

## 6. DYNAMIC PROGRAMMING II

---

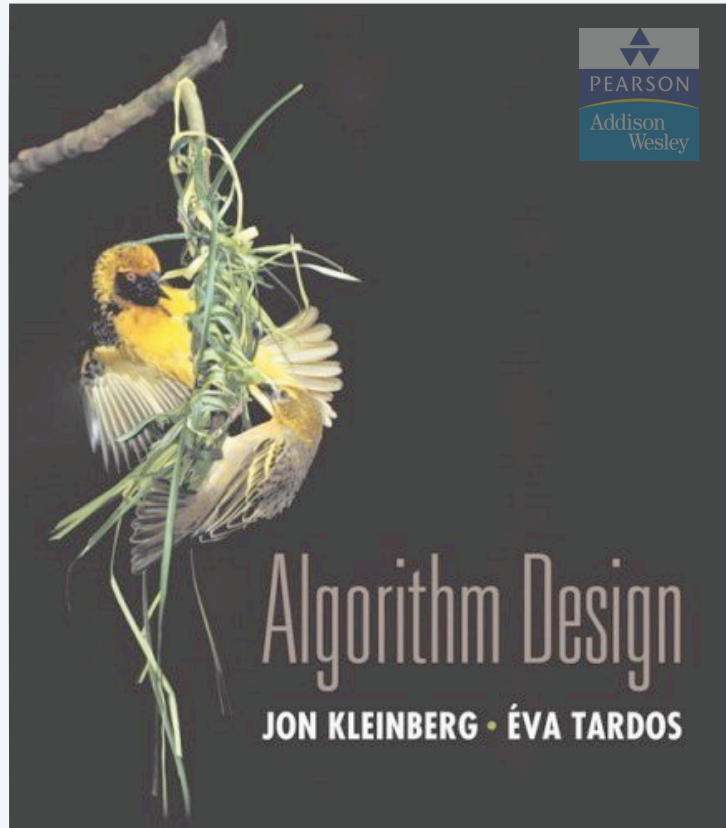
- ▶ *sequence alignment*
- ▶ *Hirschberg's algorithm*
- ▶ *Bellman-Ford algorithm*
- ▶ *distance vector protocols*
- ▶ *negative cycles in a digraph*

Lecture slides by Kevin Wayne

Copyright © 2005 Pearson-Addison Wesley

Copyright © 2013 Kevin Wayne

<http://www.cs.princeton.edu/~wayne/kleinberg-tardos>



## SECTION 6.6

# 6. DYNAMIC PROGRAMMING II

---

- ▶ *sequence alignment*
- ▶ *Hirschberg's algorithm*
- ▶ *Bellman-Ford algorithm*
- ▶ *distance vector protocols*
- ▶ *negative cycles in a digraph*

# String similarity

---

Q. How similar are two strings?

Ex. occurrence and occurance.

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

# Edit distance

---

**Edit distance.** [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty  $\delta$ ; mismatch penalty  $\alpha_{pq}$ .
- Cost = sum of gap and mismatch penalties.

*Cost depends on which characters are.*

C	T	-	G	A	C	C	T	A	C	G
C	T	G	G	A	C	G	A	A	C	G

$$\text{cost} = \delta + \alpha_{CG} + \alpha_{TA}$$

**Applications.** Unix diff, speech recognition, computational biology, ...

# Sequence alignment

**Goal.** Given two strings  $x_1 x_2 \dots x_m$  and  $y_1 y_2 \dots y_n$  find min cost alignment.

**Def.** An **alignment**  $M$  is a set of ordered pairs  $x_i - y_j$  such that each item occurs in at most one pair and no crossings.

$x_i - y_j$  and  $x_{i'} - y_{j'}$  cross if  $i < i'$ , but  $j > j'$

**Def.** The **cost** of an alignment  $M$  is:

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta}_{\text{gap}}$$

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$		$x_6$
C	T	A	C	C	—	G
—	T	A	C	A	T	G
$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	

an alignment of CTACCG and TACATG:

$$M = \{x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6\}$$

## Sequence alignment: problem structure

*reduce into recursive. (DYNAMIC NO CONVERSIONS)*

**Def.**  $OPT(i, j) = \min$  cost of aligning prefix strings  $x_1 x_2 \dots x_i$  and  $y_1 y_2 \dots y_j$ .

**Case 1.**  $OPT$  matches  $x_i - y_j$ .

Pay mismatch for  $x_i - y_j$  +  $\min$  cost of aligning  $x_1 x_2 \dots x_{i-1}$  and  $y_1 y_2 \dots y_{j-1}$ .

**Case 2a.**  $OPT$  leaves  $x_i$  unmatched.

Pay gap for  $x_i$  +  $\min$  cost of aligning  $x_1 x_2 \dots x_{i-1}$  and  $y_1 y_2 \dots y_j$ .

**Case 2b.**  $OPT$  leaves  $y_j$  unmatched.

Pay gap for  $y_j$  +  $\min$  cost of aligning  $x_1 x_2 \dots x_i$  and  $y_1 y_2 \dots y_{j-1}$ .

optimal substructure property  
(proof via exchange argument)

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

*dynamic programming algorithm.*

# Sequence alignment: algorithm

---

SEQUENCE-ALIGNMENT ( $m, n, x_1, \dots, x_m, y_1, \dots, y_n, \delta, \alpha$ )

FOR  $i = 0$  TO  $m$

$M[i, 0] \leftarrow i\delta.$

FOR  $j = 0$  TO  $n$

$M[0, j] \leftarrow j\delta.$

FOR  $i = 1$  TO  $m$

FOR  $j = 1$  TO  $n$

$M[i, j] \leftarrow \min \{ \alpha[x_i, y_j] + M[i-1, j-1],$

$\delta + M[i-1, j],$

$\delta + M[i, j-1] \}.$

RETURN  $M[m, n].$

Bottom-up  
Recursion.

min.

gaps.

# Sequence alignment: analysis

---

**Theorem.** The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length  $m$  and  $n$  in  $\Theta(mn)$  time and  $\Theta(mn)$  space.

**Pf.**

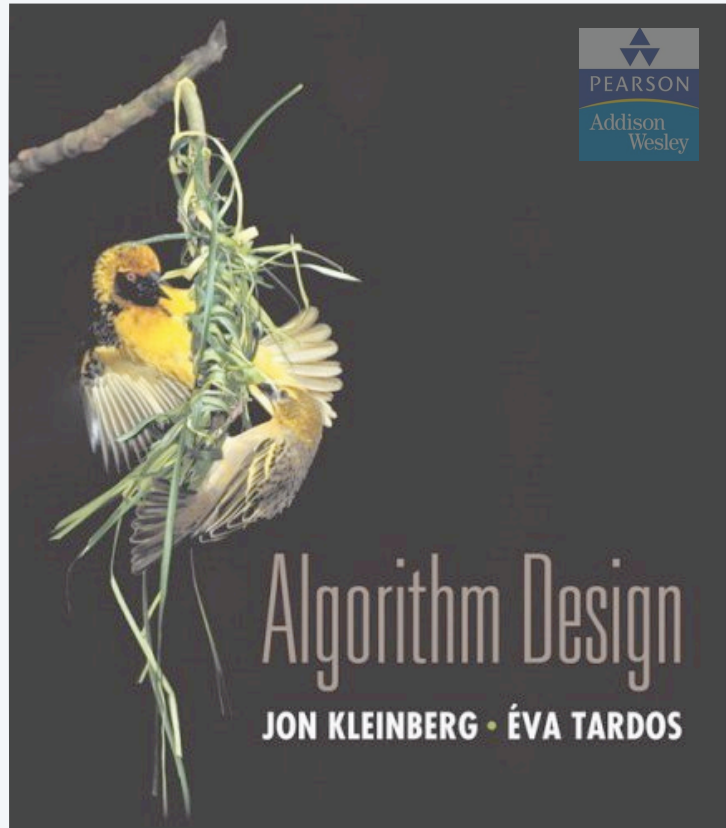
- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ■

**Q.** Can we avoid using quadratic space?

**A.** Easy to compute optimal value in  $O(mn)$  time and  $O(m + n)$  space.

- Compute  $\text{OPT}(i, \bullet)$  from  $\text{OPT}(i - 1, \bullet)$ .
- **But**, no longer easy to recover optimal alignment itself.





## SECTION 6.7

# 6. DYNAMIC PROGRAMMING II

---

- ▶ *sequence alignment*
- ▶ *Hirschberg's algorithm*
- ▶ *Bellman-Ford algorithm*
- ▶ *distance vector protocols*
- ▶ *negative cycles in a digraph*

Sequence alignment with linear memory  
occupy

# Sequence alignment in linear space

---

**Theorem.** There exist an algorithm to find an optimal alignment in  $O(mn)$  time and  $O(m + n)$  space.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Programming  
Techniques

G. Manacher  
Editor

---

## A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg  
Princeton University

---

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

Key Words and Phrases: subsequence, longest common subsequence, string correction, editing

