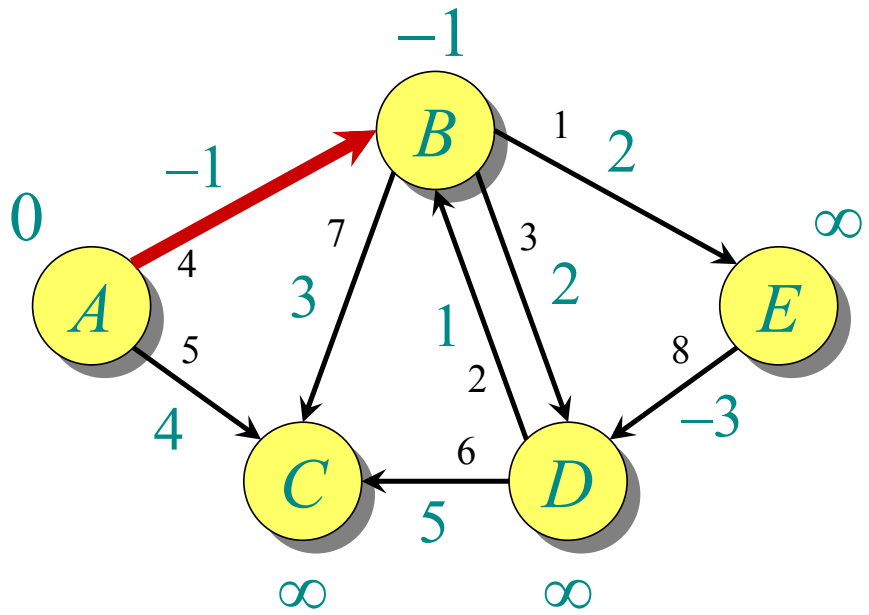


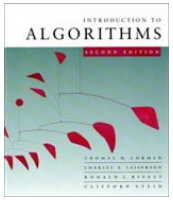
Example of Bellman-Ford



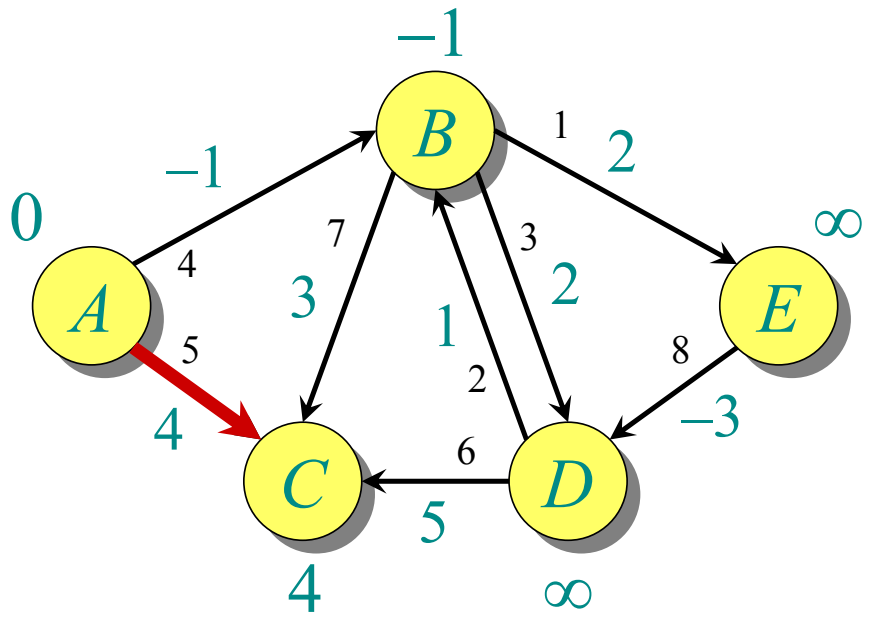


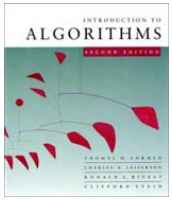
Example of Bellman-Ford



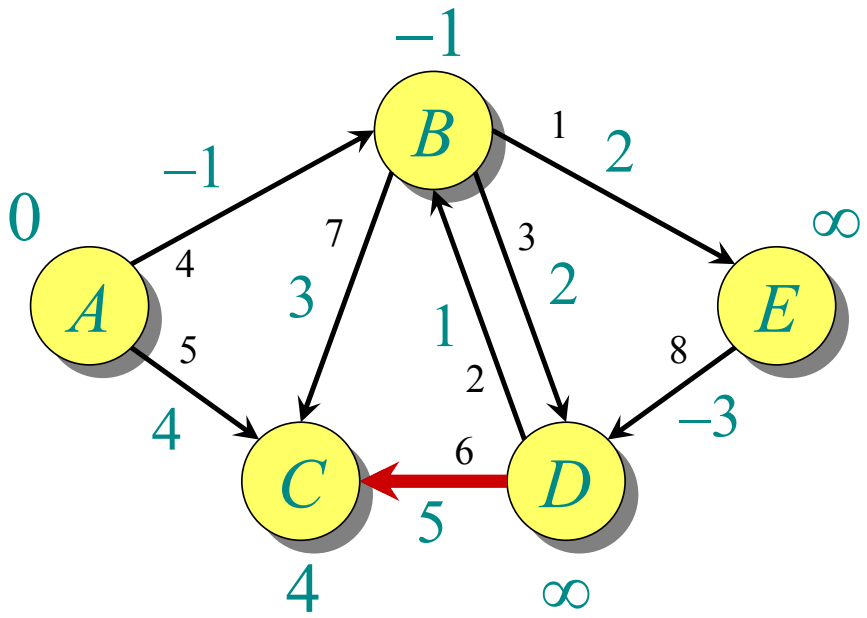


