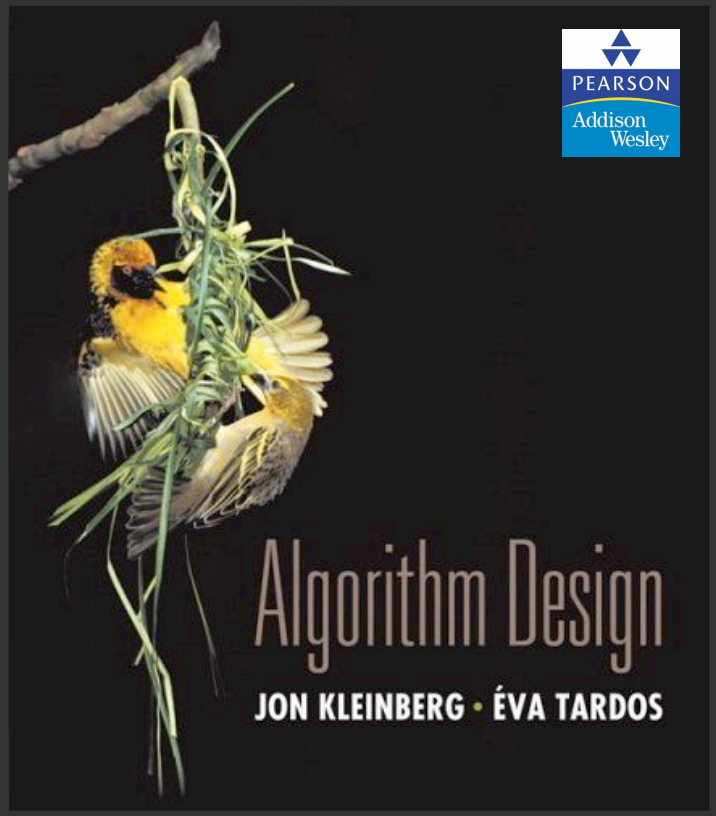


6. Dynamic Programming I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Lecture slides by Kevin Wayne

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

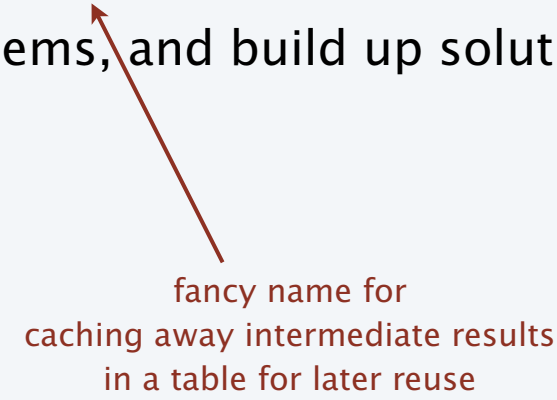
Last updated on Sep 8, 2013 6:33 AM

Algorithmic paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.



Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

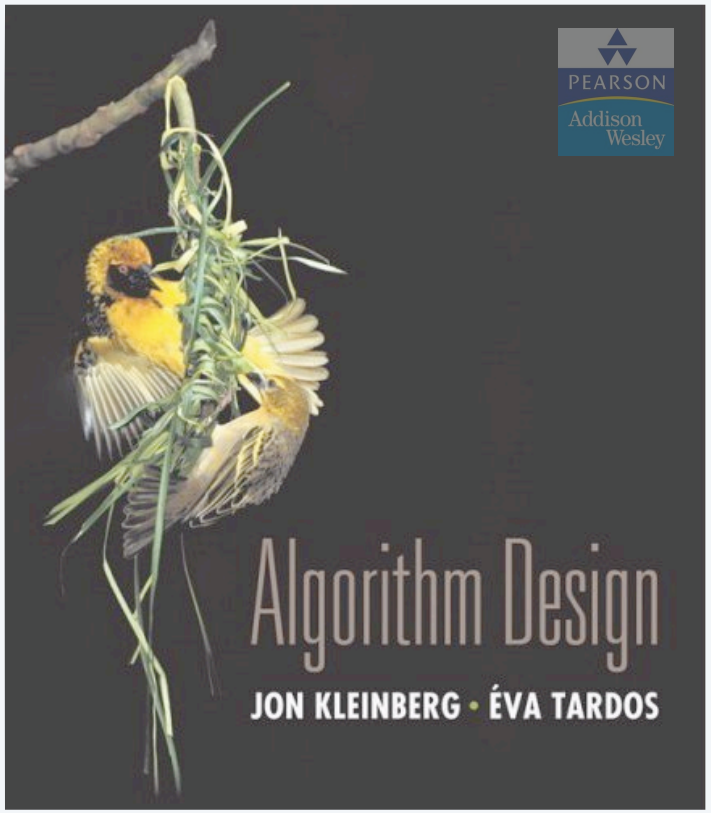
Dynamic programming applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ...
- ...

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.
- ...



SECTION 6.1–6.2

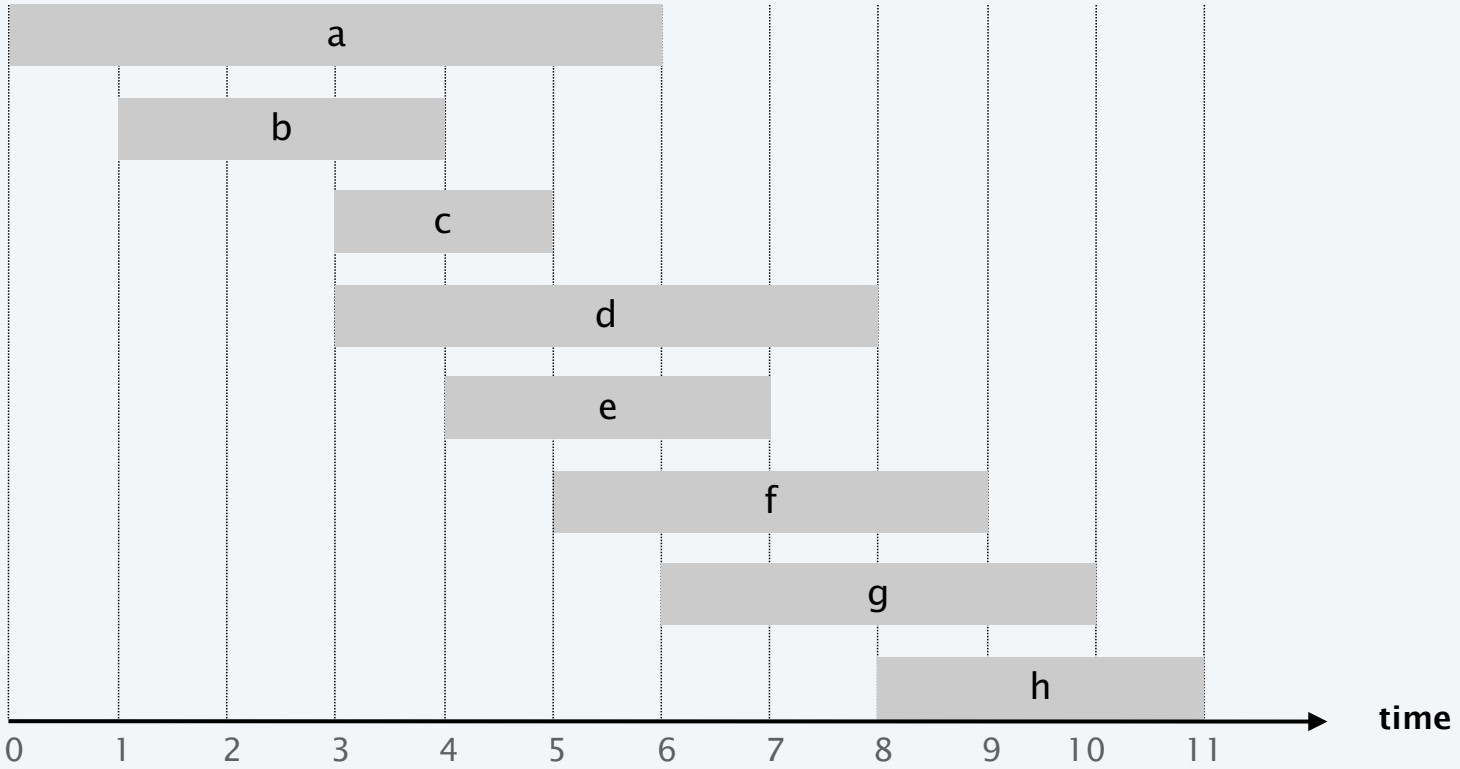
6. DYNAMIC PROGRAMMING I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Weighted interval scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



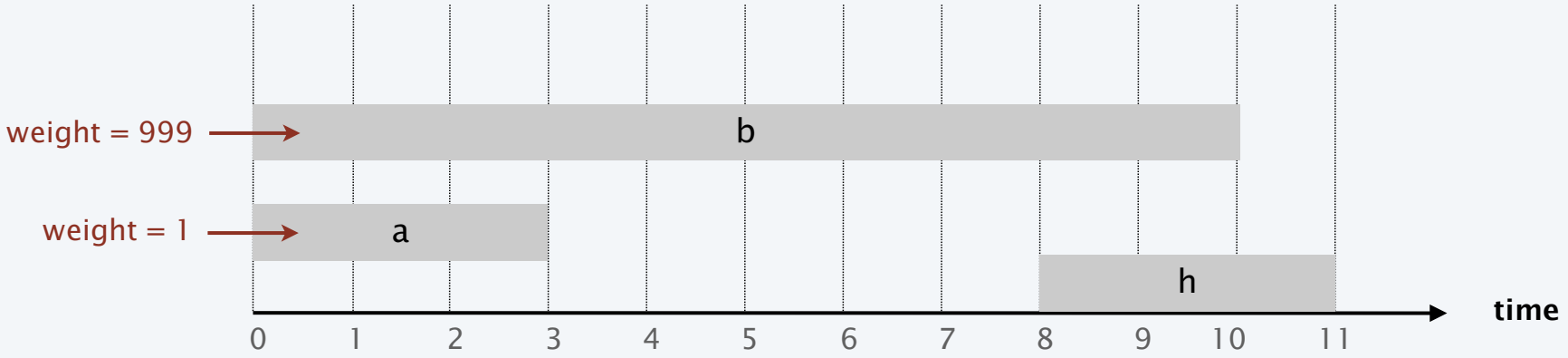
Earliest-finish-time first algorithm

Earliest finish-time first.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.

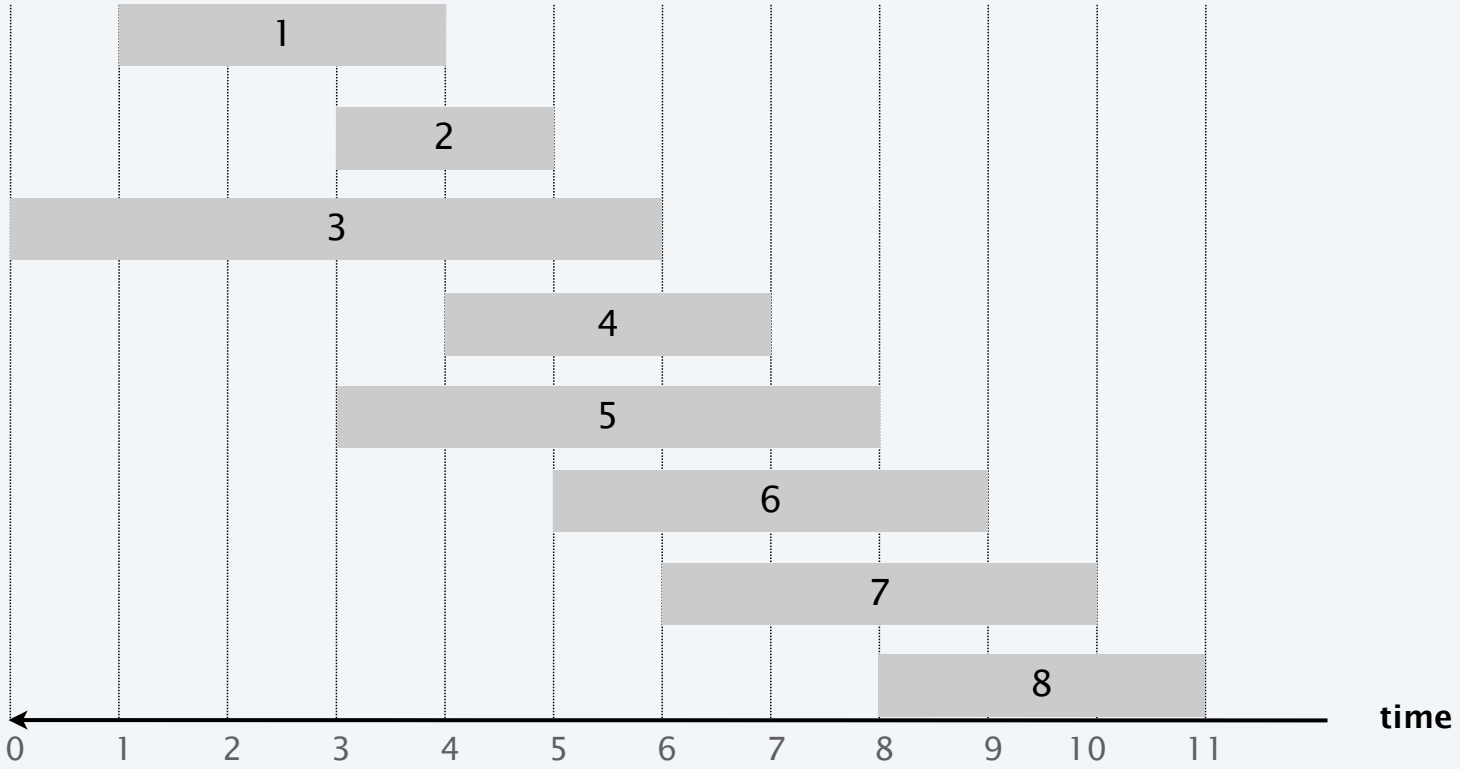


Weighted interval scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex. $p(8) = 5, p(7) = 3, p(2) = 0.$



Dynamic programming: binary choice

Notation. $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \ldots, j$.

Case 1. OPT selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, \ldots, j - 1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT does not select job j.

- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, j - 1$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted interval scheduling: brute force

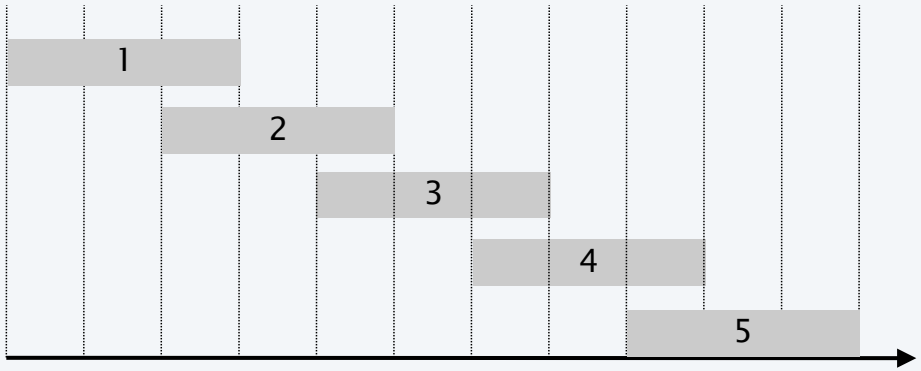
```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

Compute-Opt(j)
if j = 0
   return 0.
else
   return max(v[j] + Compute-Opt(p[j], Compute-Opt(j-1))).
```

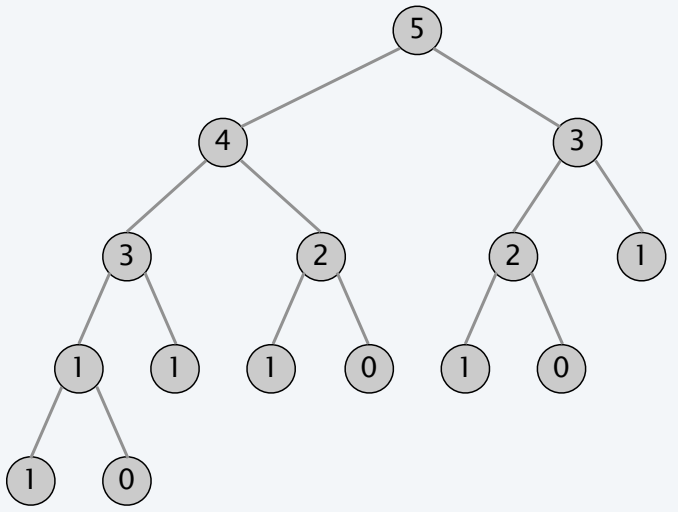
Weighted interval scheduling: brute force

Observation. Recursive algorithm fails spectacularly because of redundant subproblems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



recursion tree

Weighted interval scheduling: memoization

Memoization. Cache results of each subproblem; lookup as needed.

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
-----------------------------------------------------------
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j - 1)).
return M[j].
```

Weighted interval scheduling: running time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- M-COMPUTE-OPT(j): each invocation takes $O(1)$ time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure $\Phi = \#$ nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by $1 \Rightarrow$ at most $2n$ recursive calls.

- Overall running time of M-COMPUTE-OPT(n) is $O(n)$. ▪

Remark. $O(n)$ if jobs are presorted by start and finish times.

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass.

```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return { j } ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

Analysis. # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted interval scheduling: bottom-up

Bottom-up dynamic programming. Unwind recursion.

BOTTOM-UP $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

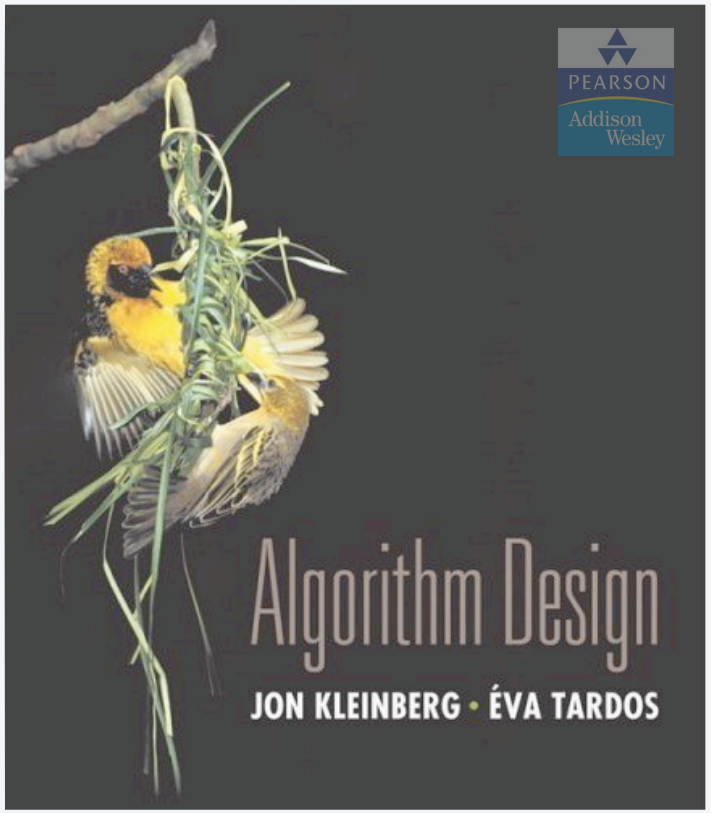
Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0$.

FOR j = 1 TO n

$M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \}$.



SECTION 6.3

6. DYNAMIC PROGRAMMING I

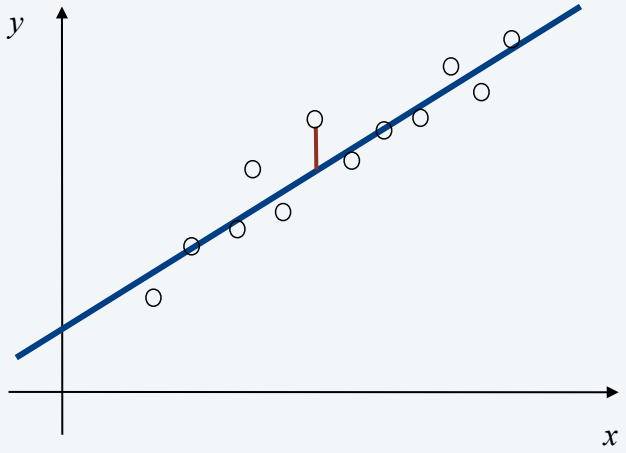
- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Least squares

Least squares. Foundational problem in statistics.

- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Solution. Calculus $\Rightarrow$ min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented least squares

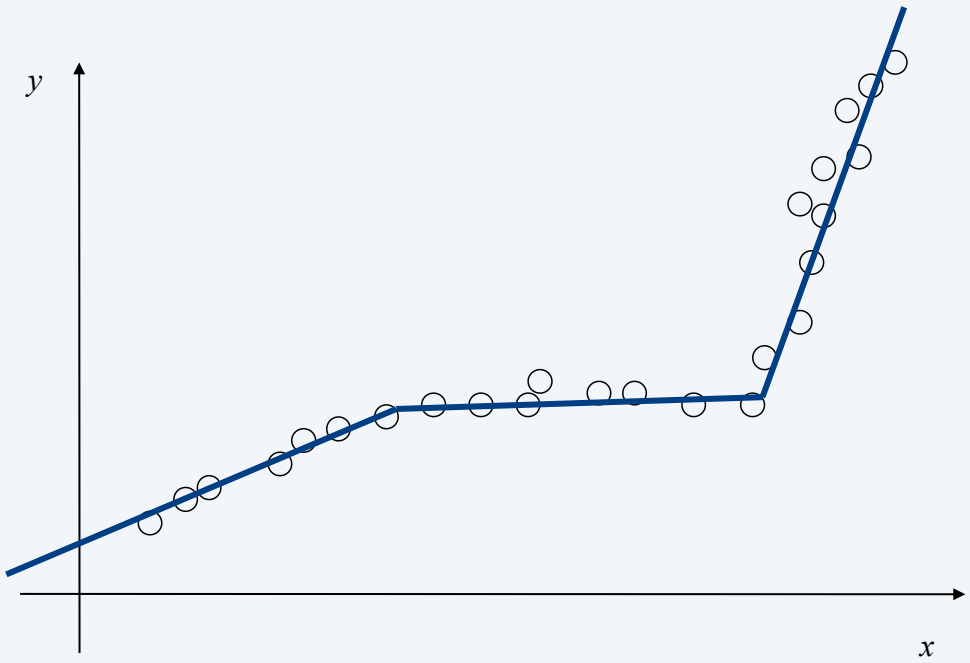
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

Q. What is a reasonable choice for $f(x)$ to balance accuracy and parsimony?

goodness of fit

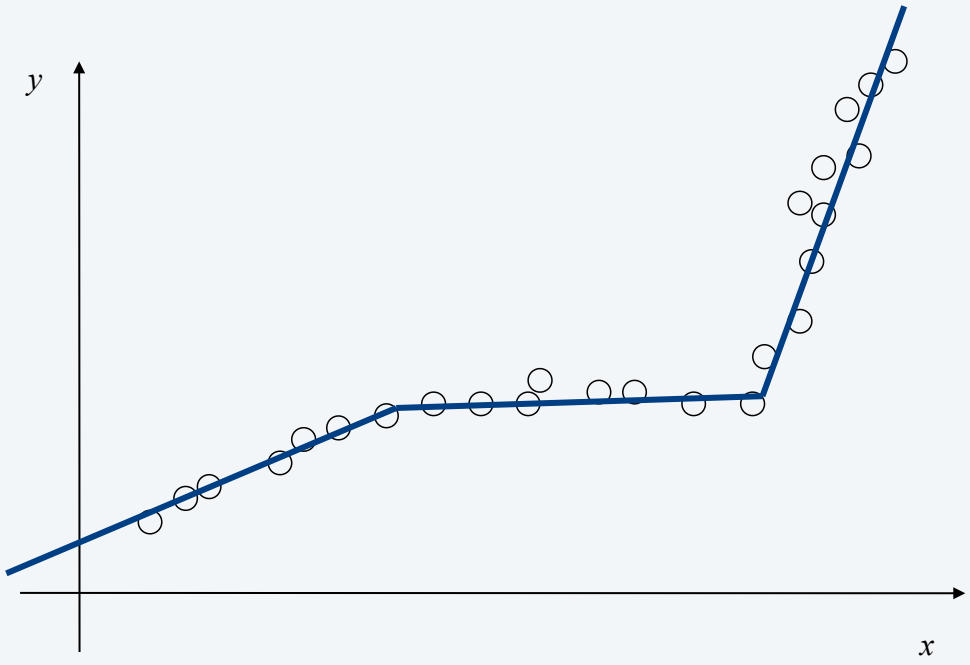
number of lines



Segmented least squares

Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$ and a constant $c > 0$, find a sequence of lines that minimizes $f(x) = E + c\,L$:

- E = the sum of the sums of the squared errors in each segment.
- L = the number of lines.



Dynamic programming: multiway choice

Notation.

- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i, j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute $OPT(j)$:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
- Cost = $e(i, j) + c + OPT(i-1)$. ← optimal substructure property (proof via exchange argument)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \left\{ e(i,j) + c + OPT(i-1) \right\} & \text{otherwise} \end{cases}$$

Segmented least squares algorithm

SEGMENTED-LEAST-SQUARES $(n,\ p_1,\ \ldots,\ p_n\ ,\ c)$

FOR j = 1 TO n

  FOR i = 1 TO j

    Compute the least squares $e(i, j)$ for the segment $p_i, p_{i+1}, \ldots, p_j$.

$M[0] \leftarrow 0.$

FOR $j = 1$ TO n

  $M[j] \leftarrow \min_{1 \le i \le j} \{ e_{ij} + c + M[i-1] \}.$

RETURN $M[n]$.

Segmented least squares analysis

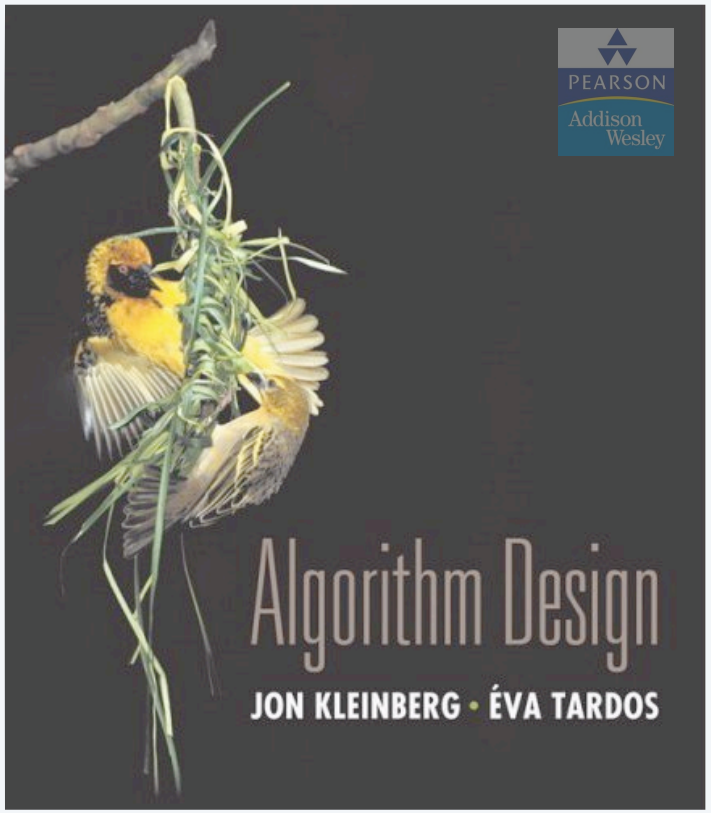
Theorem. [Bellman 1961] The dynamic programming algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

Pf.

- Bottleneck = computing $e(i, j)$ for $O(n^2)$ pairs.
- $O(n)$ per pair using formula. ▪

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Remark. Can be improved to $O(n^2)$ time and $O(n)$ space by precomputing various statistics. How?



SECTION 6.4

6. DYNAMIC PROGRAMMING I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Knapsack problem

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

Ex. $\{ 1, 2, 5 \}$ has value 35.

Ex. $\{ 3, 4 \}$ has value 40.

Ex. $\{ 3, 5 \}$ has value 46 (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

Greedy by value. Repeatedly add item with maximum v_i.

Greedy by weight. Repeatedly add item with minimum w_i.

Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i.

Observation. None of greedy algorithms is optimal.

Dynamic programming: false start

Def. $OPT(i)$ = max profit subset of items $1, \ldots, i$.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before i, we don't even know if we have enough room for i.

Conclusion. Need more subproblems!

Dynamic programming: adding a new variable

Def. $OPT(i, w) =$ max profit subset of items $1, \ldots, i$ with weight limit w.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit w.

Case 2. OPT selects item i.

- New weight limit $= w - w_i$.
- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using this new weight limit.

optimal substructure property
(proof via exchange argument)

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ \ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

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

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$.