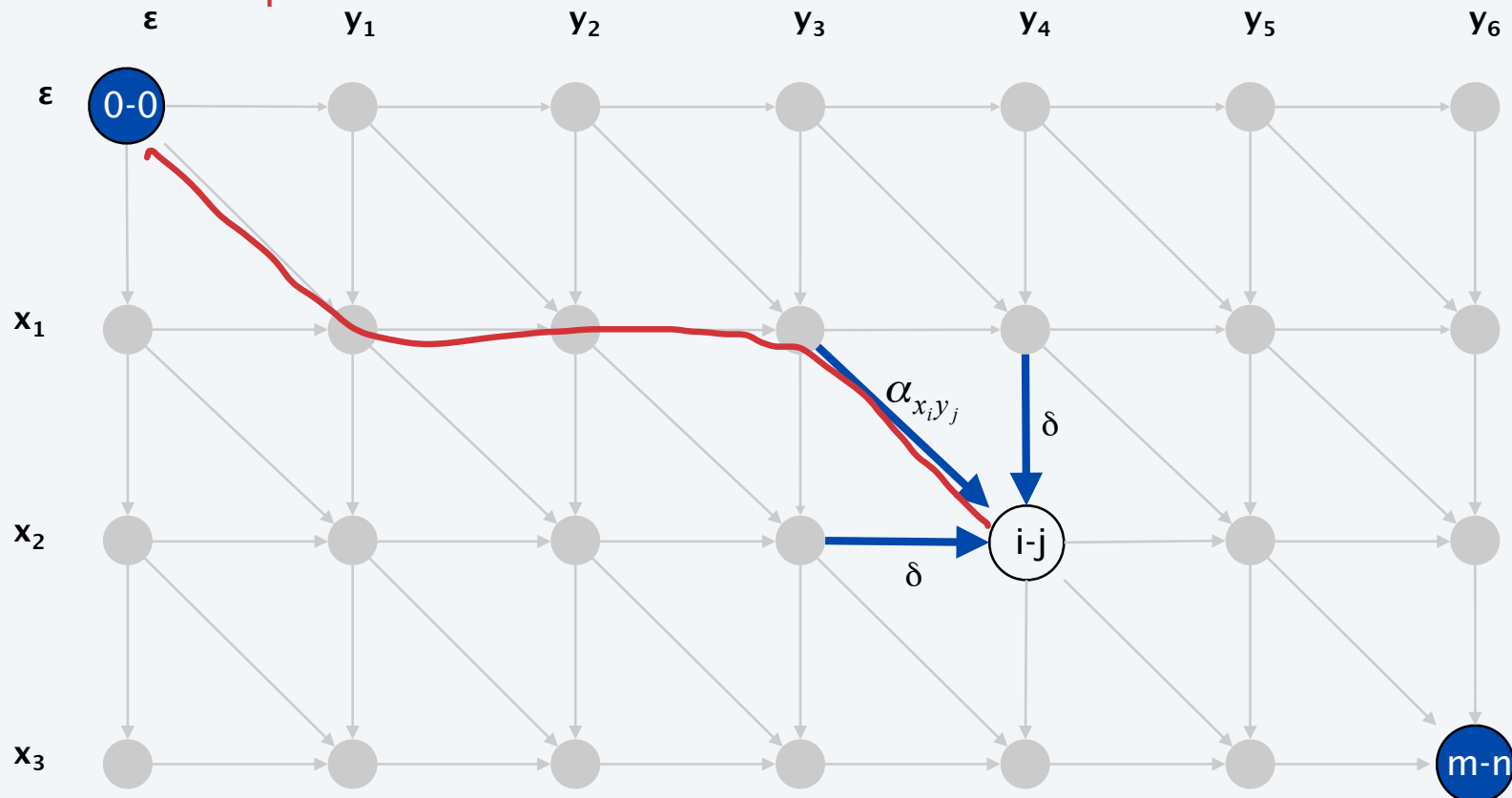
CR Categories: 3.63, 3.73, 3.79, 4.22, 5.25

# Hirschberg's algorithm

## Edit distance graph.

- Let  $f(i,j)$  be shortest path from  $(0,0)$  to  $(i,j)$ .
- Lemma:  $f(i,j) = OPT(i,j)$  for all  $i$  and  $j$ .

Opt is the shortest path from 0-0



# Hirschberg's algorithm

---

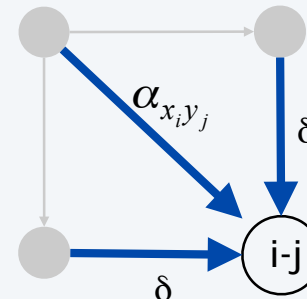
## Edit distance graph.

- Let  $f(i, j)$  be shortest path from  $(0, 0)$  to  $(i, j)$ .
- Lemma:  $f(i, j) = OPT(i, j)$  for all  $i$  and  $j$ .

## Pf of Lemma. [ by strong induction on $i + j$ ]

- Base case:  $f(0, 0) = OPT(0, 0) = 0$ .
- Inductive hypothesis: assume true for all  $(i', j')$  with  $i' + j' < i + j$ .
- Last edge on shortest path to  $(i, j)$  is from  $(i - 1, j - 1)$ ,  $(i - 1, j)$ , or  $(i, j - 1)$ .
- Thus,

$$\begin{aligned} \underline{f(i, j)} &= \min\{\alpha_{x_i y_j} + f(i - 1, j - 1), \delta + f(i - 1, j), \delta + f(i, j - 1)\} \\ &= \min\{\alpha_{x_i y_j} + OPT(i - 1, j - 1), \delta + OPT(i - 1, j), \delta + OPT(i, j - 1)\} \\ &= OPT(i, j) \quad \blacksquare \end{aligned}$$



## Edit distance graph.

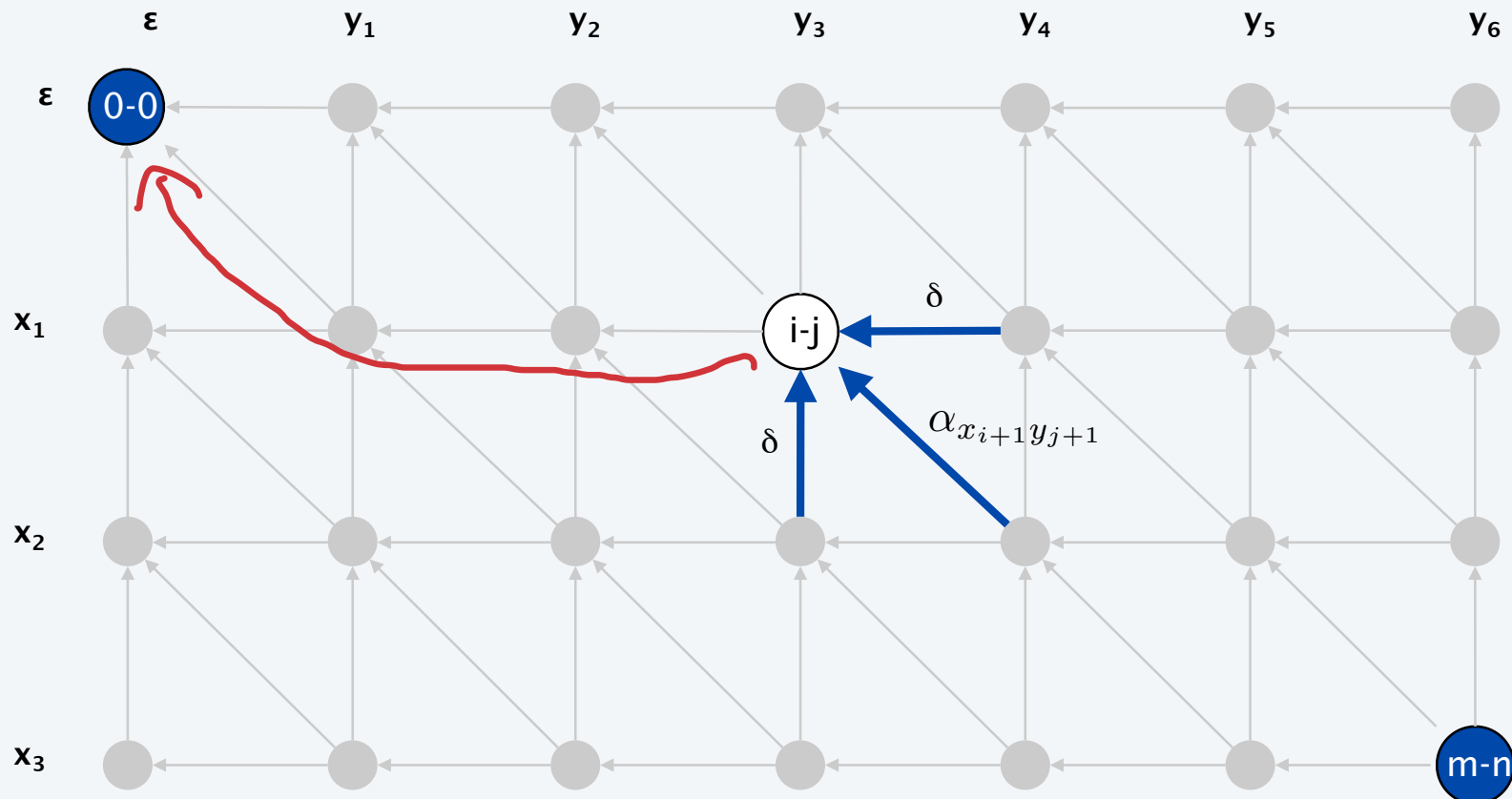
- 
- A red line graph on a light blue background. The line starts at a high level, remains relatively flat, then dips slightly to a minimum point, and finally rises again. A vertical black line segment marks the minimum point, with the letter 'j' written below it.



# Hirschberg's algorithm

## Edit distance graph.

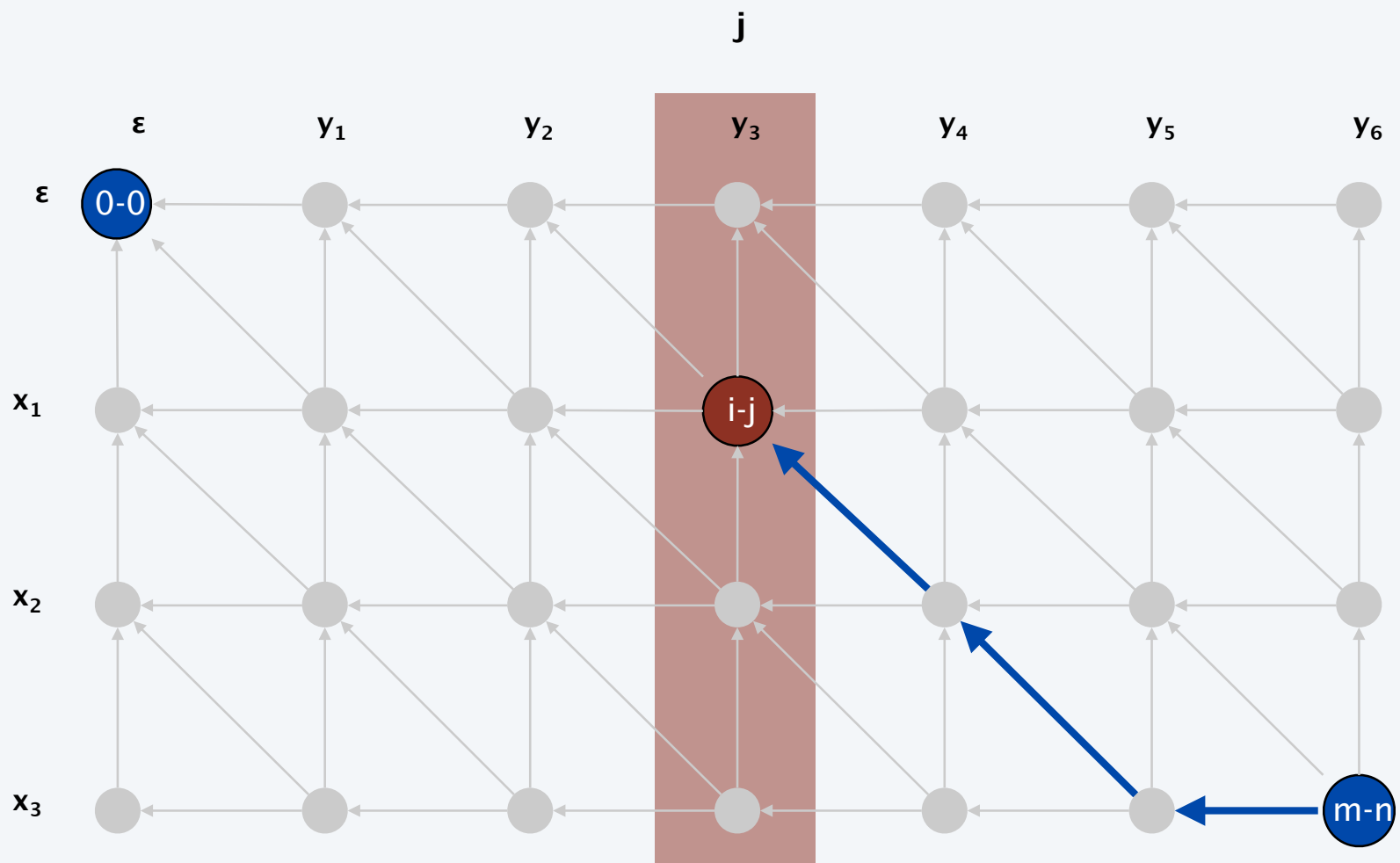
- Let  $g(i, j)$  be shortest path from  $(i, j)$  to  $(m, n)$ .
- Can compute by reversing the edge orientations and inverting the roles of  $(0, 0)$  and  $(m, n)$ .