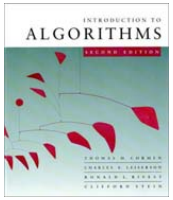
Example of Bellman-Ford



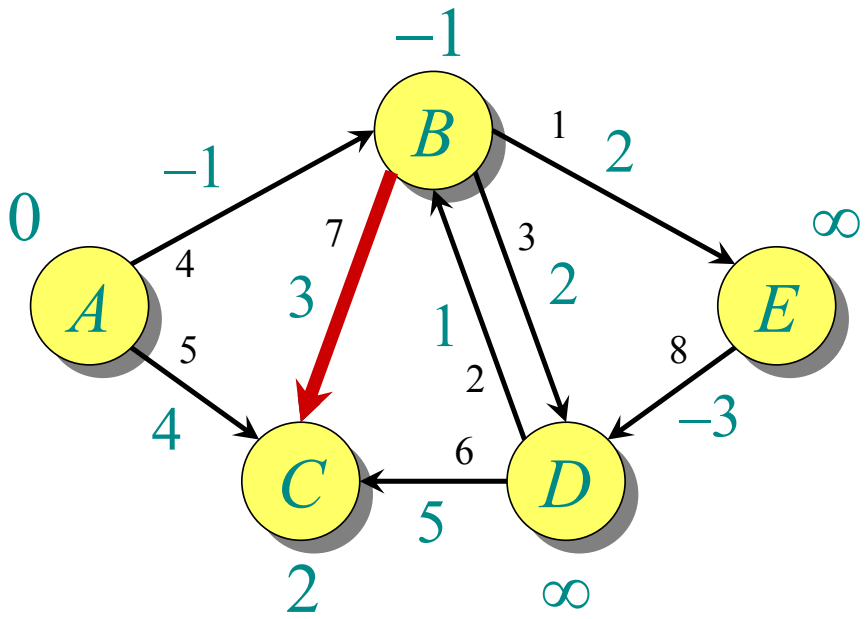


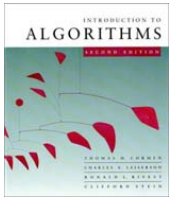
Example of Bellman-Ford



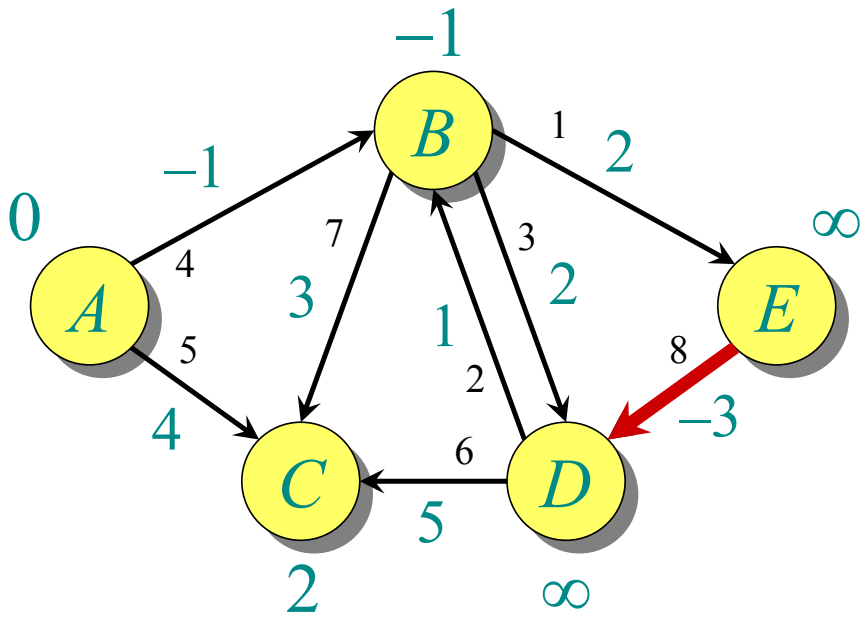


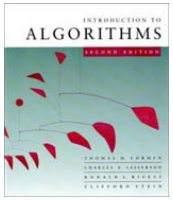