RETURN $M[n, W]$.

Knapsack problem: bottom-up demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i

	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

Knapsack problem: running time

Theorem. There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(n\,W)$ time and $\Theta(n\,W)$ space.

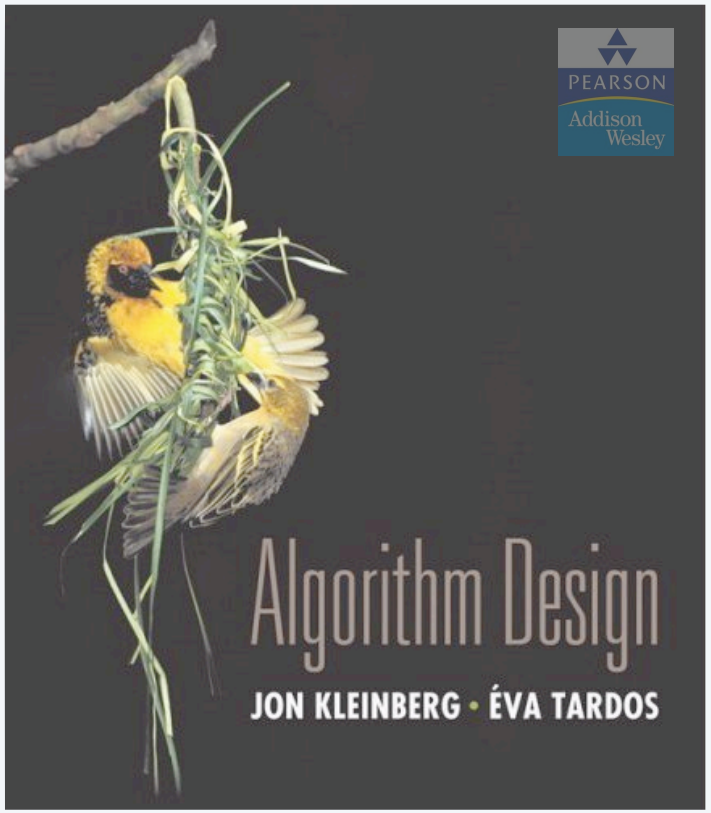
weights are integers between 1 and W

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\,W)$ table entries.
- After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] < M[i-1, w]$. ▪

Remarks.

- Not polynomial in input size! ← "pseudo-polynomial"
- Decision version of knapsack problem is NP-COMPLETE. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]



SECTION 6.5

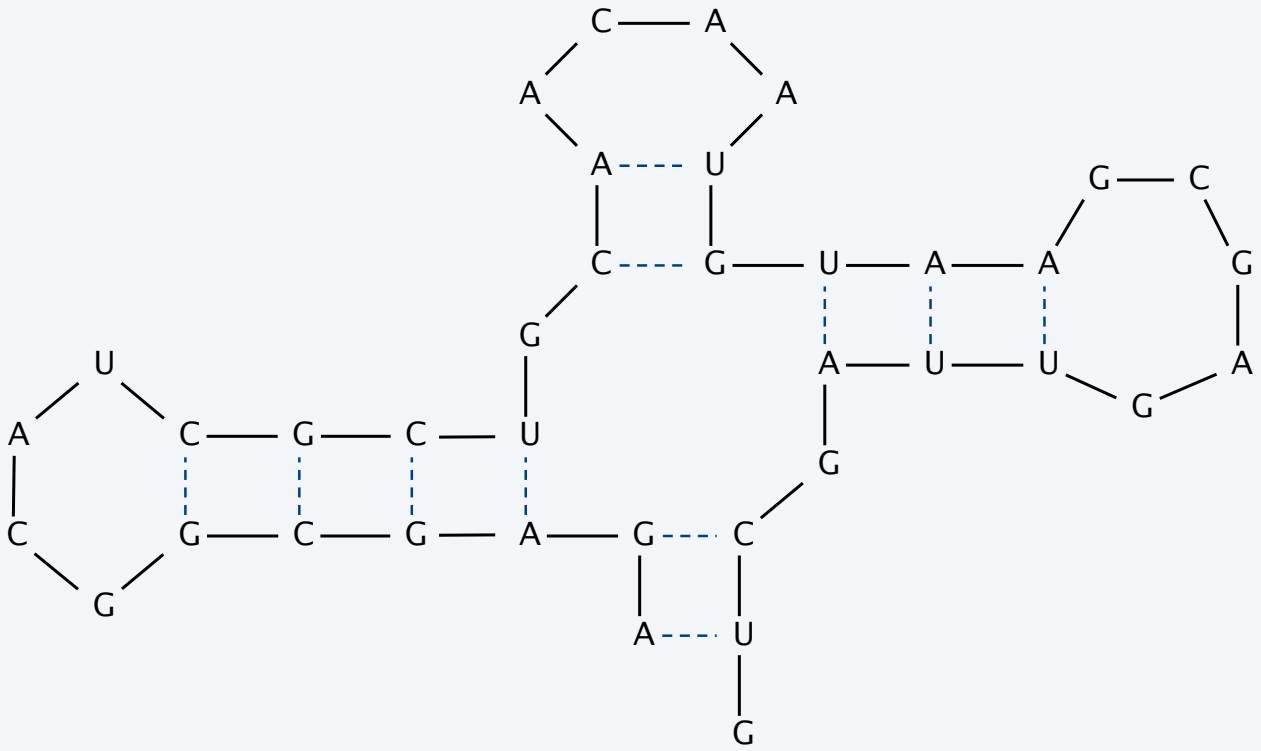
6. DYNAMIC PROGRAMMING I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

RNA secondary structure

RNA. String $B = b_1 b_2 \ldots b_n$ over alphabet $\{ \mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{U} \}$.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.



RNA secondary structure for GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

RNA secondary structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.

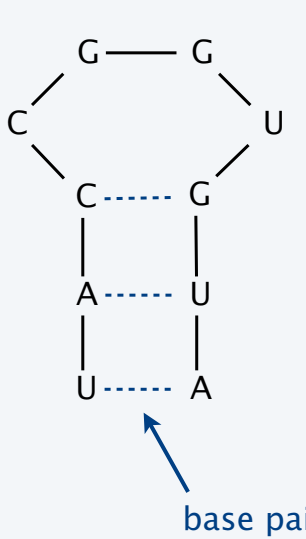
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy.

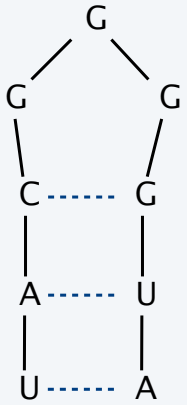
approximate by number of base pairs

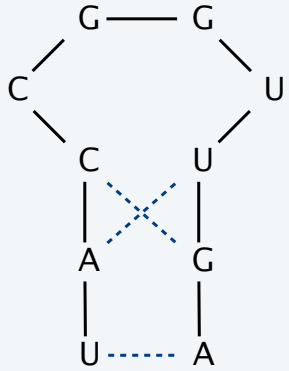
Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

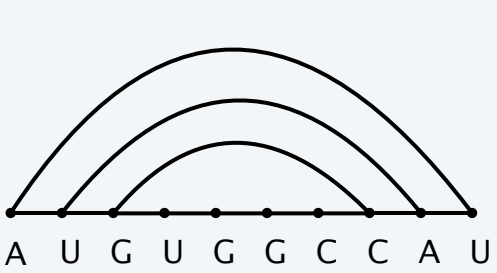
RNA secondary structure

Examples.

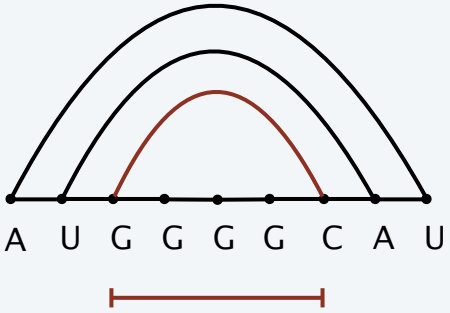




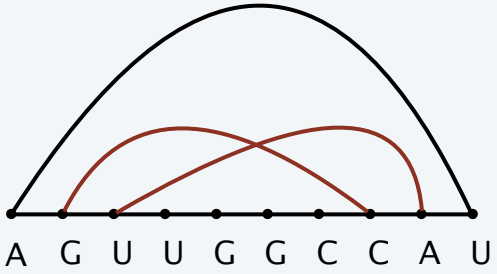




ok



sharp turn
(≤4 intervening bases)

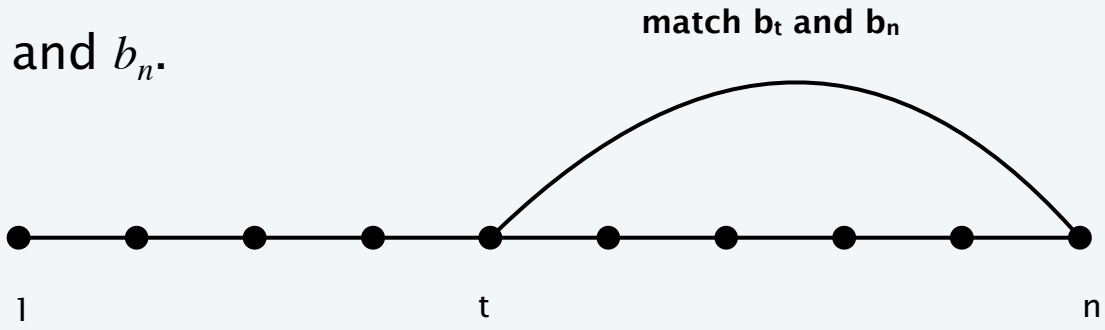


crossing

RNA secondary structure: subproblems

First attempt. $OPT(j) =$ maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_j$.

Choice. Match b_t and b_n.



Difficulty. Results in two subproblems but one of wrong form.

- Find secondary structure in $b_1 b_2 \ldots b_{t-1}$. ← OPT(t−1)
- Find secondary structure in $b_{t+1} b_{t+2} \ldots b_{n-1}$. ← need more subproblems

Dynamic programming over intervals

Notation. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Case 1. If $i \geq j - 4$.

- $OPT(i, j) = 0$ by no-sharp turns condition.

Case 2. Base b_j is not involved in a pair.

- $OPT(i, j) = \text{OPT}(i, j-1)$.

Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.

- Noncrossing constraint decouples resulting subproblems.
- $\text{OPT}(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$.

take max over t such that $i \leq t < j - 4$ and b_t and b_j are Watson-Crick complements

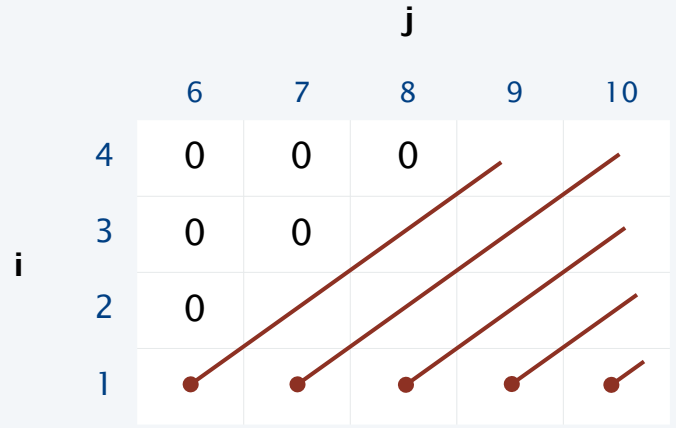
Bottom-up dynamic programming over intervals

Q. In which order to solve the subproblems?

A. Do shortest intervals first.

```
RNA (n, b_1, ..., b_n)

FOR k = 5 TO n – 1
    FOR i = 1 TO n – k
        j ← i + k.
        Compute M[i, j] using formula.
RETURN M[1, n].
```



order in which to solve subproblems

Theorem. The dynamic programming algorithm solves the RNA secondary substructure problem in $O(n^3)$ time and $O(n^2)$ space.

Dynamic programming summary

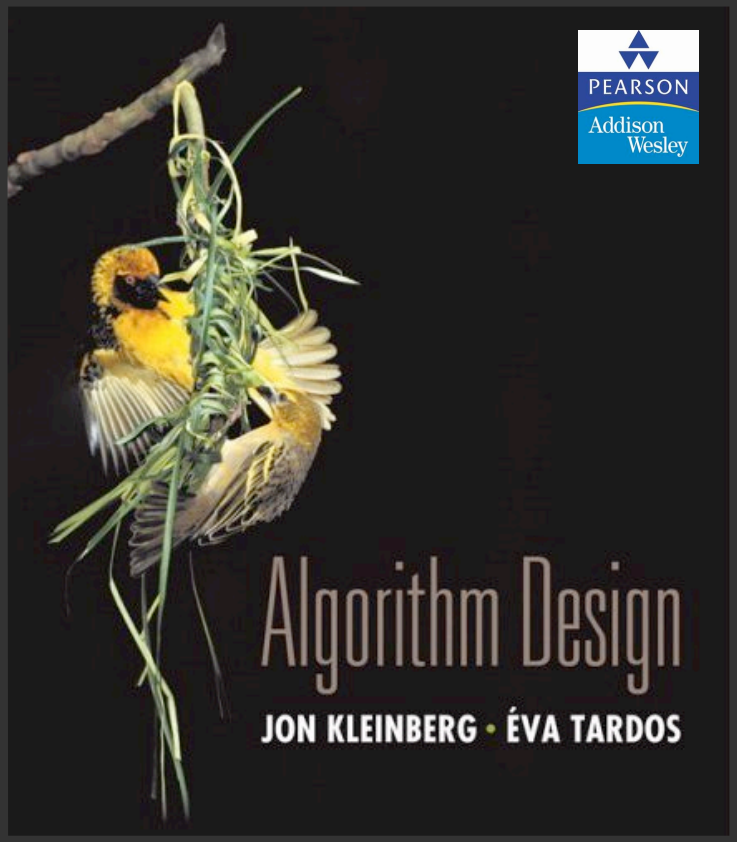
Outline.

- Polynomial number of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from smallest to largest, with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solution to smaller subproblems.

Techniques.

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Adding a new variable: knapsack problem.
- Dynamic programming over intervals: RNA secondary structure.

Top-down vs. bottom-up. Different people have different intuitions.