# Hirschberg's algorithm

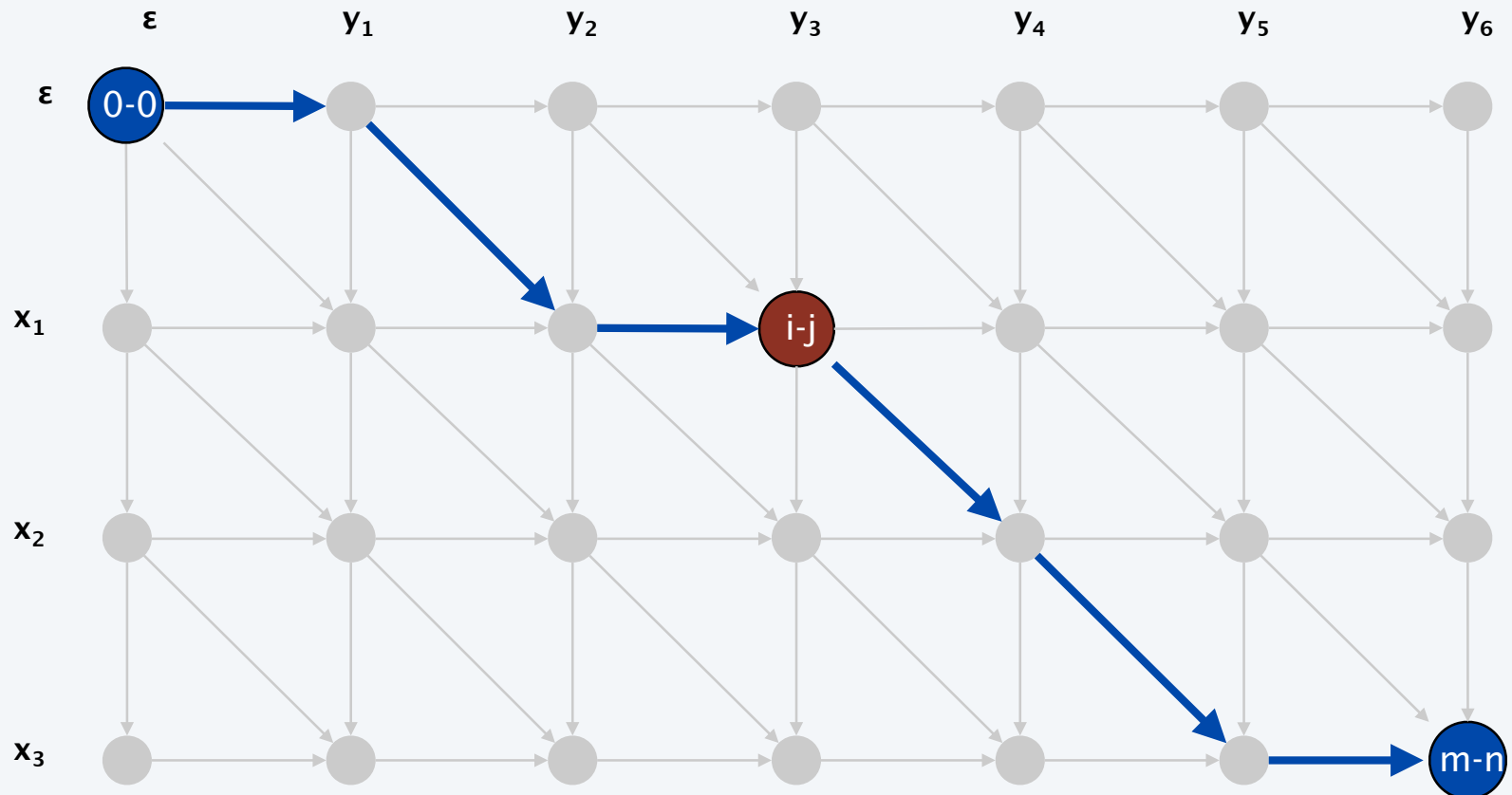
## Edit distance graph.

- Let  $g(i, j)$  be shortest path from  $(i, j)$  to  $(m, n)$ .
- Can compute  $g(\bullet, j)$  for any  $j$  in  $O(mn)$  time and  $O(m + n)$  space.



# Hirschberg's algorithm

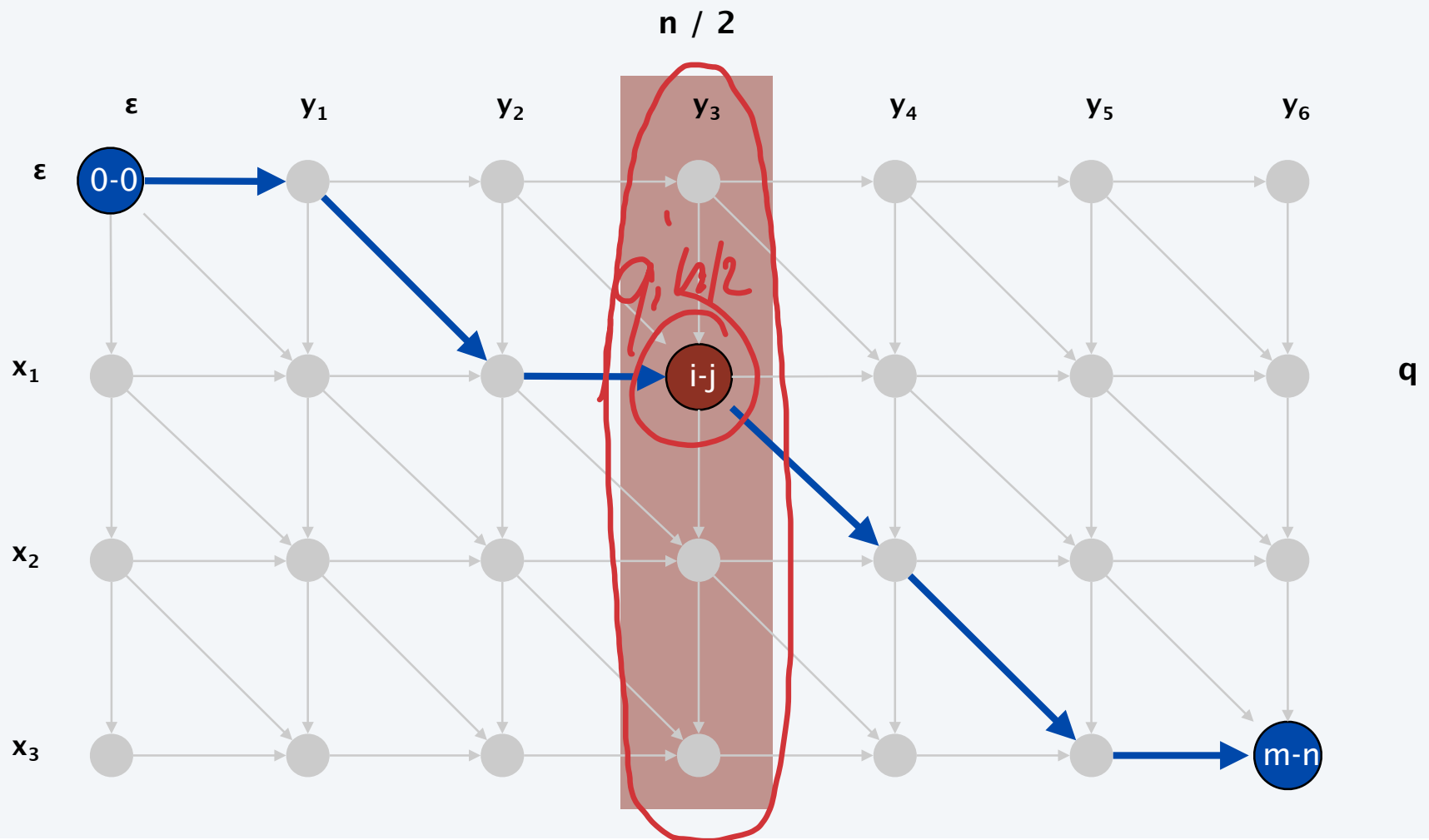
**Observation 1.** The cost of the shortest path that uses  $(i, j)$  is  $f(i, j) + g(i, j)$ .