Example of Bellman-Ford



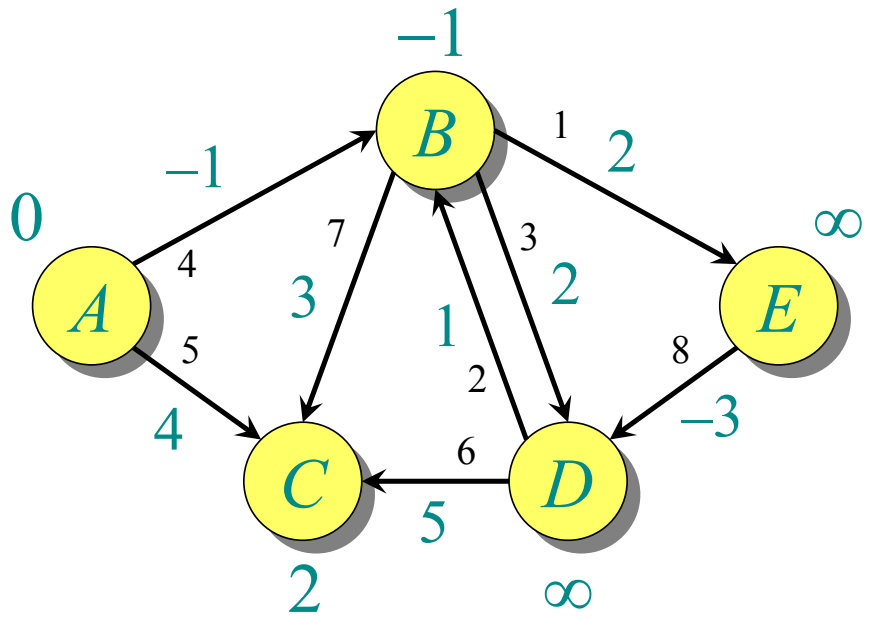


Example of Bellman-Ford

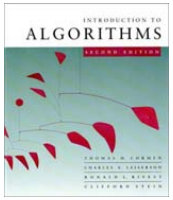




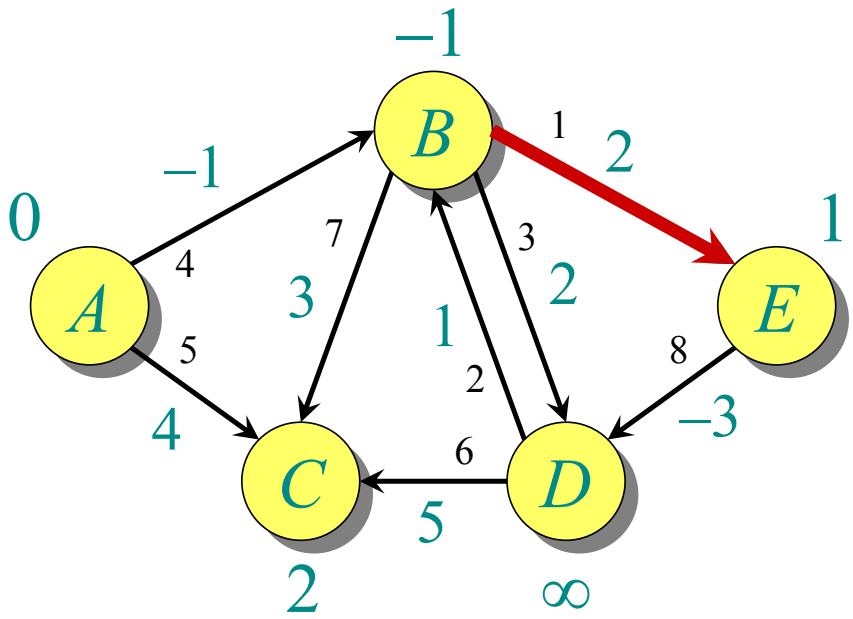
Example of Bellman-Ford

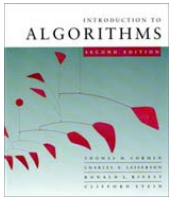


End of pass 1.