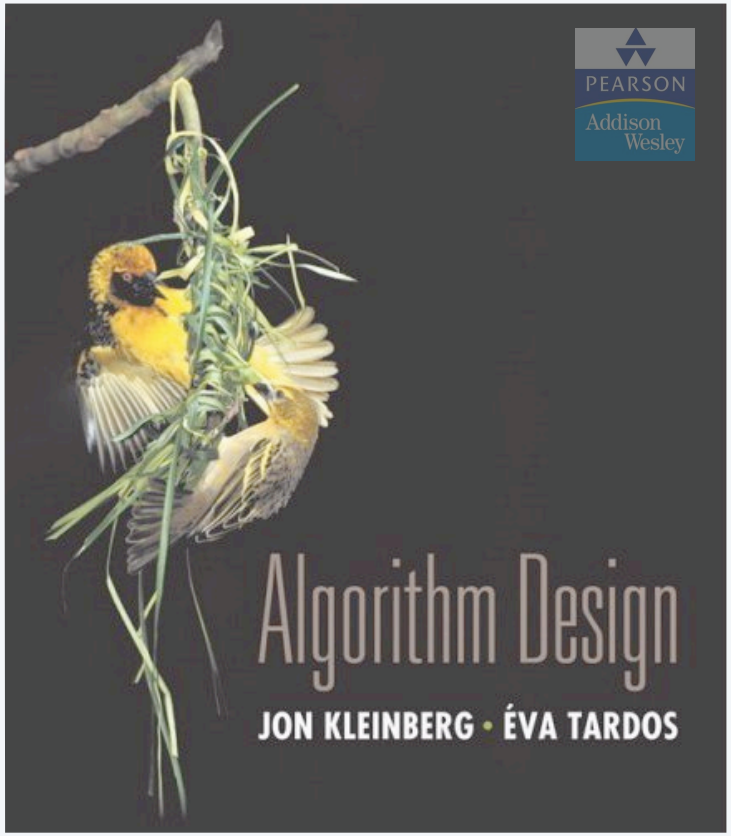
6. Dynamic Programming II

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

Lecture slides by Kevin Wayne

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

Last updated on Mar 14, 2014, 3:49 PM



SECTION 6.6

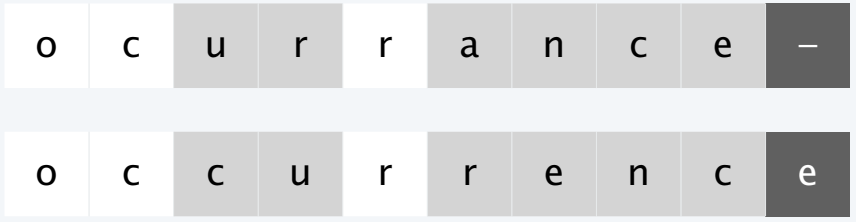
6. DYNAMIC PROGRAMMING II

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

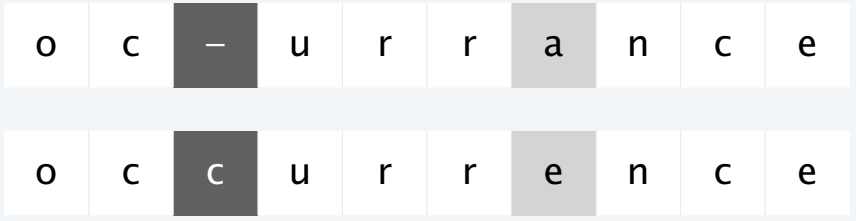
String similarity

Q. How similar are two strings?

Ex. ocurrance and occurrence.



6 mismatches, 1 gap



1 mismatch, 1 gap

0 mismatches, 3 gaps

Edit distance

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

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

cost = δ + α_{CG} + α_{TA}

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

Sequence alignment

Goal. Given two strings $x_1 x_2 \ldots x_m$ and $y_1 y_2 \ldots y_n$ 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:

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

Sequence alignment: problem structure

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

Case 1. OPT matches $x_i - y_j$.

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

Case 2a. OPT leaves x_i unmatched.

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

Case 2b. OPT leaves y_j unmatched.

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

optimal substructure property
(proof via exchange argument)

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

Sequence alignment: algorithm

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

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 $M[m, n]$.

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.

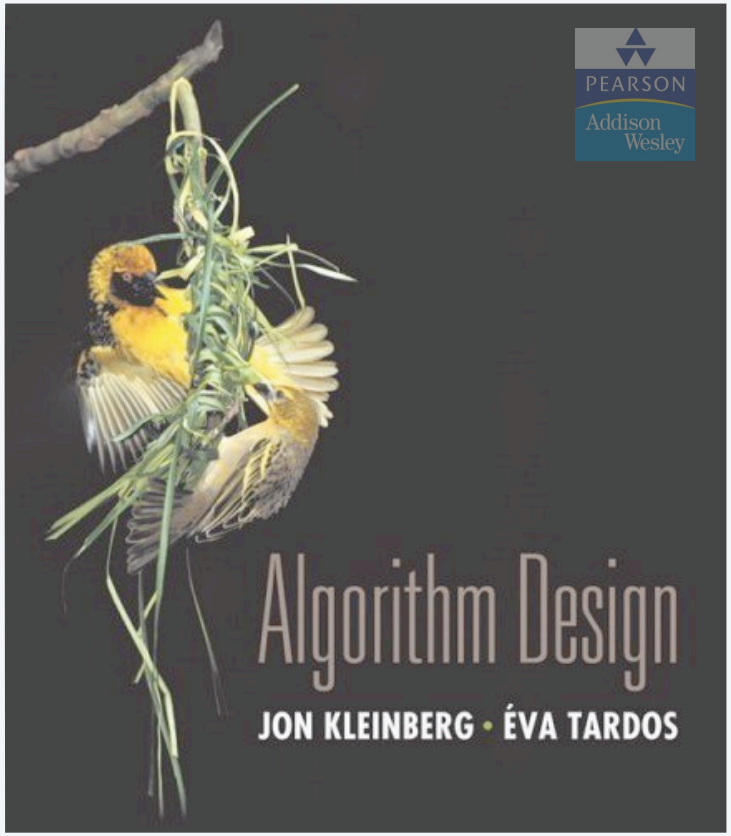
Pf.

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



SECTION 6.7

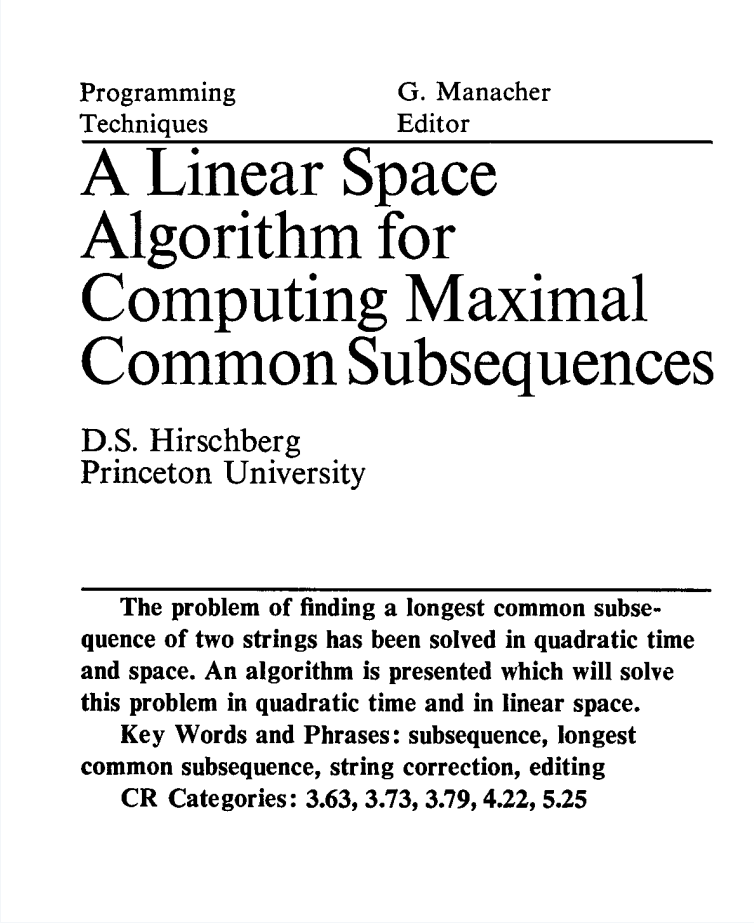
6. DYNAMIC PROGRAMMING II

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

Sequence alignment 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

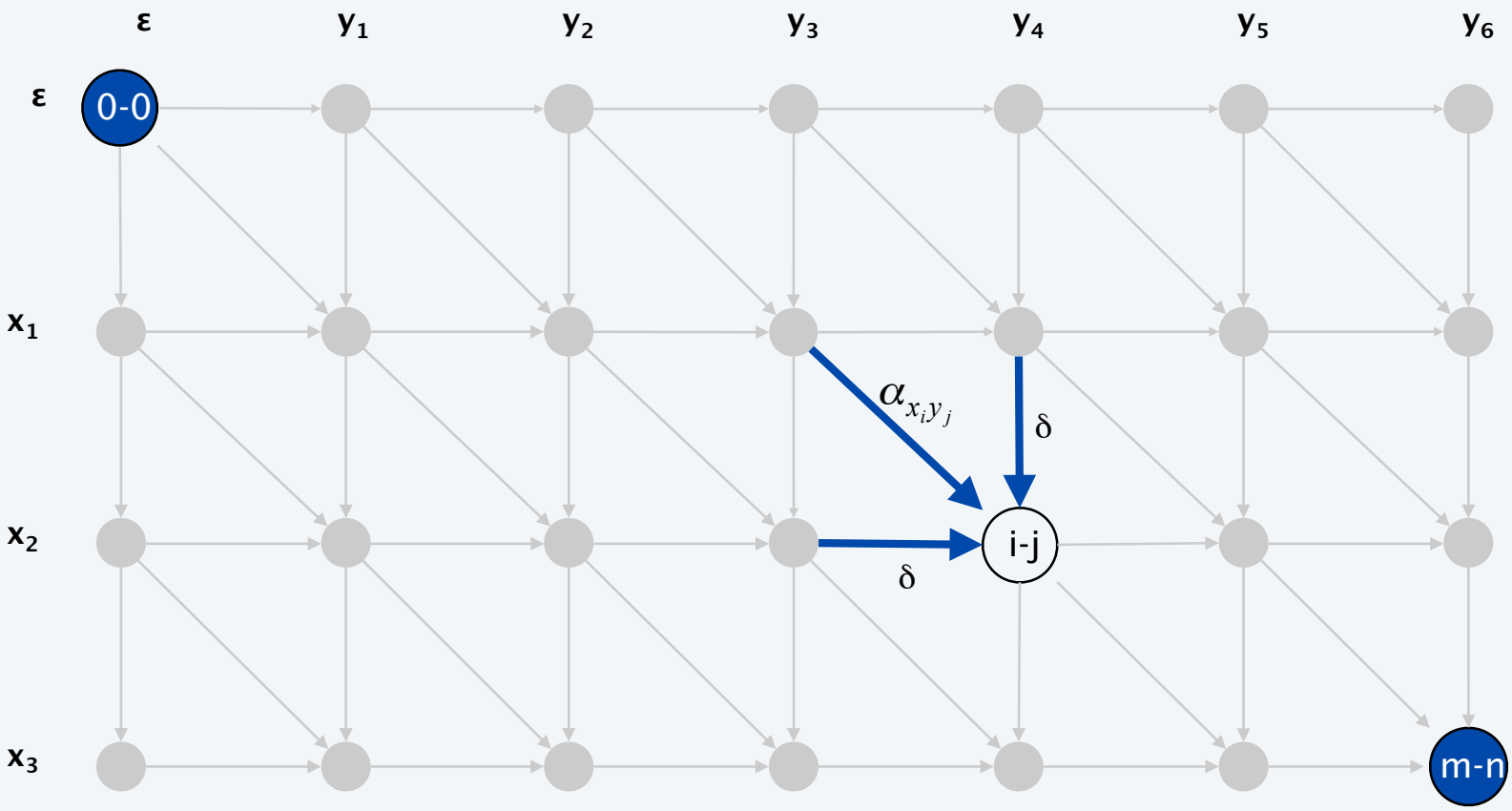
The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

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.



Hirschberg's algorithm

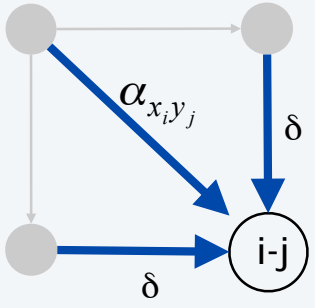
Edit distance graph.

- 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,

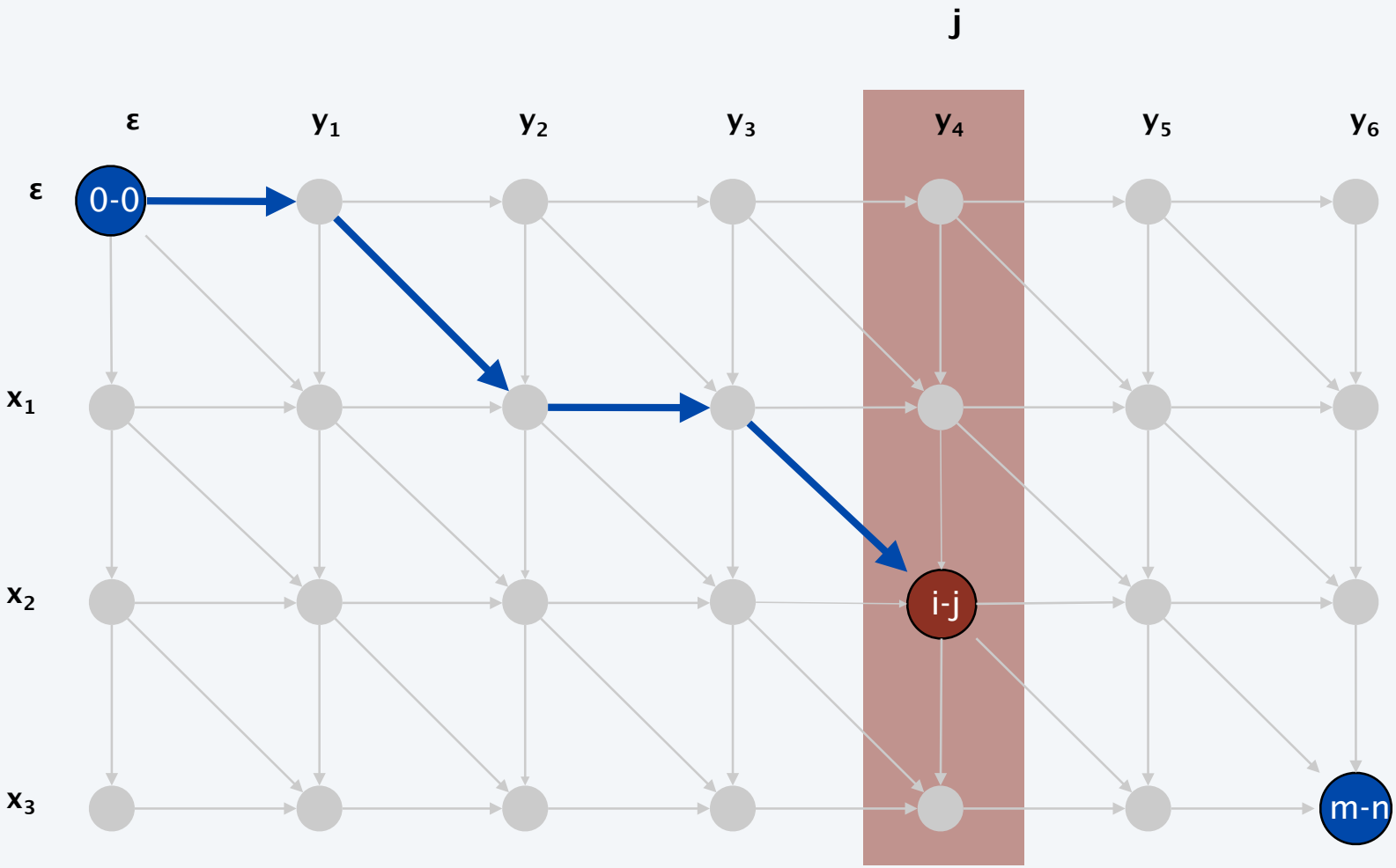
$$
\begin{aligned}
f(i, j) &= \min\{\alpha_{x_i y_j} + f(i-1, j-1),\ \delta + f(i-1, j),\ \delta + f(i, j-1)\} \\
&= \min\{\alpha_{x_i y_j} + OPT(i-1, j-1),\ \delta + OPT(i-1, j),\ \delta + OPT(i, j-1)\} \\
&= OPT(i, j) \quad \blacksquare
\end{aligned}
$$



Hirschberg's algorithm

Edit distance graph.