# Hirschberg's algorithm

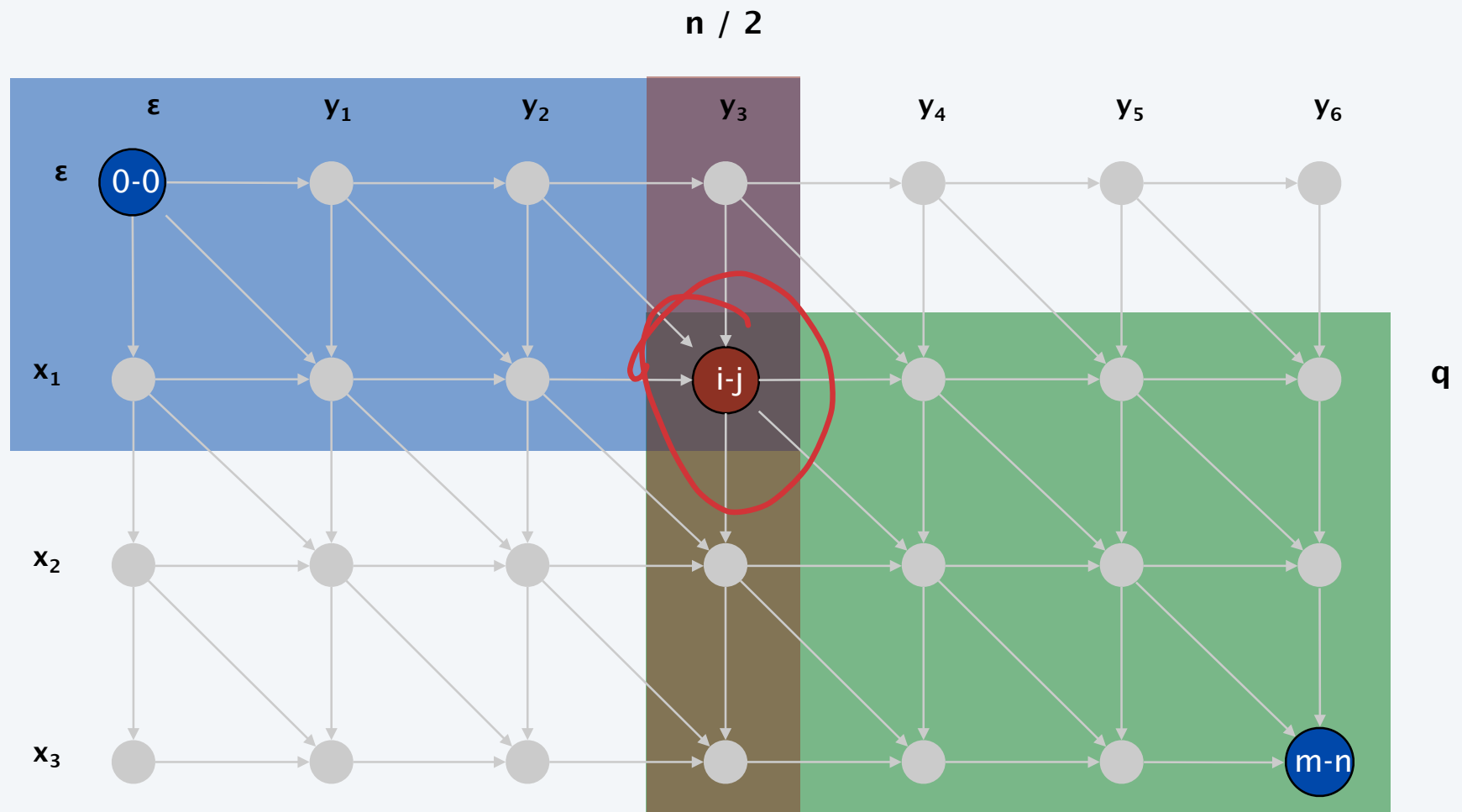
**Observation 2.** let  $q$  be an index that minimizes  $f(q, n/2) + g(q, n/2)$ . Then, there exists a shortest path from  $(0, 0)$  to  $(m, n)$  uses  $(q, n/2)$ .



# Hirschberg's algorithm

**Divide.** Find index  $q$  that minimizes  $f(q, n/2) + g(q, n/2)$ ; align  $x_q$  and  $y_{n/2}$ .

**Conquer.** Recursively compute optimal alignment in each piece.



# Hirschberg's algorithm: running time analysis warmup

---

**Theorem.** Let  $T(m, n)$  = max running time of Hirschberg's algorithm on strings of length at most  $m$  and  $n$ . Then,  $T(m, n) = O(m n \log n)$ .

**Pf.**  $T(m, n) \leq 2 T(m, n/2) + O(m n)$   
 $\Rightarrow T(m, n) = O(m n \log n)$ .

**Remark.** Analysis is not tight because two subproblems are of size  $(q, n/2)$  and  $(m - q, n/2)$ . In next slide, we save  $\log n$  factor.

# Hirschberg's algorithm: running time analysis

---

**Theorem.** Let  $T(m, n)$  = max running time of Hirschberg's algorithm on strings of length at most  $m$  and  $n$ . Then,  $T(m, n) = O(mn)$ .

**Pf.** [ by induction on  $n$  ]

- $O(mn)$  time to compute  $f(\bullet, n/2)$  and  $g(\bullet, n/2)$  and find index  $q$ .
- $T(q, n/2) + T(m - q, n/2)$  time for two recursive calls.

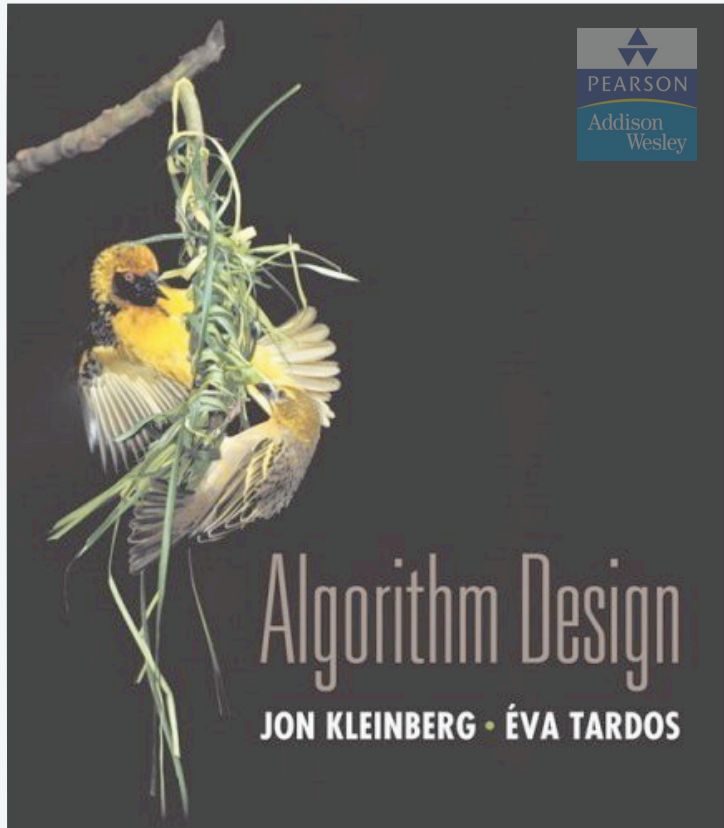
- Choose constant  $c$  so that:  $T(m, 2) \leq cm$

$$T(2, n) \leq cn$$

$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$

- Claim.  $T(m, n) \leq 2cmn$ .
- Base cases:  $m = 2$  or  $n = 2$ .
- Inductive hypothesis:  $T(m', n') \leq 2cm'n'$  for all  $(m', n')$  with  $m' + n' < m + n$ .

$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\ &\leq 2cq n/2 + 2c(m - q) n/2 + cmn \\ &= cq n + cmn - cq n + cmn \\ &= 2cmn \quad \blacksquare \end{aligned}$$



## SECTION 6.8

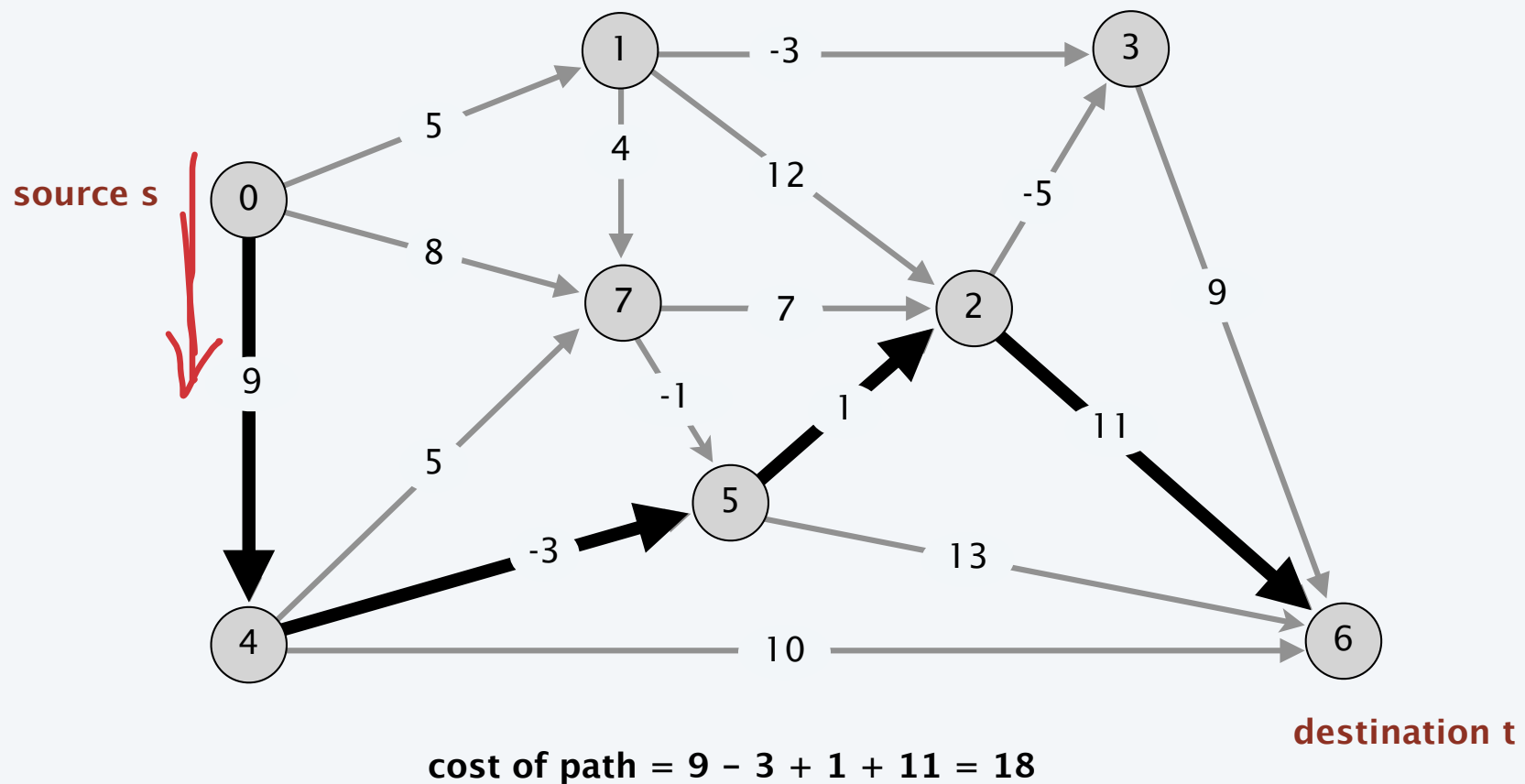
# 6. DYNAMIC PROGRAMMING II

---

- ▶ *sequence alignment*
- ▶ *Hirschberg's algorithm*
- ▶ ***Bellman-Ford***
- ▶ *distance vector protocols*
- ▶ *negative cycles in a digraph*

# Shortest paths

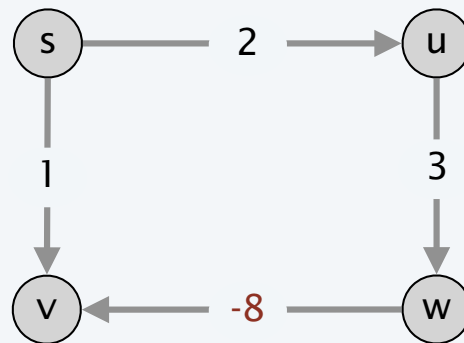
**Shortest path problem.** Given a digraph  $G = (V, E)$ , with arbitrary edge weights or costs  $c_{vw}$ , find cheapest path from node  $s$  to node  $t$ .