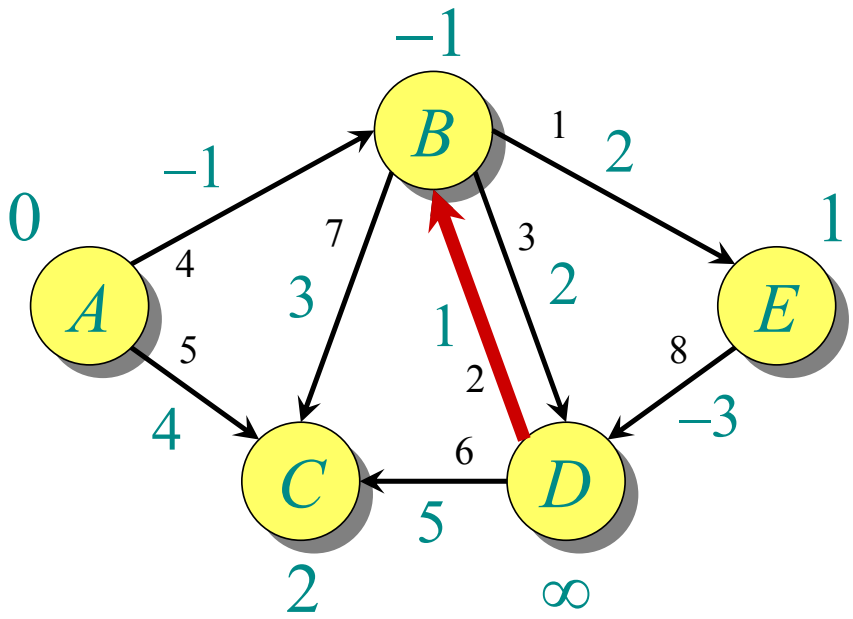


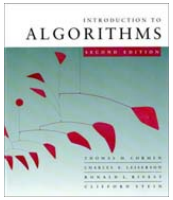
Example of Bellman-Ford



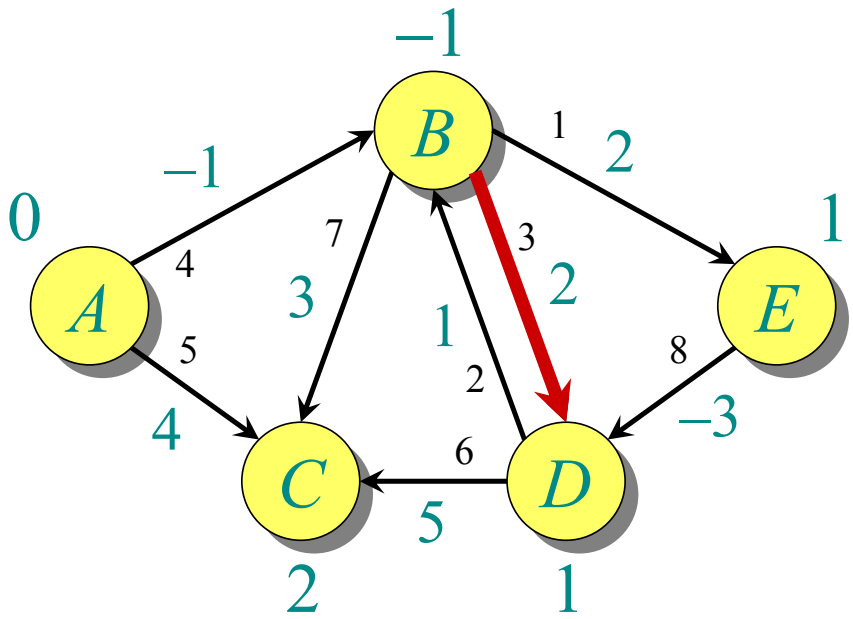


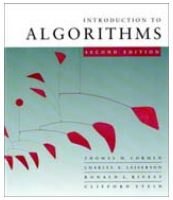
Example of Bellman-Ford



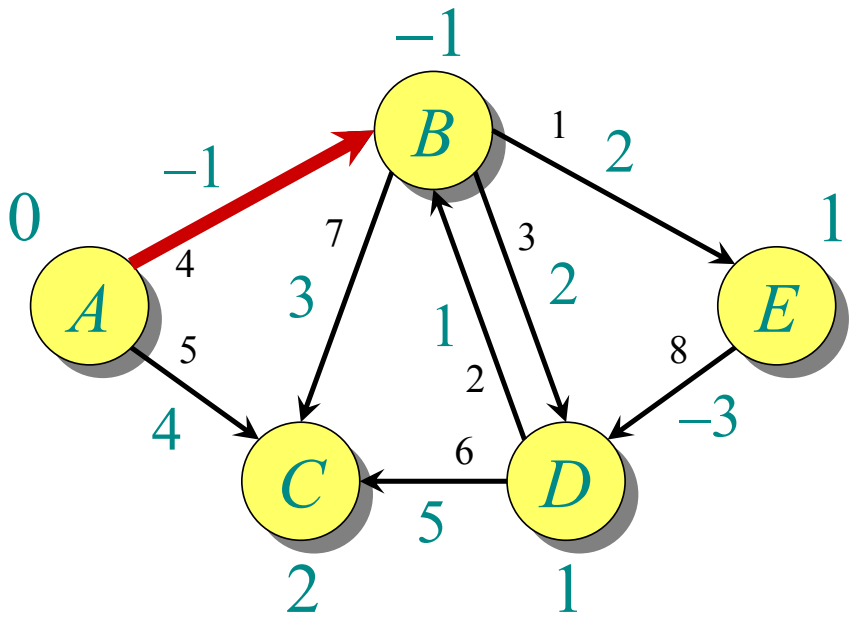


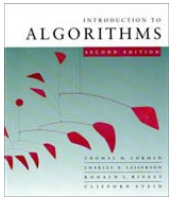
Example of Bellman-Ford



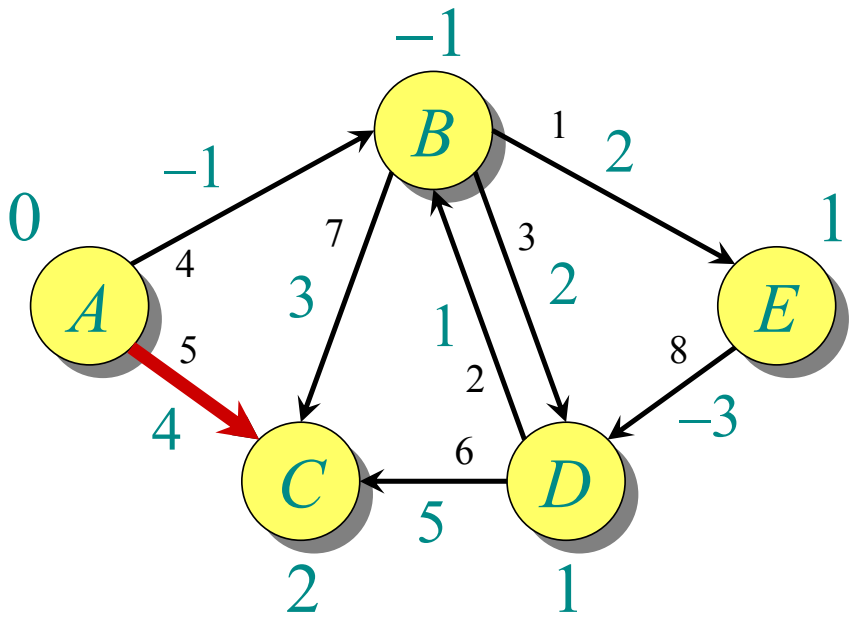


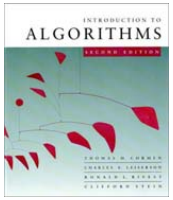
Example of Bellman-Ford



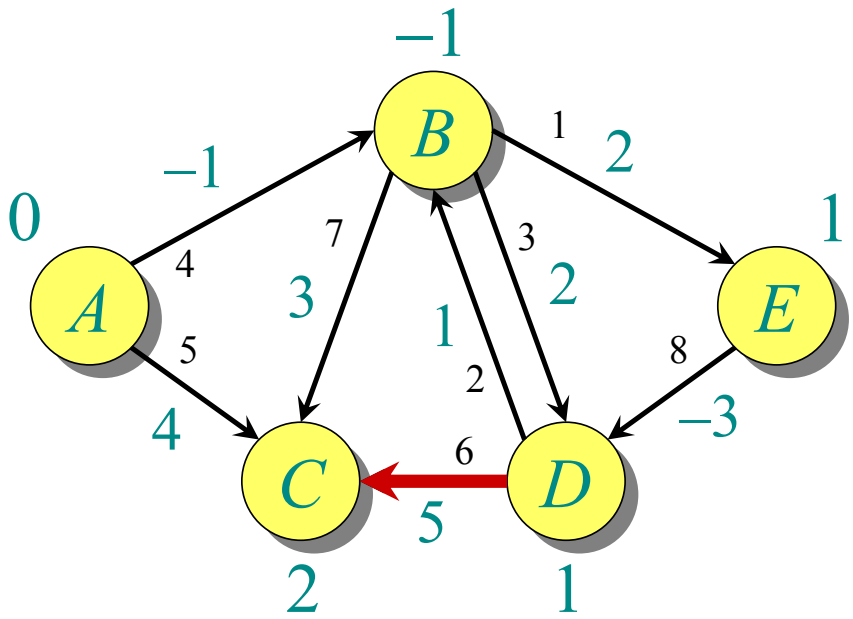


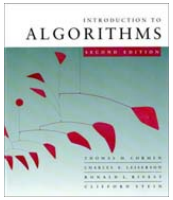
Example of Bellman-Ford



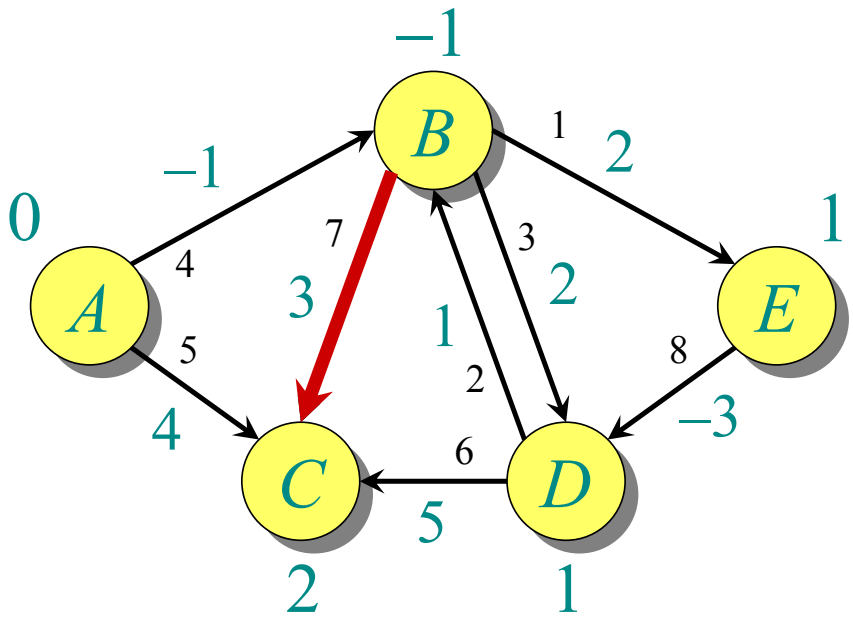


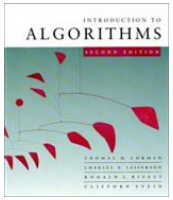