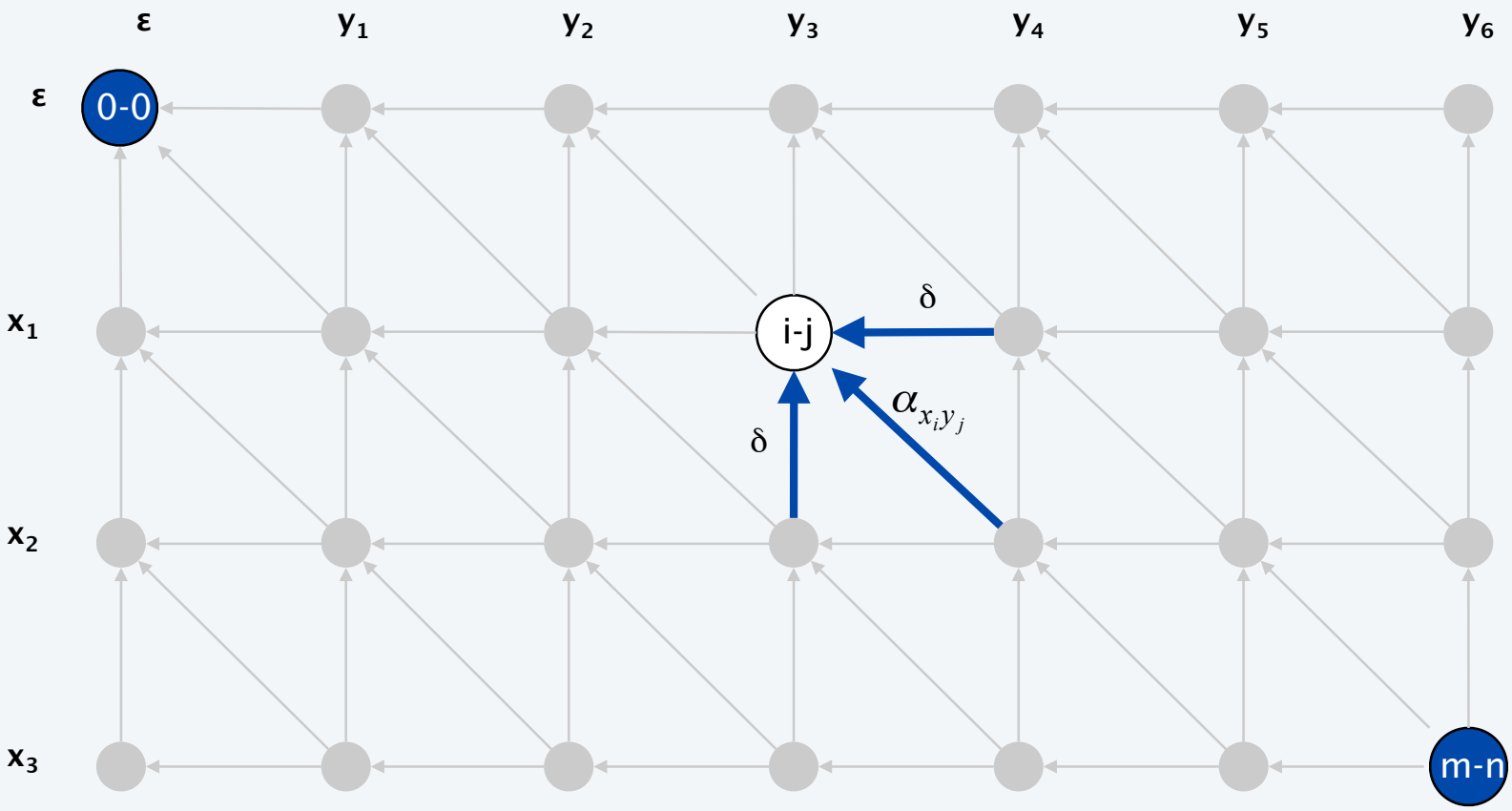
- Let $f(i, j)$ be shortest path from $(0,0)$ to (i, j).
- Lemma: $f(i, j) = OPT(i, j)$ for all i and j.
- Can compute $f(\bullet, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



Hirschberg's algorithm

Edit distance graph.

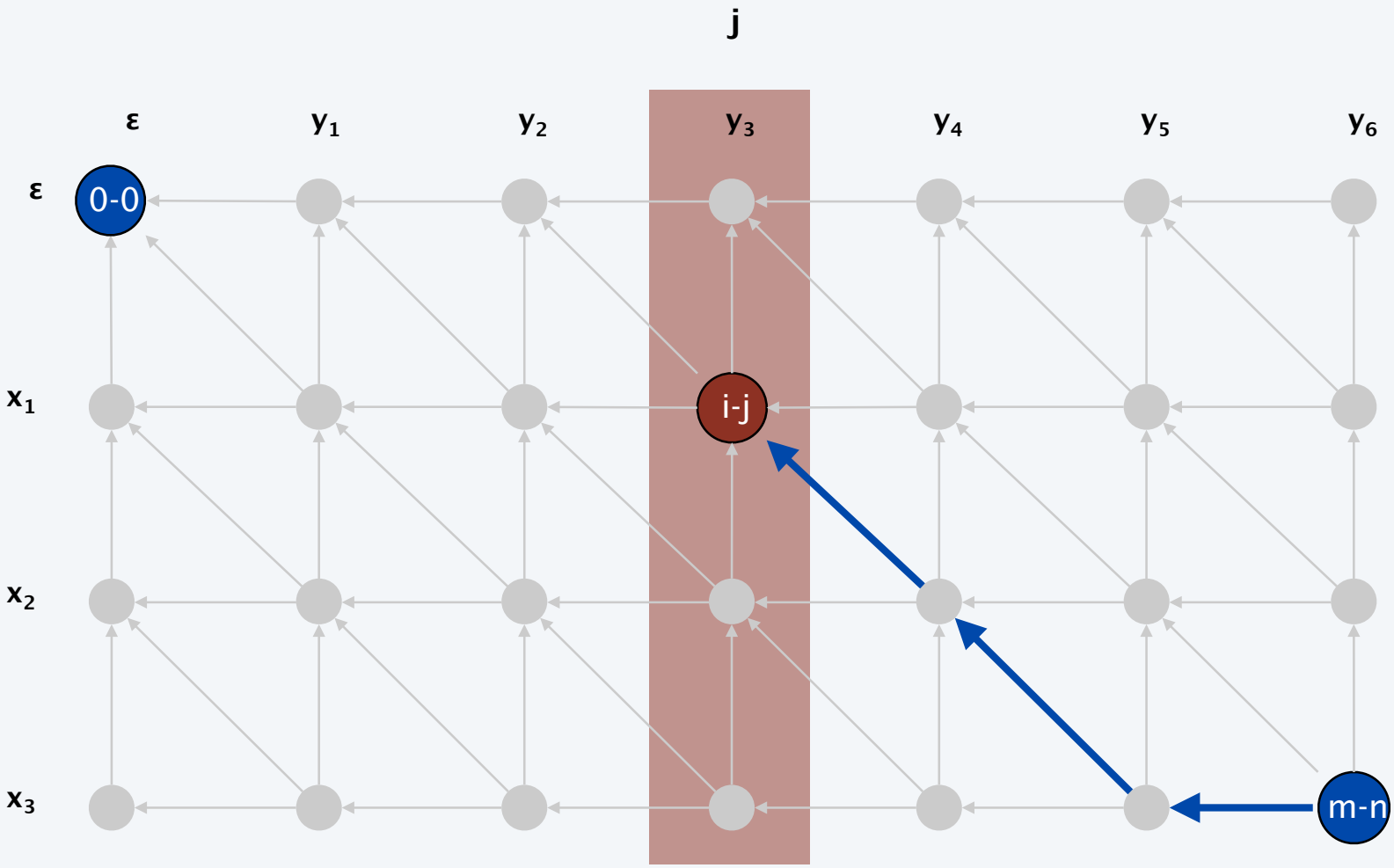
- Let $g(i, j)$ be shortest path from (i, j) to (m, n).
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and (m, n).



Hirschberg's algorithm

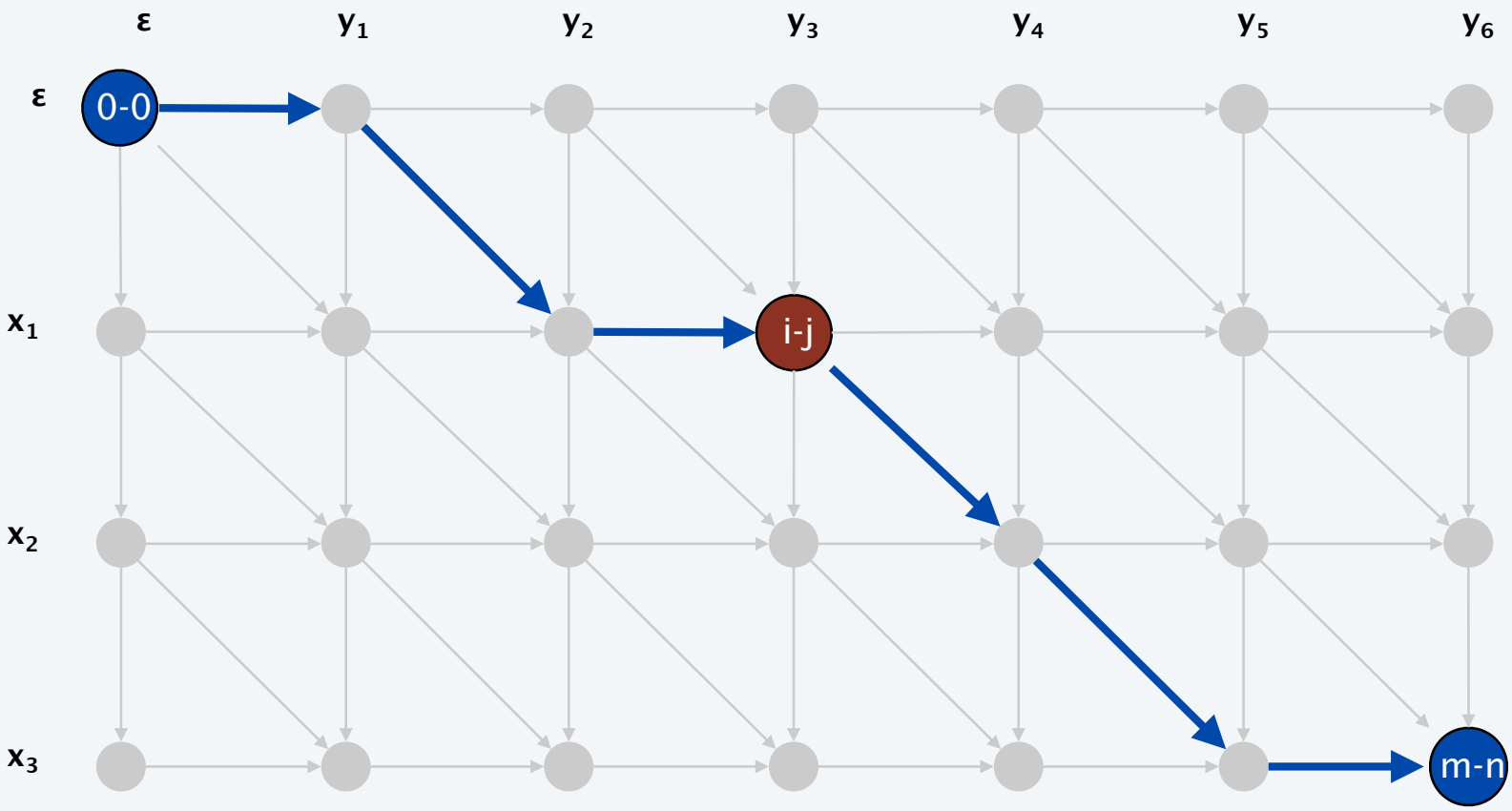
Edit distance graph.

- 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.