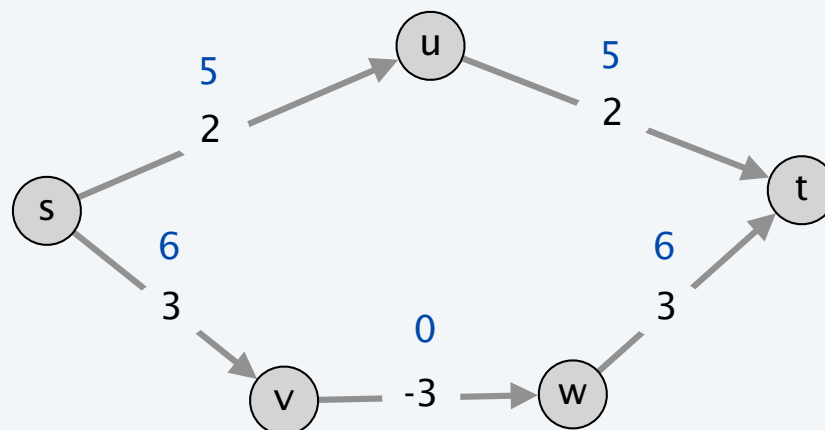
# Shortest paths: failed attempts

---

Dijkstra. Can fail if negative edge weights.



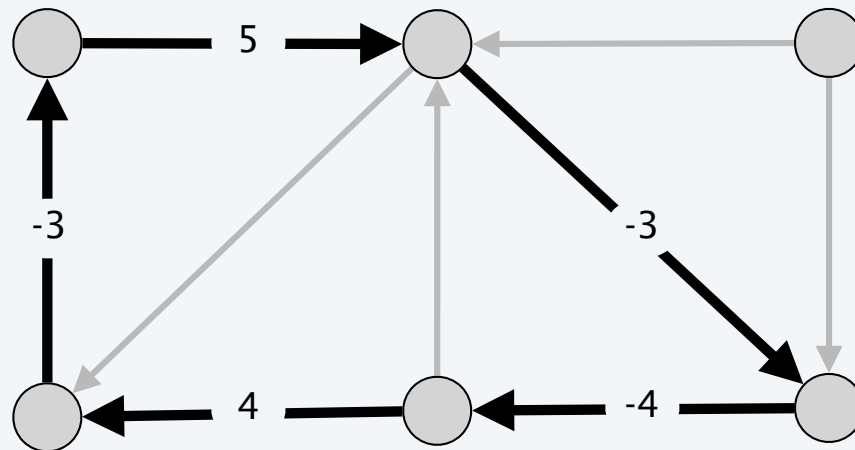
Reweighting. Adding a constant to every edge weight can fail.



# Negative cycles

---

**Def.** A **negative cycle** is a directed cycle such that the sum of its edge weights is negative.



a negative cycle  $W$  :  $c(W) = \sum_{e \in W} c_e < 0$

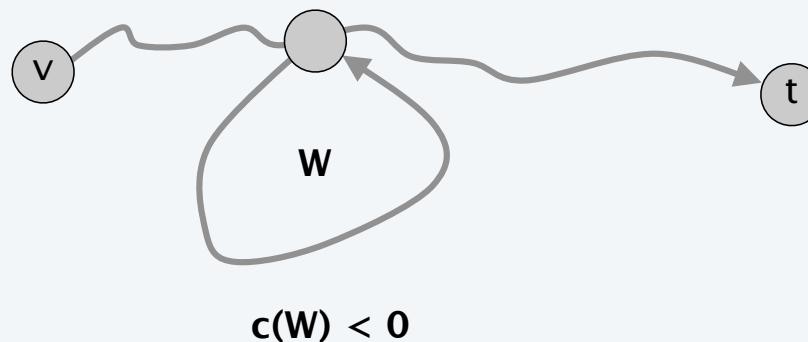


# Shortest paths and negative cycles

---

**Lemma 1.** If some path from  $v$  to  $t$  contains a negative cycle, then there does not exist a cheapest path from  $v$  to  $t$ .

**Pf.** If there exists such a cycle  $W$ , then can build a  $v \rightarrow t$  path of arbitrarily negative weight by detouring around cycle as many times as desired. ■



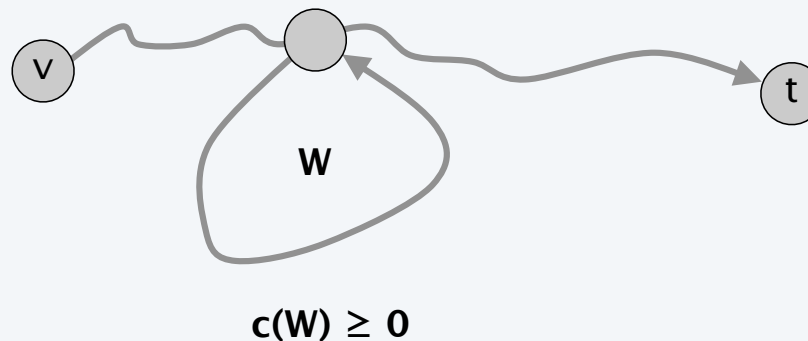
# Shortest paths and negative cycles

---

**Lemma 2.** If  $G$  has no negative cycles, then there exists a cheapest path from  $v$  to  $t$  that is simple (and has  $\leq n - 1$  edges).

**Pf.**

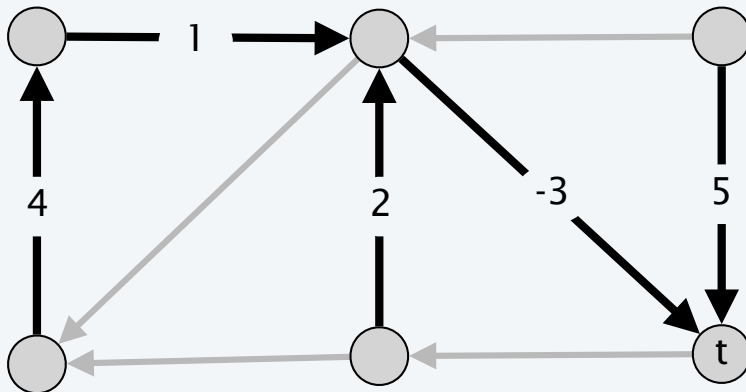
- Consider a cheapest  $v \rightarrow t$  path  $P$  that uses the fewest number of edges.
- If  $P$  contains a cycle  $W$ , can remove portion of  $P$  corresponding to  $W$  without increasing the cost. ■



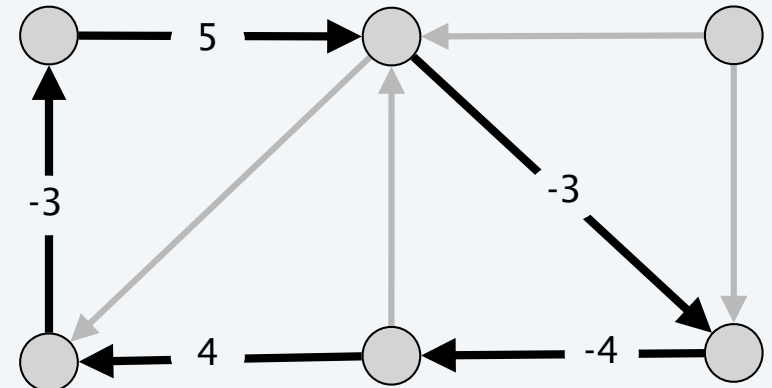
# Shortest path and negative cycle problems

**Shortest path problem.** Given a digraph  $G = (V, E)$  with edge weights  $c_{vw}$  and no negative cycles, find cheapest  $v \rightarrow t$  path for each node  $v$ .

**Negative cycle problem.** Given a digraph  $G = (V, E)$  with edge weights  $c_{vw}$ , find a negative cycle (if one exists).



shortest-paths tree




negative cycle

# Shortest paths: dynamic programming

---

**Def.**  $OPT(i, v)$  = cost of shortest  $v \rightarrow t$  path that uses  $\leq i$  edges.

- Case 1: Cheapest  $v \rightarrow t$  path uses  $\leq i - 1$  edges.
    - $OPT(i, v) = OPT(i - 1, v)$
  - Case 2: Cheapest  $v \rightarrow t$  path uses exactly  $i$  edges.
    - if  $(v, w)$  is first edge, then  $OPT$  uses  $(v, w)$ , and then selects best  $w \rightarrow t$  path using  $\leq i - 1$  edges
-  optimal substructure property  
(proof via exchange argument)

$$OPT(i, v) = \begin{cases} \infty & \text{if } i = 0 \\ \min \left\{ OPT(i-1, v), \min_{(v, w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

**Observation.** If no negative cycles,  $OPT(n - 1, v)$  = cost of cheapest  $v \rightarrow t$  path.

**Pf.** By Lemma 2, cheapest  $v \rightarrow t$  path is simple. ■

# Shortest paths: implementation

---

SHORTEST-PATHS ( $V, E, c, t$ )

---

FOREACH node  $v \in V$

$M[0, v] \leftarrow \infty.$

$M[0, t] \leftarrow 0.$

FOR  $i = 1$  TO  $n - 1$

FOREACH node  $v \in V$

$M[i, v] \leftarrow M[i - 1, v].$

FOREACH edge  $(v, w) \in E$

$M[i, v] \leftarrow \min \{ M[i, v], M[i - 1, w] + c_{vw} \}.$

---

## Shortest paths: implementation

---

**Theorem 1.** Given a digraph  $G = (V, E)$  with no negative cycles, the dynamic programming algorithm computes the cost of the cheapest  $v \rightarrow t$  path for each node  $v$  in  $\Theta(mn)$  time and  $\Theta(n^2)$  space.

