



06 Dynamic Programming

CS216 Algorithm Design and Analysis (H)

Instructor: Shan Chen

chens3@sustech.edu.cn



Algorithmic Paradigms

- **Greedy.** Process the input in some order, and **myopically** making irrevocable decisions to optimize some underlying 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, combine solutions to smaller subproblems to form solution to large subproblem.

fancy name for caching intermediate results for later use



Dynamic Programming History

- **Richard Bellman.** Pioneered the systematic study of dynamic programming in 1950s.
- **Etymology of “dynamic programming” (DP).**
 - Dynamic programming = planning over time.
 - Secretary of Defense had pathological fear to mathematical research.
 - Bellman sought a “dynamic” adjective to avoid conflict.



“...it's impossible to use the word dynamic in a pejorative sense... Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to.”

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography*.



Dynamic Programming Applications

- **Application areas:**

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

- **Some famous DP algorithms:**

- Unix **diff** for comparing two files.
- Viterbi algorithm for hidden Markov models.
- Needleman-Wunsch/**Smith-Waterman** algorithm for sequence alignment.
- Bellman-Ford-Moore algorithm for shortest path routing in networks.



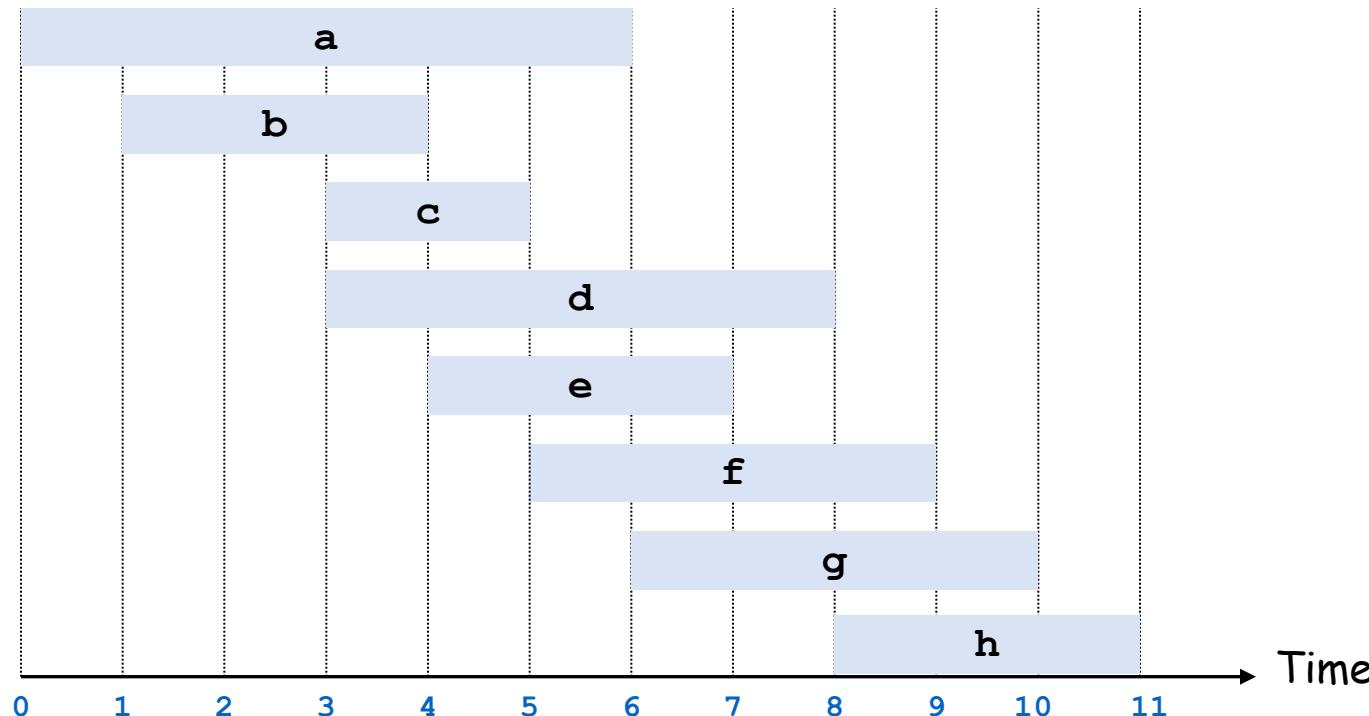
1. Weighted Interval Scheduling



Weighted Interval Scheduling

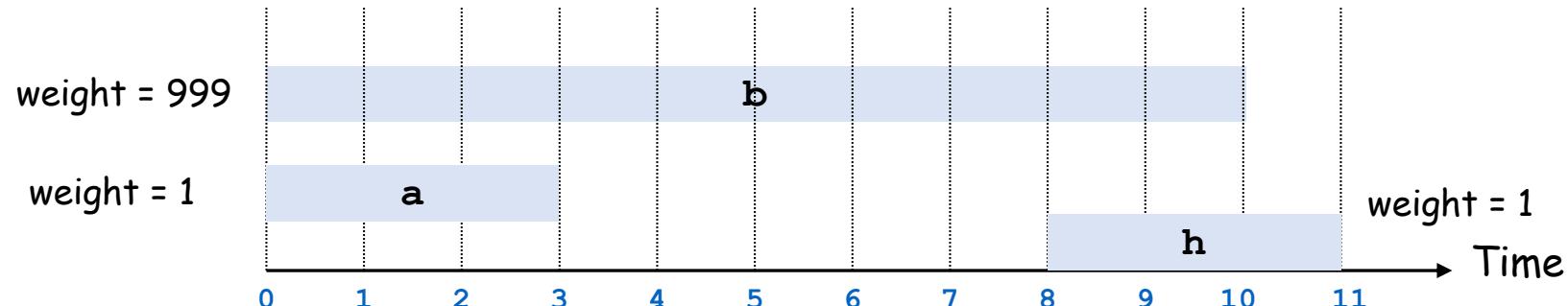
- **Weighted interval scheduling problem:**

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



Recall: Unweighted Interval Scheduling

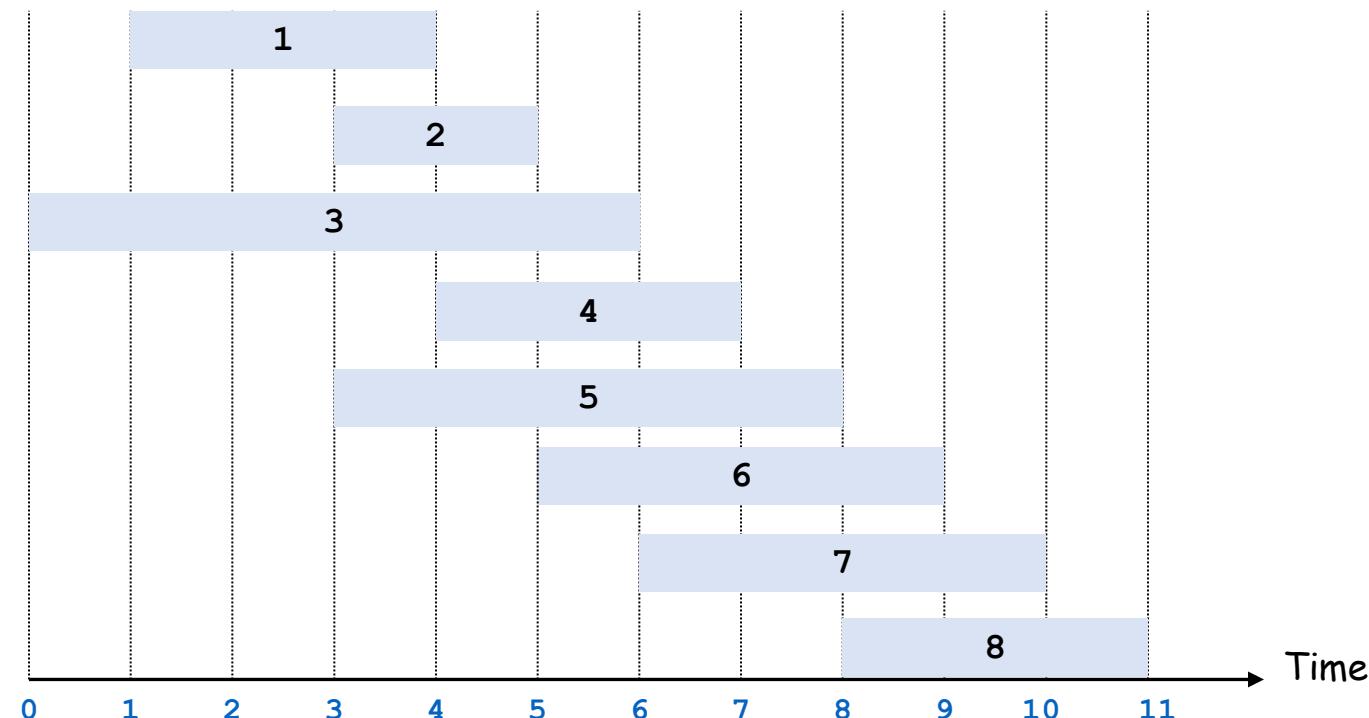
- **Recall.** Greedy algorithm works if all weights are **1**.
 - Consider jobs in **ascending order of finish time**.
 - Add job to subset if it is **compatible** with previously chosen jobs.
- **Observation.** Greedy algorithm fails spectacularly for weighted version.



Weighted Interval Scheduling

- **Convention.** Order jobs by finish time: $f_1 \leq f_2 \leq \dots \leq f_n$.
- **Def.** $p(j) =$ largest index i ($0 \leq i < j$) such that job i is compatible with job j .
- **Example.** $p(8) = 5, p(7) = 3, p(2) = 0$.