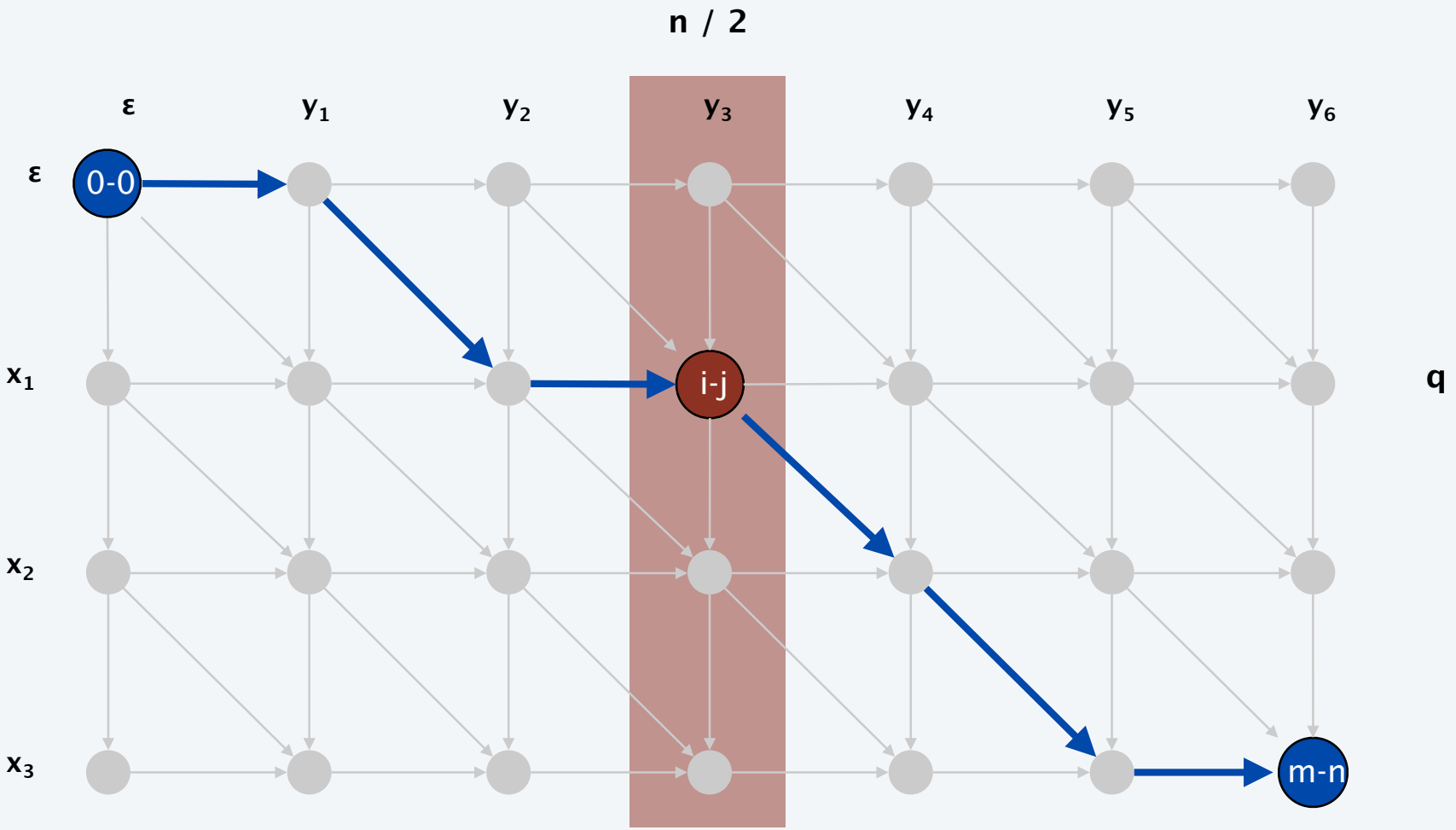
Hirschberg's algorithm

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



Hirschberg's algorithm

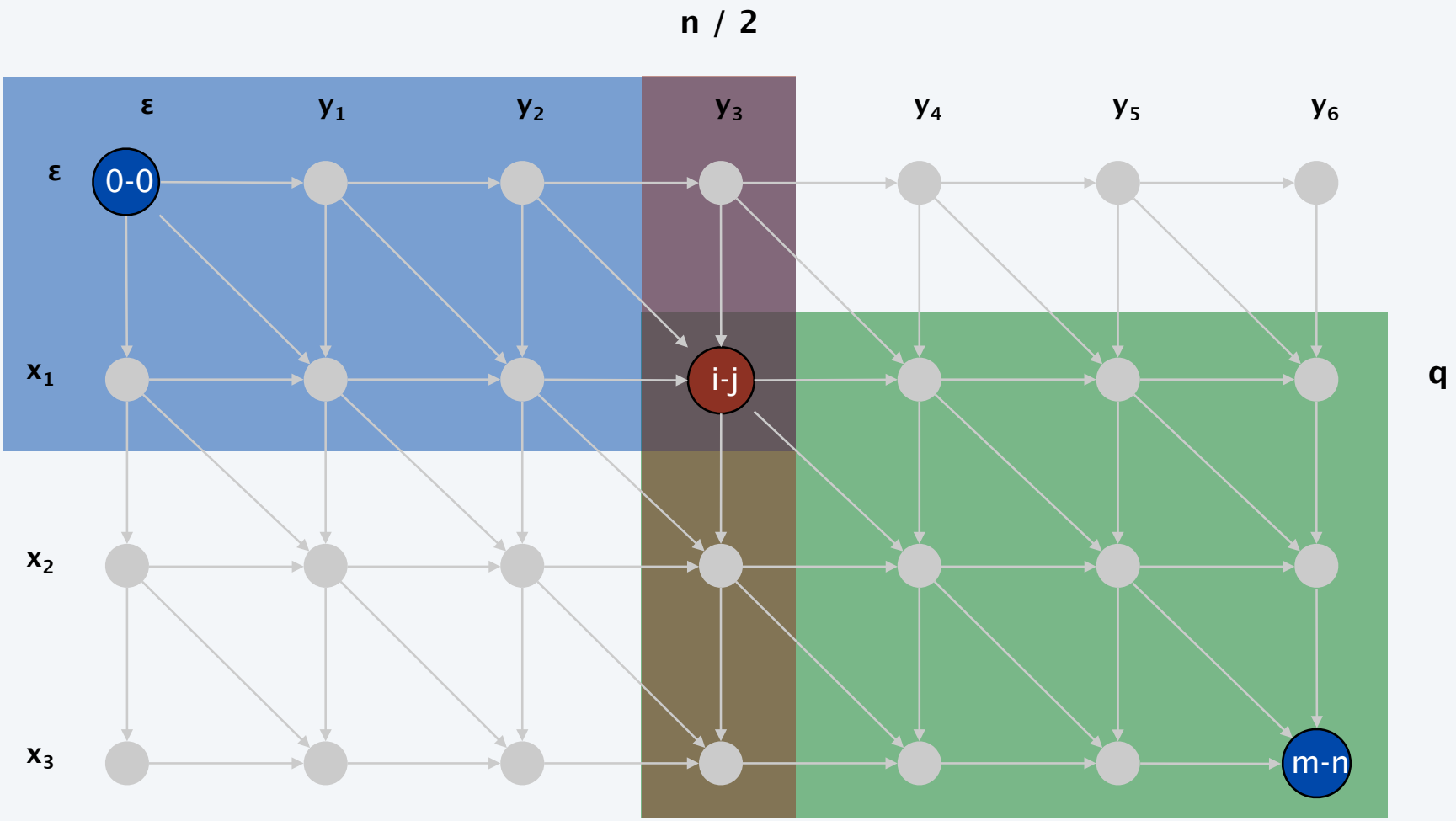
Observation 2. let q be an index that minimizes $f(q, n/2) + g(q, n/2)$.
Then, there exists a shortest path from $(0, 0)$ to (m, n) uses $(q, n/2)$.



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. $T(m, n) \leq 2\, T(m, n/2) + O(m\, n)$

$\Rightarrow T(m, n) = O(m \log n).$

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, $T(m, n) = O(mn)$.

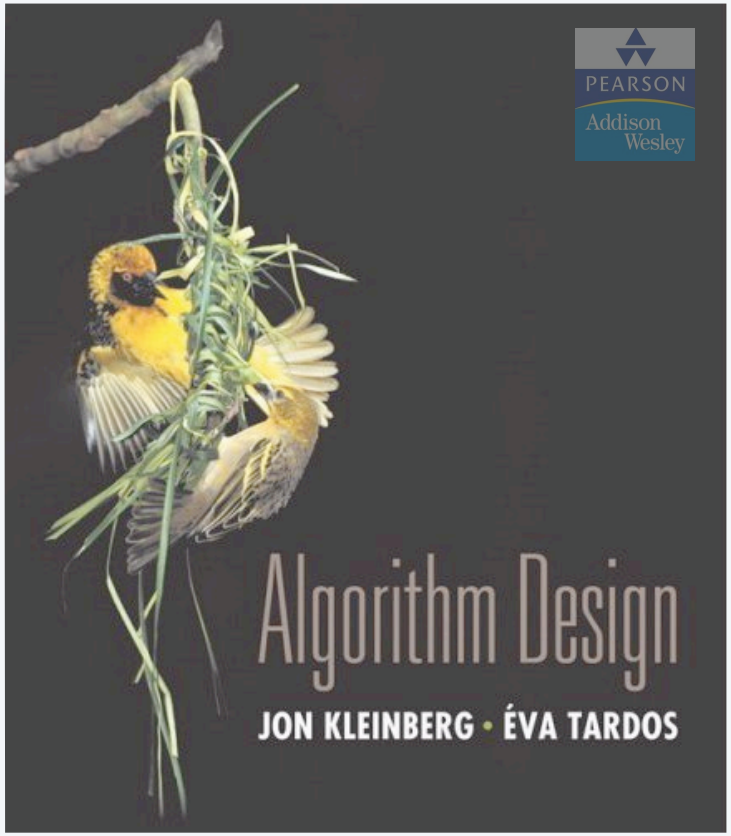
Pf. [by induction on n]

- $O(mn)$ 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:

$$\begin{aligned} T(m, 2) &\le cm \\ T(2, n) &\le cn \\ T(m, n) &\le cmn + T(q, n/2) + T(m - q, n/2) \end{aligned}$$

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

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



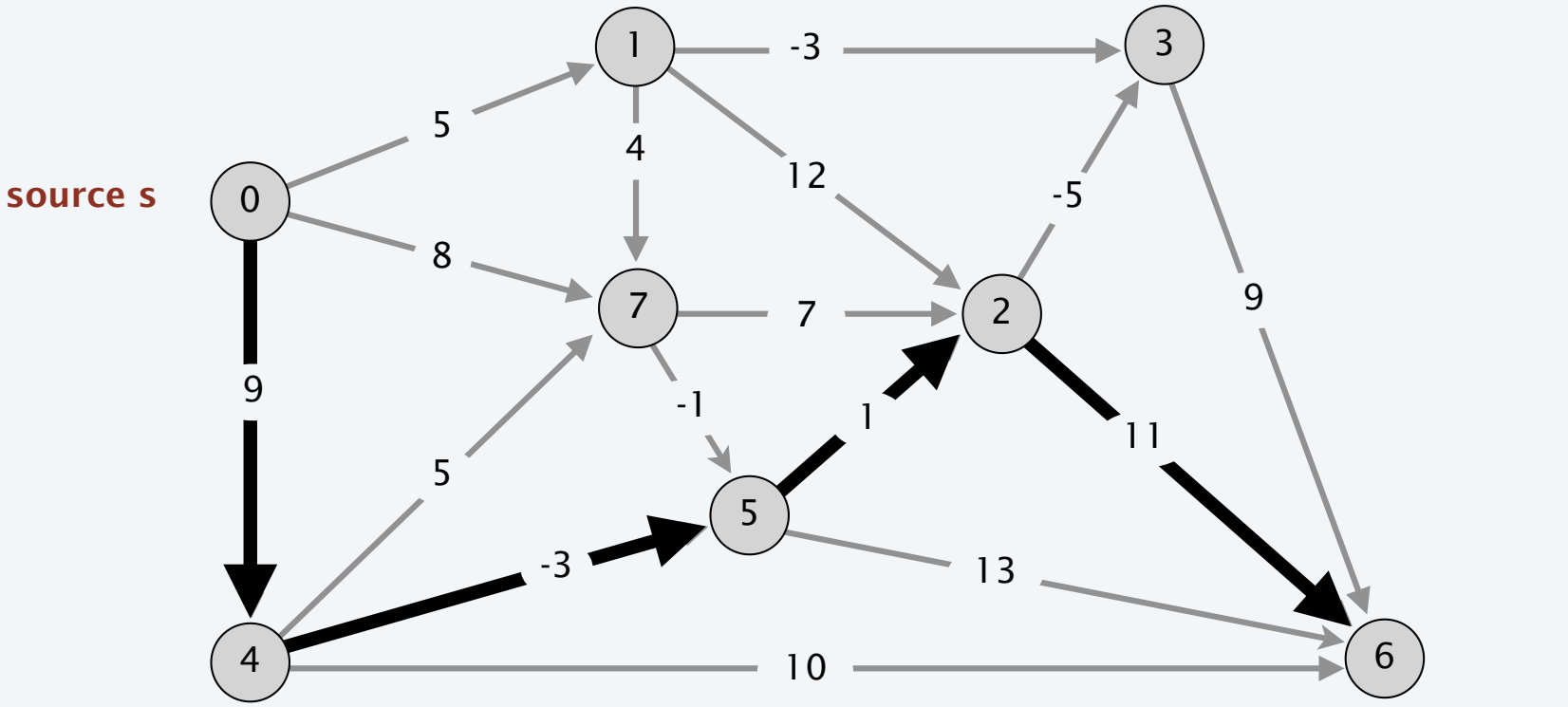
SECTION 6.8

6. DYNAMIC PROGRAMMING II

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

Shortest paths

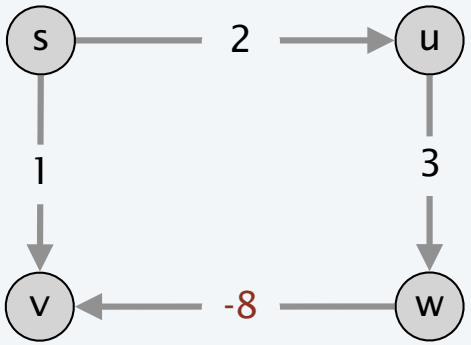
Shortest path problem. Given a digraph $G = (V, E)$, with arbitrary edge weights or costs c_{vw}, find cheapest path from node s to node t.