Example of Bellman-Ford



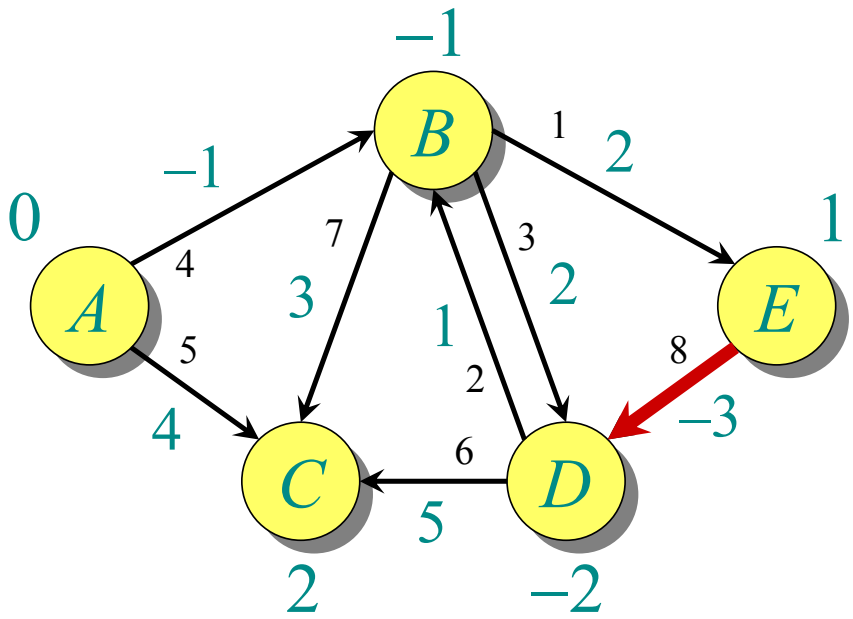


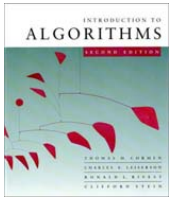
Example of Bellman-Ford



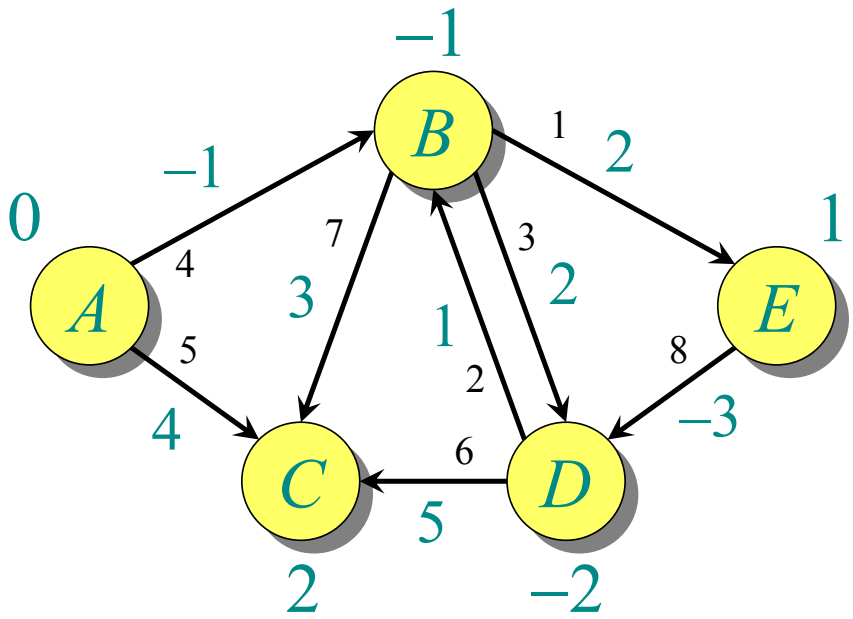


Example of Bellman-Ford





Example of Bellman-Ford



End of pass 2 (and 3 and 4).

Quick question

Is there a more efficient way to find single-source shortest paths for directed acyclic graphs?

Priority first search

- Dijkstra's algorithm for finding single-source shortest paths
- Prim's algorithm for finding minimum spanning trees

are specializations of **priority-first search**.

PRIORITYFIRSTSEACH($G, s, \ldots$)

Vertices are visited in order of their **priority**.