**Pf.**

- Table requires  $\Theta(n^2)$  space.
- Each iteration  $i$  takes  $\Theta(m)$  time since we examine each edge once. ■

**Finding the shortest paths.**

- Approach 1: Maintain a  $successor(i, v)$  that points to next node on cheapest  $v \rightarrow t$  path using at most  $i$  edges.
- Approach 2: Compute optimal costs  $M[i, v]$  and consider only edges with  $M[i, v] = M[i - 1, w] + c_{vw}$ .

## Shortest paths: practical improvements

---

Space optimization. Maintain two 1d arrays (instead of 2d array).

- $d(v)$  = cost of cheapest  $v \rightarrow t$  path that we have found so far.
- $successor(v)$  = next node on a  $v \rightarrow t$  path.

Performance optimization. If  $d(w)$  was not updated in iteration  $i - 1$ , then no reason to consider edges entering  $w$  in iteration  $i$ .

# Bellman-Ford: efficient implementation

---

BELLMAN-FORD ( $V, E, c, t$ )

FOREACH node  $v \in V$

$d(v) \leftarrow \infty.$

$successor(v) \leftarrow null.$

$d(t) \leftarrow 0.$

FOR  $i = 1$  TO  $n - 1$

FOREACH node  $w \in V$

IF ( $d(w)$  was updated in previous iteration)

FOREACH edge  $(v, w) \in E$

                IF ( $d(v) > d(w) + c_{vw}$ )

$d(v) \leftarrow d(w) + c_{vw}.$

$successor(v) \leftarrow w.$

    IF no  $d(w)$  value changed in iteration  $i$ , STOP.

1 pass

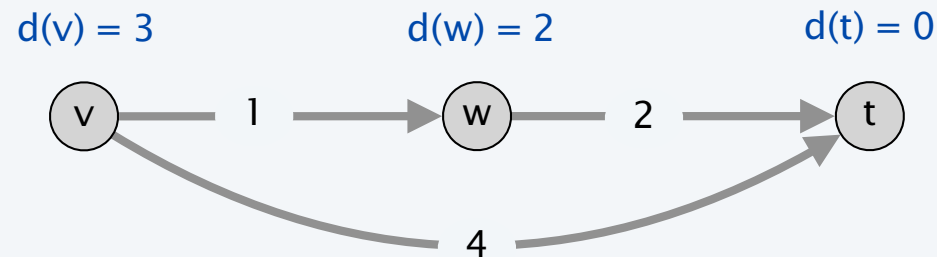


## Bellman-Ford: analysis

---

**Claim.** After the  $i^{\text{th}}$  pass of Bellman Ford,  $d(v)$  equals the cost of the cheapest  $v \rightarrow t$  path using at most  $i$  edges.

**Counterexample.** Claim is false!



if nodes  $w$  considered before node  $v$ ,  
then  $d(v) = 3$  after 1 pass

## Bellman-Ford: analysis

---

**Lemma 3.** Throughout Bellman-Ford algorithm,  $d(v)$  is the cost of some  $v \rightarrow t$  path; after the  $i^{\text{th}}$  pass,  $d(v)$  is no larger than the cost of the cheapest  $v \rightarrow t$  path using  $\leq i$  edges.

**Pf.** [by induction on  $i$ ]


- Assume true after  $i^{\text{th}}$  pass.
- Let  $P$  be any  $v \rightarrow t$  path with  $i + 1$  edges.
- Let  $(v, w)$  be first edge on path and let  $P'$  be subpath from  $w$  to  $t$ .
- By inductive hypothesis,  $d(w) \leq c(P')$  since  $P'$  is a  $w \rightarrow t$  path with  $i$  edges.
- After considering  $v$  in pass  $i+1$ :  $d(v) \leq c_{vw} + d(w)$

$$\leq c_{vw} + c(P')$$

$$= c(P) \quad \blacksquare$$

**Theorem 2.** Given a digraph with no negative cycles, Bellman-Ford computes the costs of the cheapest  $v \rightarrow t$  paths in  $O(mn)$  time and  $\Theta(n)$  extra space.

**Pf.** Lemmas 2 + 3.  $\blacksquare$

  
can be substantially  
faster in practice

# Bellman-Ford: analysis

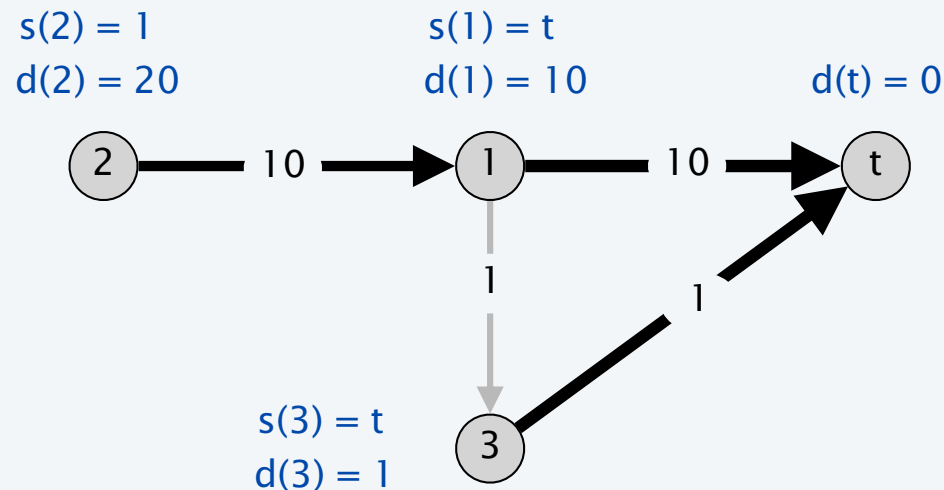
---

**Claim.** ~~Throughout the Bellman-Ford algorithm, following  $\text{successor}(v)$  pointers gives a directed path from  $v$  to  $t$  of cost  $d(v)$ .~~

**Counterexample.** Claim is false!

- Cost of successor  $v \rightarrow t$  path may have strictly lower cost than  $d(v)$ .

consider nodes in order:  $t, 1, 2, 3$



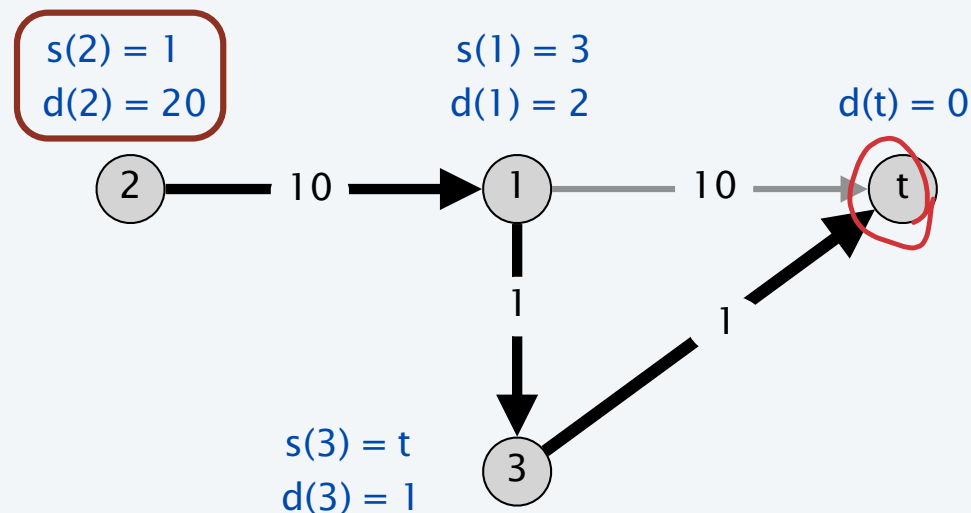
## Bellman-Ford: analysis

**Claim.** ~~Throughout the Bellman-Ford algorithm, following  $\text{successor}(v)$  pointers gives a directed path from  $v$  to  $t$  of cost  $d(v)$ .~~

**Counterexample.** Claim is false!

- Cost of successor  $v \rightarrow t$  path may have strictly lower cost than  $d(v)$ .

consider nodes in order:  $t, 1, 2, 3$



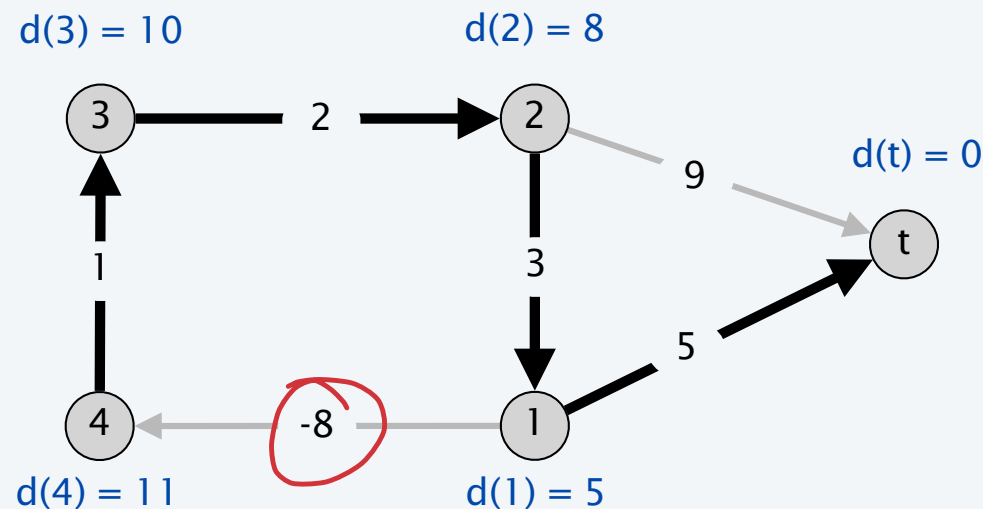
# Bellman-Ford: analysis

**Claim.** ~~Throughout the Bellman-Ford algorithm, following  $\text{successor}(v)$  pointers gives a directed path from  $v$  to  $t$  of cost  $d(v)$ .~~