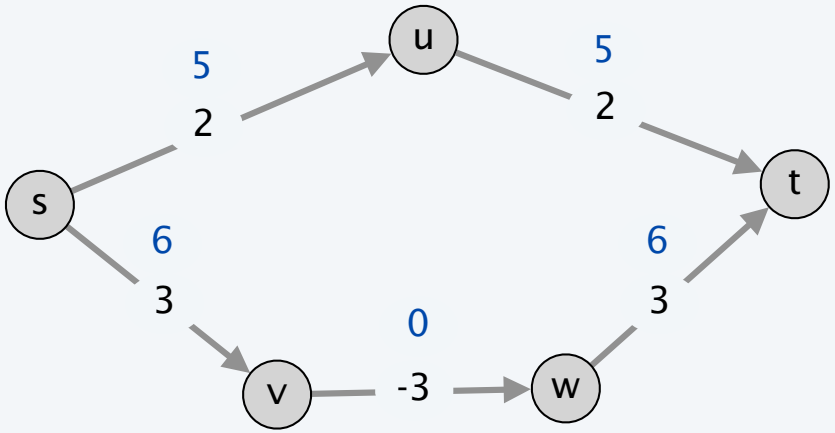
cost of path = 9 − 3 + 1 + 11 = 18

Shortest paths: failed attempts

Dijkstra. Can fail if negative edge weights.

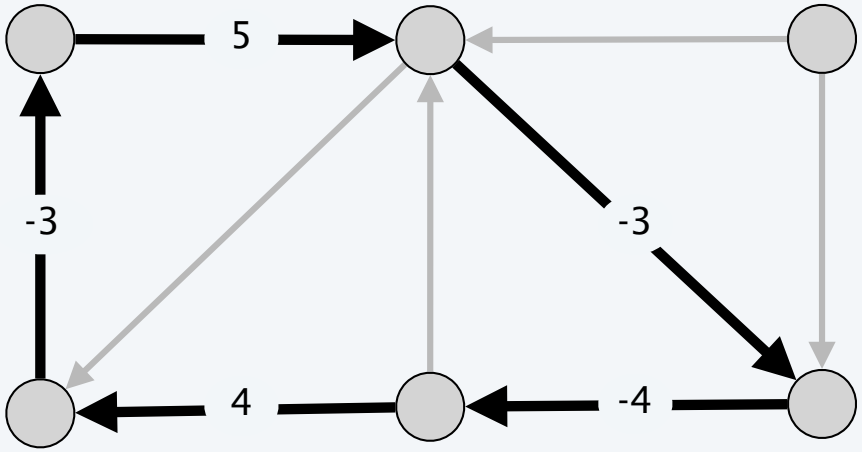


Reweighting. Adding a constant to every edge weight can fail.



Negative cycles

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

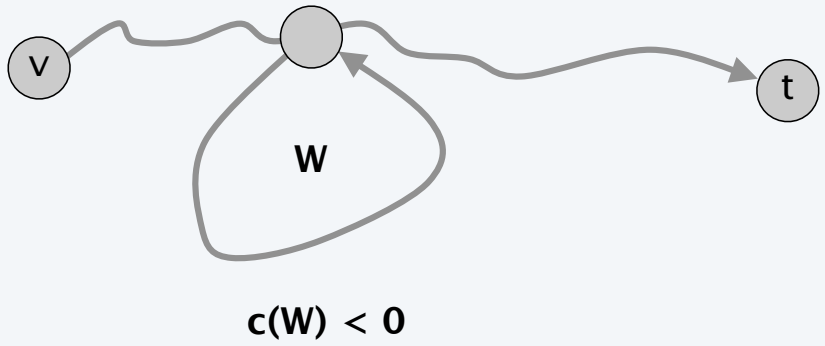


a negative cycle W : $c(W) = \sum_{e \in W} c_e < 0$

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. ▪

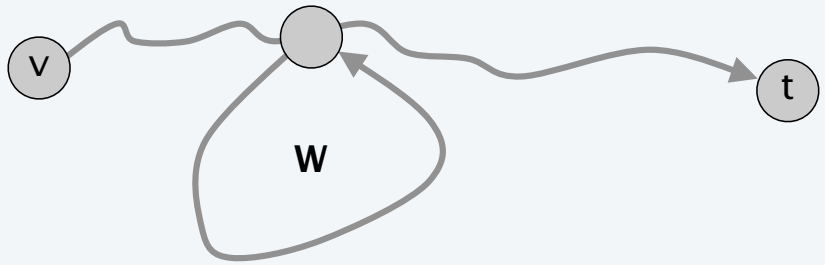


Shortest paths and negative cycles

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 $v \rightsquigarrow t$ 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. ▪

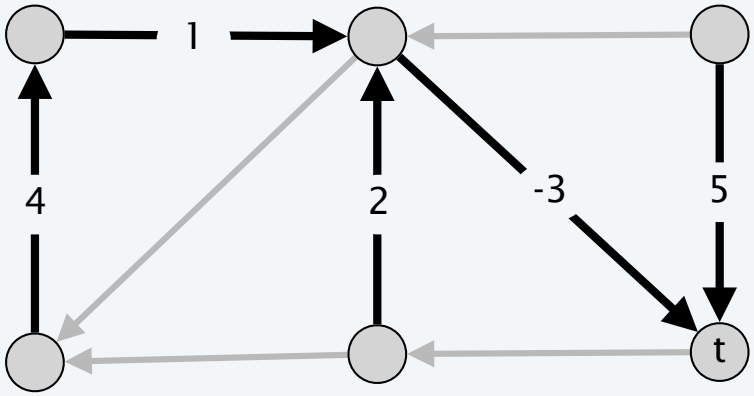


c(W) ≥ 0

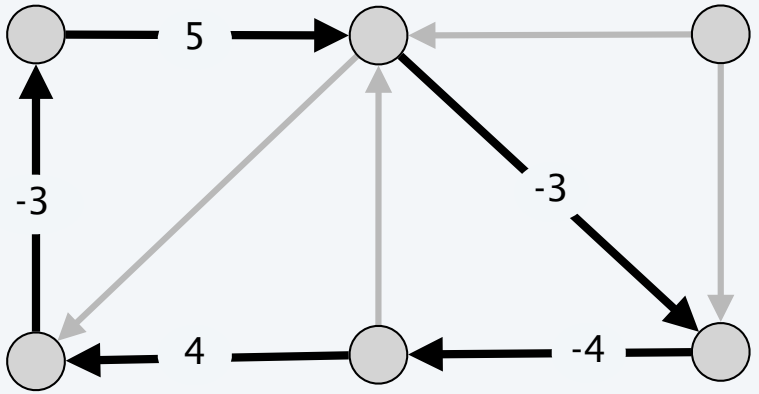
Shortest path and negative cycle problems

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 $v \to t$ path that uses $\le i$ edges.

- Case 1: Cheapest $v \to t$ path uses $\le i - 1$ edges.
  - $OPT(i, v) = \mathrm{OPT}(i - 1, v)$

optimal substructure property
(proof via exchange argument)

- Case 2: Cheapest $v \to t$ path uses exactly i edges.
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best $w \to t$ path using $\le i - 1$ edges

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

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

Pf. By Lemma 2, cheapest $v \to t$ 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 $v \rightsquigarrow t$ path that we have found so far.
- $successor(v)$ = next node on a $v \rightsquigarrow t$ 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 ($d(v) > d(w) + c_{vw}$)

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

          $successor(v) \leftarrow w$.

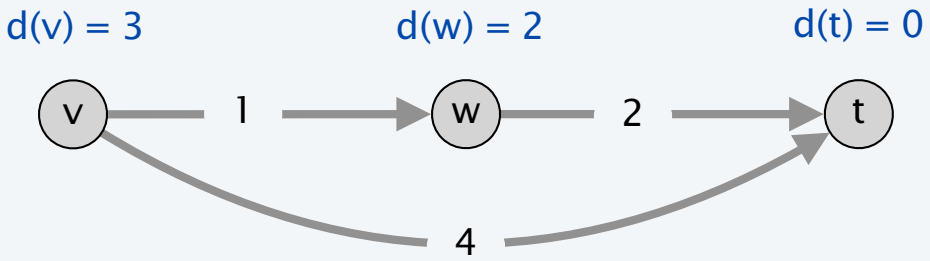
  IF no $d(w)$ value changed in iteration i, STOP.

1 pass

Bellman-Ford: analysis

Claim. After ~~the i^{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^{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^{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 $i+1$:

$$\begin{aligned} d(v) &\leq c_{vw} + d(w) \\ &\leq c_{vw} + c(P') \\ &= c(P) \quad \blacksquare \end{aligned}$$

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. ▪

can be substantially faster in practice

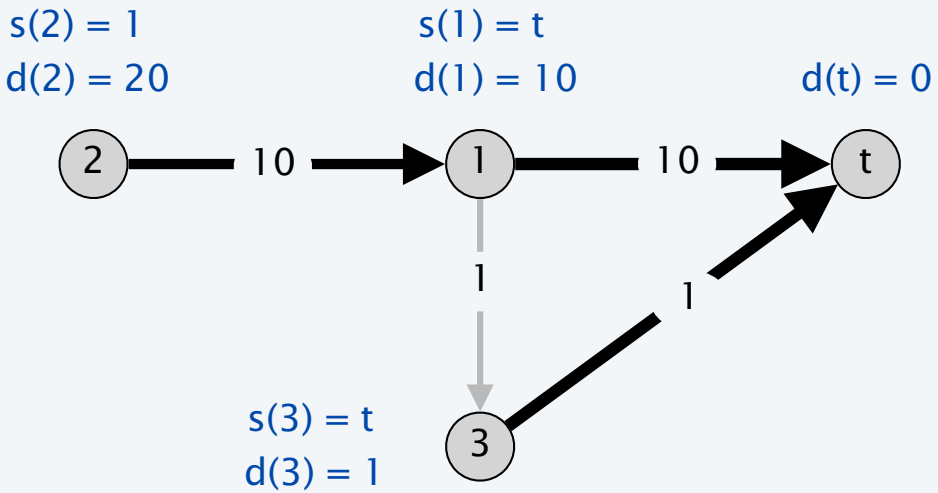
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *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



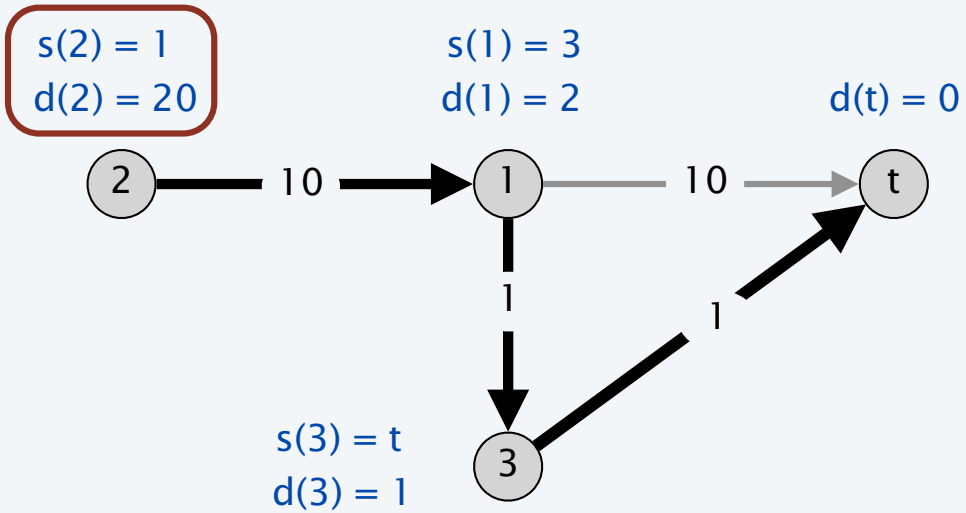
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *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



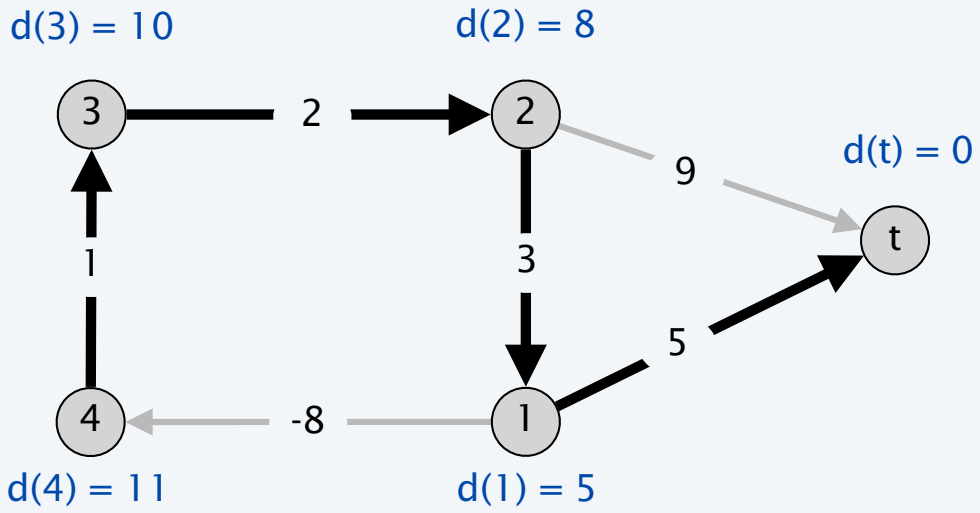
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *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



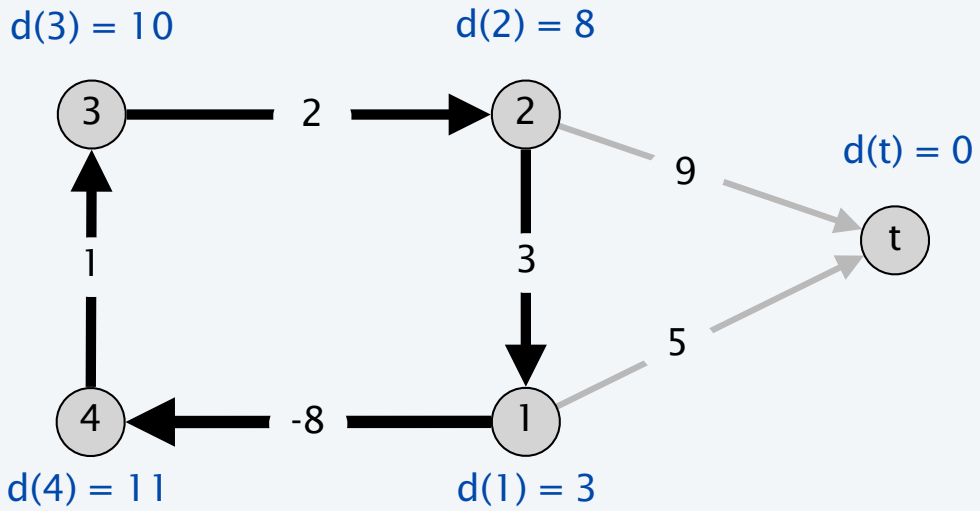
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *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 $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ 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:

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

holds with strict inequality since we are updating $d(v_k)$

- Adding inequalities yields $c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k) + c(v_k, v_1) < 0$. ▪

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 $s \rightsquigarrow t$ 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 $s = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = t$ be the nodes along this path P.
- Upon termination, if $successor(v) = w$, we must have $d(v) = d(w) + c_{vw}$.
  (LHS and RHS are equal when $successor(v)$ is set; $d(\cdot)$ did not change)
- Thus,
- $$\begin{aligned} d(v_1) &= d(v_2) + c(v_1, v_2) \\ d(v_2) &= d(v_3) + c(v_2, v_3) \\ &\vdots \quad\quad \vdots \quad\quad \vdots \\ d(v_{k-1}) &= d(v_k) + c(v_{k-1}, v_k) \end{aligned}$$

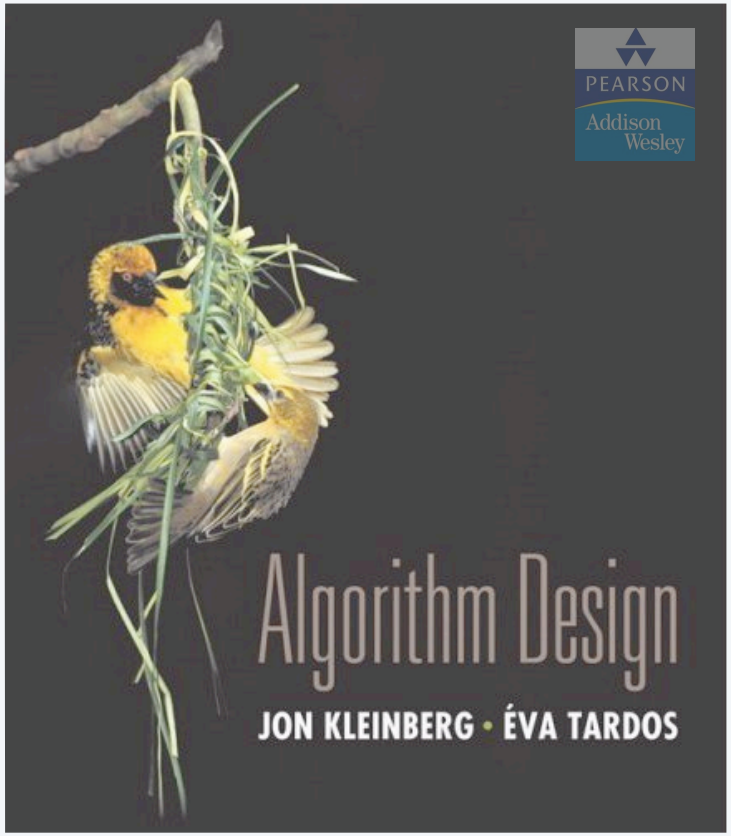
since algorithm terminated

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

min cost of any s→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 ≈ router.
- Edge ≈ direct communication link.
- Cost of edge ≈ delay on link. ⟵ 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 in not important. Moreover, algorithm still converges even if updates are asynchronous.

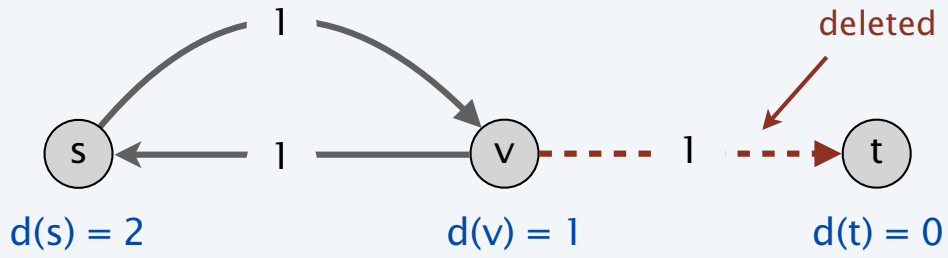
Distance vector protocols

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.

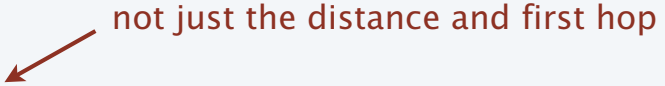
Caveat. Edge costs may change during algorithm (or fail completely).



"counting to infinity"

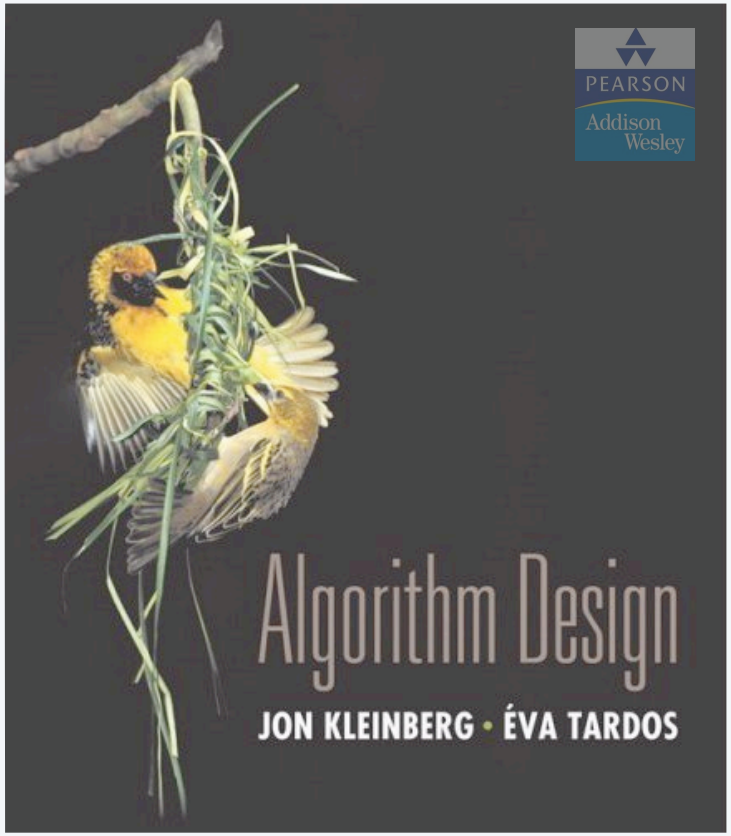
Path vector protocols

Link state routing.



- Each router also stores the entire path.
- 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).



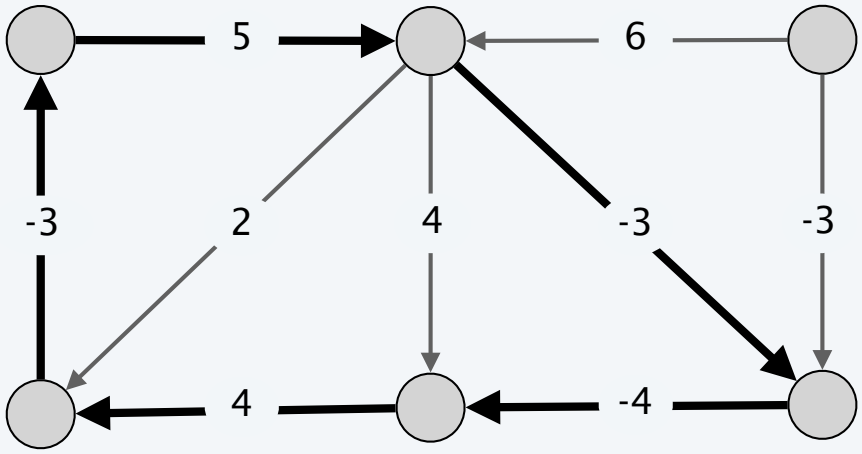
SECTION 6.10

6. DYNAMIC PROGRAMMING II

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

Detecting negative cycles

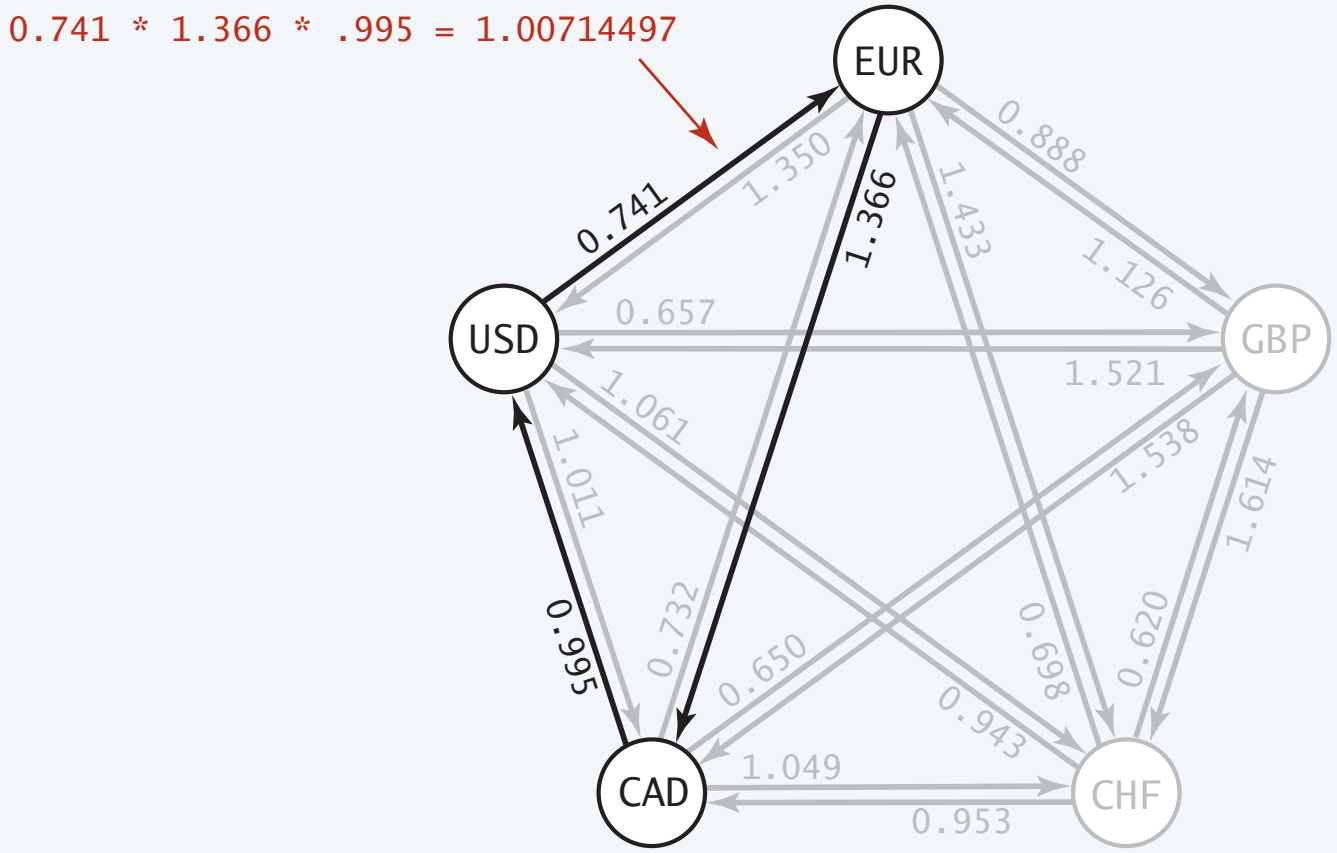
Negative cycle detection problem. Given a digraph $G = (V, E)$, with edge weights c_{vw}, find a negative cycle (if one exists).



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!



Detecting negative cycles

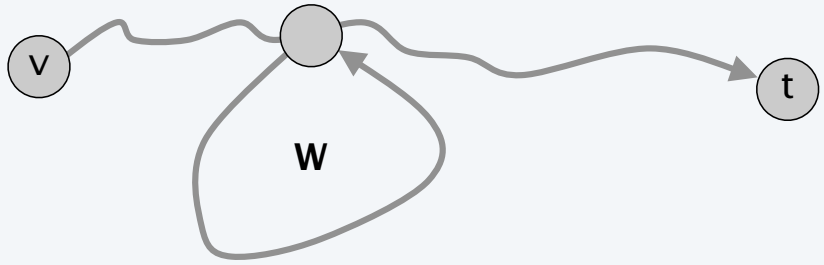
Lemma 5. If $OPT(n, v) = \text{OPT}(n-1, v)$ for all v, then no negative cycle can reach t.

Pf. Bellman-Ford algorithm. ▪

Lemma 6. If $OPT(n, v) < \text{OPT}(n-1, v)$ 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 $OPT(n, v) < \text{OPT}(n-1, v)$, 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. ▪



c(W) < 0

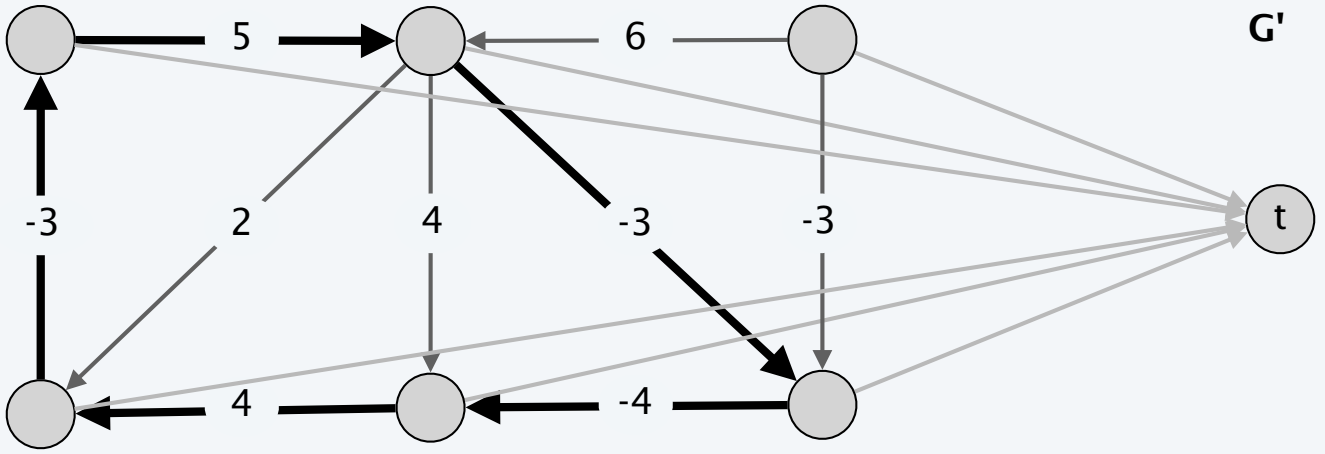
Detecting negative cycles

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)