For each vertex $v \in G.V$ we calculate:

$v.key$ it's priority relative to source vertex s

$v.\pi$ the predecessor to v in a **priority-first search tree**

The algorithm uses:

Q a priority queue of unvisited vertices with key $v.key$

Priority-first search

```
PRIORITYFIRSTSEACH(G, s, ...)
1   INIT-SINGLE-SOURCE(G, s)
2   Q = G.V
3   while Q ≠ ∅
4         u = EXTRACT-MIN(Q)
5         for each vertex v ∈ G.Adj[u]
6               RELAX(u, v, ...)
```

```
INIT-SINGLE-SOURCE(G, s)
1   for each vertex v ∈ G.V
2         v.key = ∞
3         v.π = NIL
4   s.key = 0
```

```
RELAX(u, v, ...)
1   if v.key > PRIORITY
2         v.key = PRIORITY
3         v.π = u
```

Priority-first search: Prim's algorithm

Instantiate PRIORITY $= \mathit{weight}(u, v)$.

- $G.V - Q$ is the subset of vertices in the MST under construction (T)
- $v.\mathit{key}$ is the least weight edge connecting v to T
- $v.\pi$ is the vertex adjacent to v on that least-weight edge

At the end, the priority-first search tree is a **minimum spanning tree**.

Priority-first search: Dijkstra's algorithm

Instantiate PRIORITY $= u.key + w(u, v)$.

- $G.V - Q$ is the set of vertices already visited in order of their distance from s
- $v.key$ is the length of the shortest-path from s to v that only uses edges adjacent to visited vertices.
- $v.\pi$ is the predecessor of v on the shortest-path from s to v that only used edges adjacent to visited vertices.

At the end, the priority-first search tree is a **shortest-path-tree**.

Recap of this week

1. Single-source shortest path algorithms:
   - Dijkstra's algorithm:
     - For graphs without negative-weight edges
     - $O(E \lg V)$ using binary heap or
     - $O(E + V \lg V)$ (using Fibonacci heap)
   - Bellman-ford algorithm:
     - Handles graphs with negative-weight edges
     - Can detect negative-weight cycles
     - $O(EV)$
   - Priority-first search:
     - Generalises Prim and Dijkstra's algorithms