**Counterexample.** Claim is false!

- Cost of successor  $v \rightarrow t$  path may have strictly lower cost than  $d(v)$ .
- Successor graph may have cycles.

consider nodes in order:  $t, 1, 2, 3, 4$



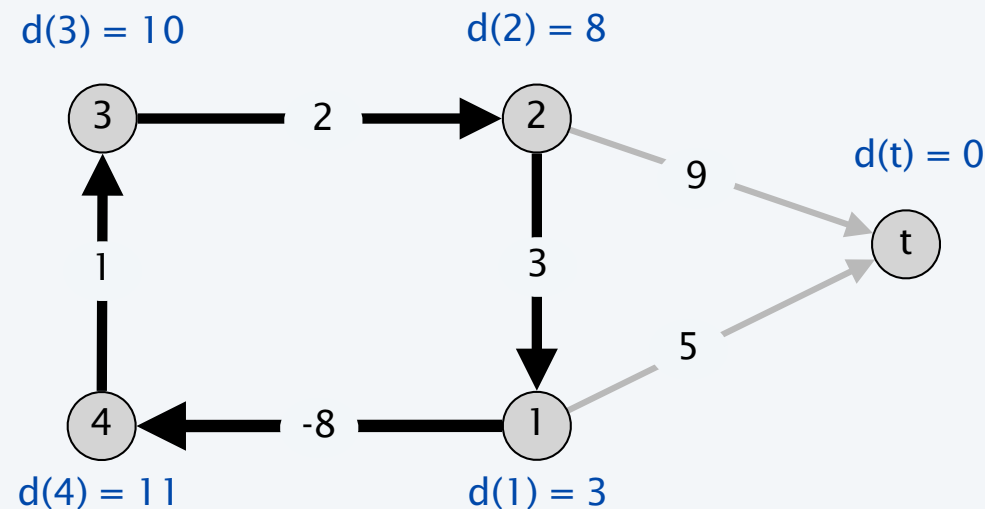
# Bellman-Ford: analysis

**Claim.** ~~Throughout the Bellman-Ford algorithm, following  $\text{successor}(v)$  pointers gives a directed path from  $v$  to  $t$  of cost  $d(v)$ .~~

**Counterexample.** Claim is false!

- Cost of successor  $v \rightarrow t$  path may have strictly lower cost than  $d(v)$ .
- Successor graph may have cycles.

consider nodes in order:  $t, 1, 2, 3, 4$




# Bellman-Ford: finding the shortest path

**Lemma 4.** If the successor graph contains a directed cycle  $W$ , then  $W$  is a negative cycle.

**Pf.**

- If  $successor(v) = w$ , we must have  $d(v) \geq d(w) + c_{vw}$ .  
(LHS and RHS are equal when  $successor(v)$  is set;  $d(w)$  can only decrease;  $d(v)$  decreases only when  $successor(v)$  is reset)
- Let  $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  be the nodes along the cycle  $W$ .
- Assume that  $(v_k, v_1)$  is the last edge added to the successor graph.
- Just prior to that:
$$\begin{array}{rcl} d(v_1) & \geq & d(v_2) + c(v_1, v_2) \\ d(v_2) & \geq & d(v_3) + c(v_2, v_3) \\ \vdots & & \vdots \\ d(v_{k-1}) & \geq & d(v_k) + c(v_{k-1}, v_k) \\ d(v_k) & > & d(v_1) + c(v_k, v_1) \end{array}$$

← holds with strict inequality since we are updating  $d(v_k)$
- Adding inequalities yields  $c(v_1, v_2) + c(v_2, v_3) + \dots + c(v_{k-1}, v_k) + c(v_k, v_1) < 0$ . ■  


$W$  is a negative cycle

# Bellman-Ford: finding the shortest path

**Theorem 3.** Given a digraph with no negative cycles, Bellman-Ford finds the cheapest  $s \rightarrow t$  paths in  $O(mn)$  time and  $\Theta(n)$  extra space.

**Pf.**

- The successor graph cannot have a negative cycle. [Lemma 4]
- Thus, following the successor pointers from  $s$  yields a directed path to  $t$ .
- Let  $s = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = t$  be the nodes along this path  $P$ .
- Upon termination, if  $\text{successor}(v) = w$ , we must have  $d(v) = d(w) + c_{vw}$ .  
(LHS and RHS are equal when  $\text{successor}(v)$  is set;  $d(\cdot)$  did not change)

- Thus, 
$$\begin{aligned} d(v_1) &= d(v_2) + c(v_1, v_2) \\ d(v_2) &= d(v_3) + c(v_2, v_3) \\ \vdots &\quad \quad \quad \vdots \\ d(v_{k-1}) &= d(v_k) + c(v_{k-1}, v_k) \end{aligned}$$

since algorithm  
terminated

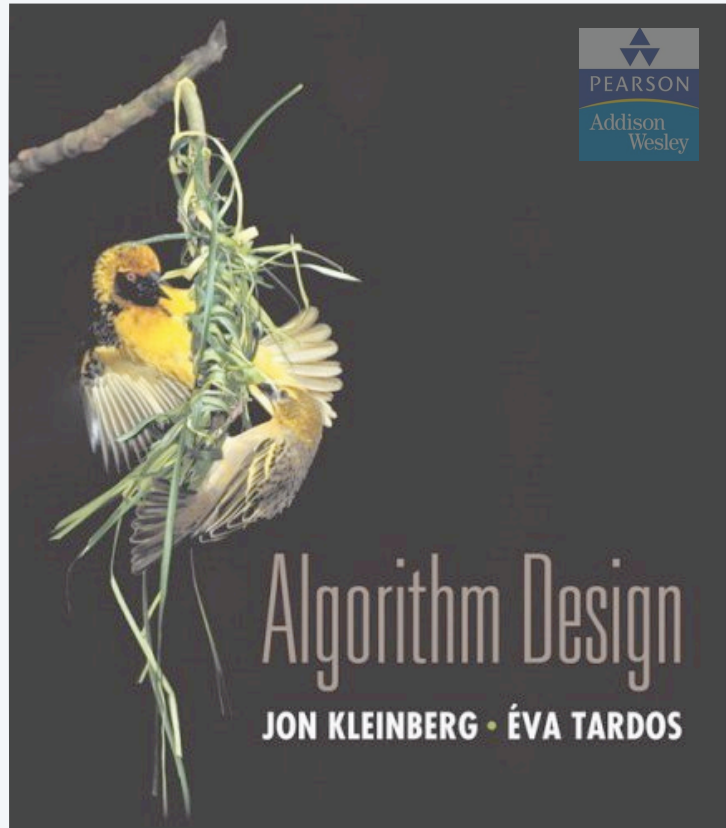
Adding equations yields  $d(s) = d(t) + c(v_1, v_2) + c(v_2, v_3) + \dots + c(v_{k-1}, v_k)$ . ■

min cost  
of any  $s \rightarrow t$  path  
(Theorem 2)

0

cost of path P





## SECTION 6.9

# 6. DYNAMIC PROGRAMMING II

---

- ▶ *sequence alignment*
- ▶ *Hirschberg's algorithm*
- ▶ *Bellman-Ford*
- ▶ *distance vector protocols*
- ▶ *negative cycles in a digraph*

# Distance vector protocols

---

## Communication network.

- Node  $\approx$  router.
- Edge  $\approx$  direct communication link.
- Cost of edge  $\approx$  delay on link.  $\longleftarrow$  naturally nonnegative, but Bellman-Ford used anyway!

**Dijkstra's algorithm.** Requires global information of network.

**Bellman-Ford.** Uses only local knowledge of neighboring nodes.

**Synchronization.** We don't expect routers to run in lockstep. The order in which each foreach loop executes is not important. Moreover, algorithm still converges even if updates are asynchronous.

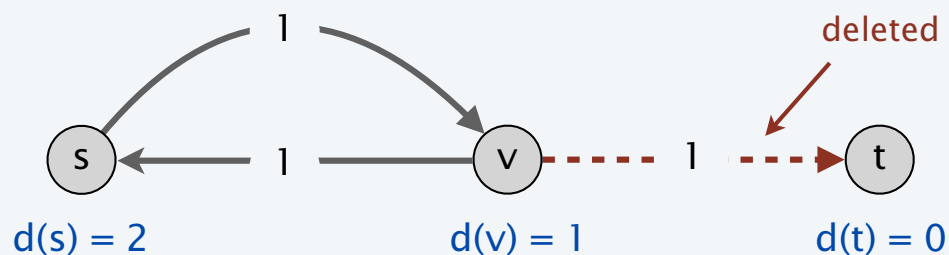
# Distance vector protocols

## Distance vector protocols. [ "routing by rumor" ]

- Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions).
- Algorithm: each router performs  $n$  separate computations, one for each potential destination node.

**Ex.** RIP, Xerox XNS RIP, Novell's IPX RIP, Cisco's IGRP, DEC's DNA Phase IV, AppleTalk's RTMP.

**Caveat.** Edge costs may **change** during algorithm (or fail completely).




"counting to infinity"

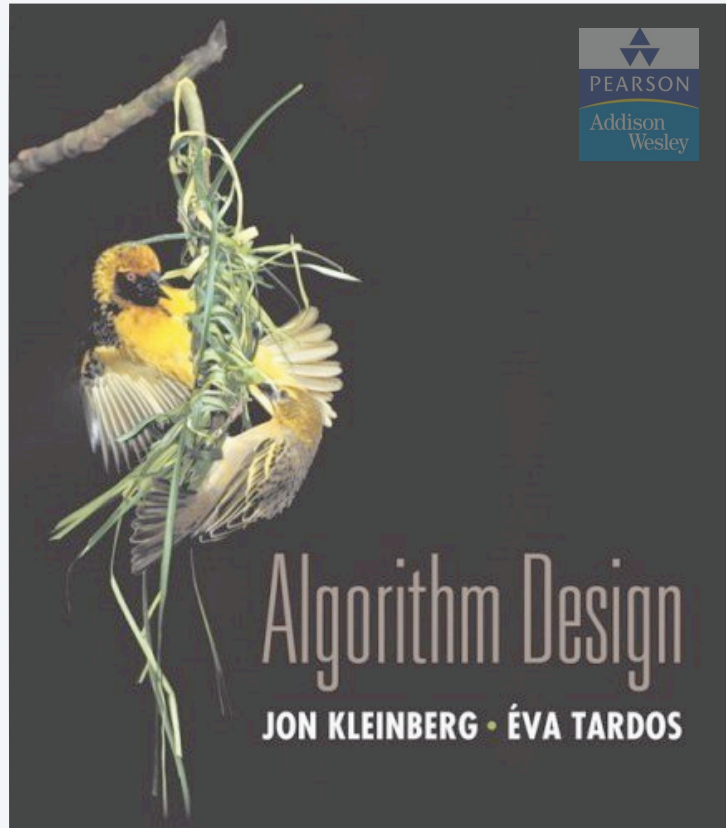
# Path vector protocols

---

## Link state routing.

- Each router also stores the entire path.  not just the distance and first hop
- Based on Dijkstra's algorithm.
- Avoids "counting-to-infinity" problem and related difficulties.
- Requires significantly more storage.

**Ex.** Border Gateway Protocol (BGP), Open Shortest Path First (OSPF).



## SECTION 6.10

# 6. DYNAMIC PROGRAMMING II

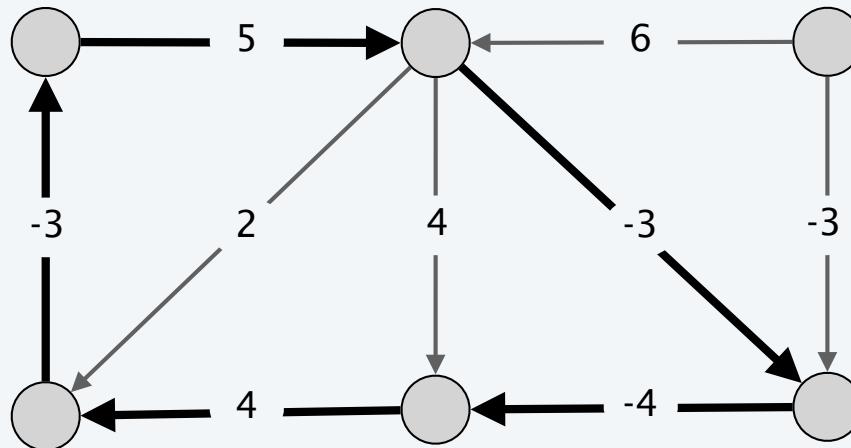
---

- ▶ *sequence alignment*
- ▶ *Hirschberg's algorithm*
- ▶ *Bellman-Ford*
- ▶ *distance vector protocol*
- ▶ *negative cycles in a digraph*

## Detecting negative cycles

---

**Negative cycle detection problem.** Given a digraph  $G = (V, E)$ , with edge weights  $c_{vw}$ , find a negative cycle (if one exists).

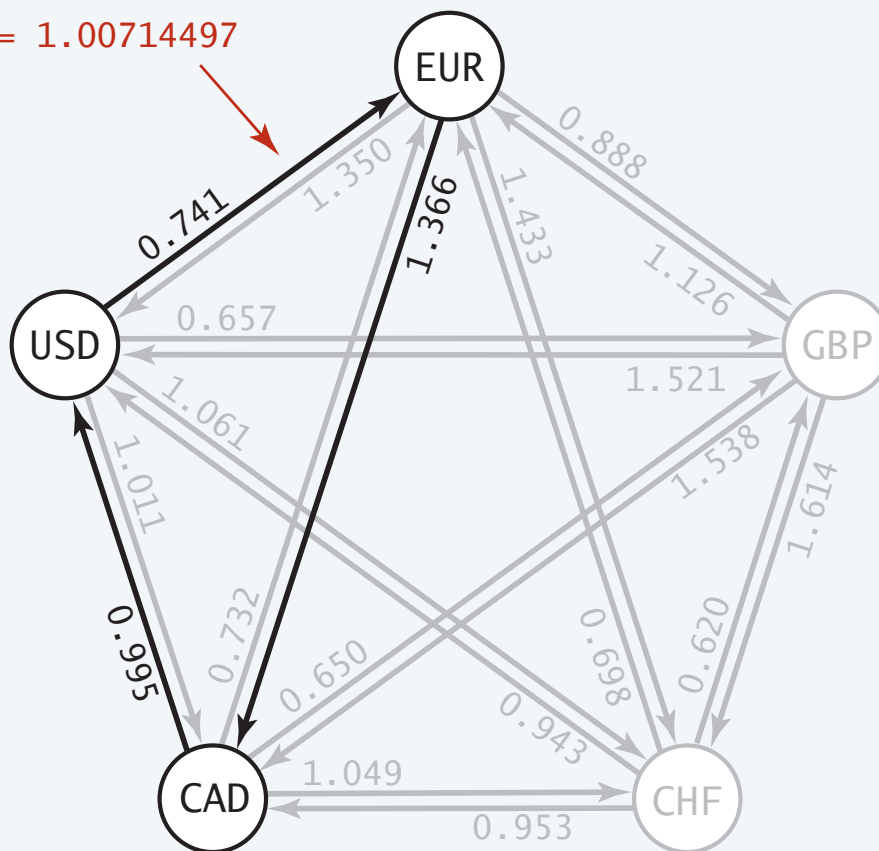


## Detecting negative cycles: application

**Currency conversion.** Given  $n$  currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

**Remark.** Fastest algorithm very valuable!

$$0.741 * 1.366 * .995 = 1.00714497$$



# Detecting negative cycles

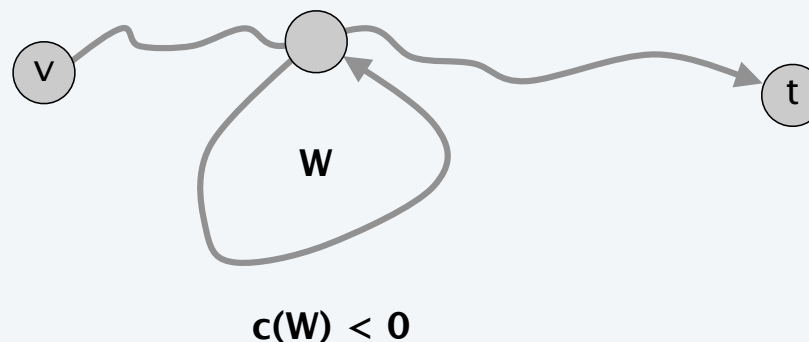
**Lemma 5.** If  $OPT(n, v) = OPT(n - 1, v)$  for all  $v$ , then no negative cycle can reach  $t$ .

**Pf.** Bellman-Ford algorithm. ▀

**Lemma 6.** If  $OPT(n, v) < OPT(n - 1, v)$  for some node  $v$ , then (any) cheapest path from  $v$  to  $t$  contains a cycle  $W$ . Moreover  $W$  is a negative cycle.

**Pf.** [by contradiction]

- Since  $OPT(n, v) < OPT(n - 1, v)$ , we know that shortest  $v \rightarrow t$  path  $P$  has exactly  $n$  edges.
- By pigeonhole principle,  $P$  must contain a directed cycle  $W$ .
- Deleting  $W$  yields a  $v \rightarrow t$  path with  $< n$  edges  $\Rightarrow W$  has negative cost. ▀

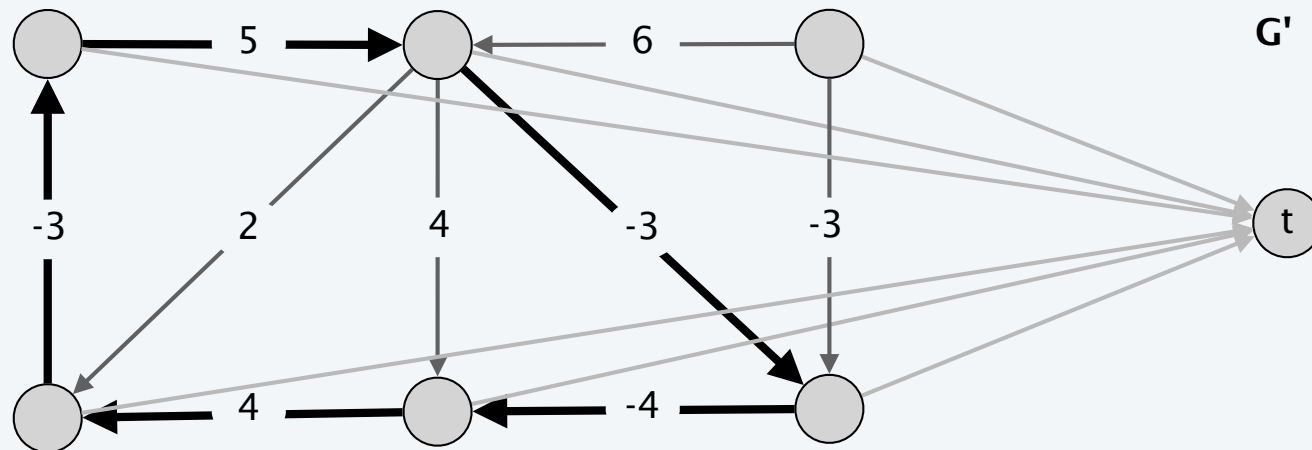




## Detecting negative cycles

**Theorem 4.** Can find a negative cycle in  $\Theta(mn)$  time and  $\Theta(n^2)$  space.  
**Pf.**

- Add new node  $t$  and connect all nodes to  $t$  with 0-cost edge.
- $G$  has a negative cycle iff  $G'$  has a negative cycle than can reach  $t$ .
- If  $OPT(n, v) = OPT(n - 1, v)$  for all nodes  $v$ , then no negative cycles.
- If not, then extract directed cycle from path from  $v$  to  $t$ .  
(cycle cannot contain  $t$  since no edges leave  $t$ ) ■



## Detecting negative cycles

---

**Theorem 5.** Can find a negative cycle in  $O(mn)$  time and  $O(n)$  extra space.  
**Pf.**

- Run Bellman-Ford for  $n$  passes (instead of  $n - 1$ ) on modified digraph.
- If no  $d(v)$  values updated in pass  $n$ , then no negative cycles.
- Otherwise, suppose  $d(s)$  updated in pass  $n$ .
- Define  $pass(v)$  = last pass in which  $d(v)$  was updated.
- Observe  $pass(s) = n$  and  $pass(successor(v)) \geq pass(v) - 1$  for each  $v$ .
- Following successor pointers, we must eventually repeat a node.
- Lemma 4  $\Rightarrow$  this cycle is a negative cycle.   ▪

**Remark.** See p. 304 for improved version and early termination rule.  
(Tarjan's subtree disassembly trick)