$p(j)$ is rightmost interval
that ends before j begins





Dynamic Programming: Binary Choice

- **Def.** $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \dots, j$.
- **Goal.** $OPT(n)$ = max weight of any subset of mutually compatible jobs.

- **Bellman equation:**
- **Pf. (by cases)**

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ OPT(j - 1), w_j + OPT(p(j)) \} & \text{if } j > 0 \end{cases}$$

$p(j)$: 是与作业 j 最后一个不冲突的作业的编号

- Case 1: $OPT(j)$ does not select job j .
 - ✓ Must be optimal solution to problem consisting of remaining jobs $1, 2, \dots, j - 1$.
- Case 2: $OPT(j)$ selects job j .
 - ✓ Collect weight w_j . (Then can't use incompatible jobs $p(j) + 1, p(j) + 2, \dots, j - 1$.)
 - ✓ Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, p(j)$.



Weighted Interval Scheduling: Brute Force

- Brute-force algorithm:

```
Input: n, s1,...,sn , f1,...,fn , w1,...,wn
```

```
Sort jobs by finish times and renumber so that f1 ≤ f2 ≤ ... ≤ fn  
Compute p(1), p(2), ..., p(n) via binary search
```

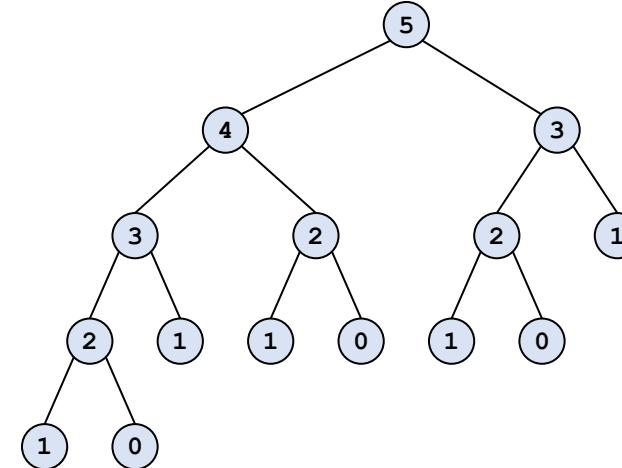
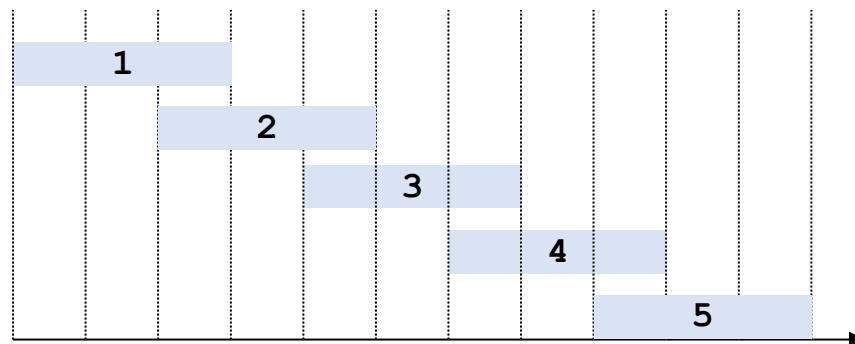
```
return Compute-Opt(n)
```

```
Compute-Opt(j) {  
    if (j = 0)  
        return 0  
    else  
        return max{ Compute-Opt(j - 1), wj + Compute-Opt(p(j)) }  
}
```

recursion could be very expensive (shown next)

Weighted Interval Scheduling: Brute Force

- **Observation.** Recursive algorithm could be spectacularly slow due to overlapping subproblems \Rightarrow exponential-time algorithm.
- **Example.** For family of “layered” instances, the number of recursive calls grows like Fibonacci sequence.



depth $\approx n$, number of recursive calls $\approx 2^n$



Weighted Interval Scheduling: Algorithm

- **Top-down DP algorithm (memorization).** Cache results of subproblem j in $M[j]$ to avoid solving the same subproblem more than once.

```
Input: n, s1, ..., sn, f1, ..., fn, w1, ..., wn
```

```
Sort jobs by finish times and renumber so that f1 ≤ f2 ≤ ... ≤ fn.
```

```
Compute p(1), p(2), ..., p(n) via binary search
```

```
M[0] = 0 ← M[] is global array
```

```
return M-Compute-Opt(n)
```

```
M-Compute-Opt(j) {  
    if (M[j] is uninitialized) → recur only when M[j] is unknown  
        M[j] = max{ M-Compute-Opt(j - 1), wj + M-Compute-Opt(p(j)) }  
    return M[j]  
}
```



Weighted Interval Scheduling: Running Time

- **Theorem.** Top-down DP algorithm takes $O(n \log n)$ time.
- **Pf. (direct proof)**
 - Sort by finish time: $O(n \log n)$.
 - Compute $p(\cdot)$ for each j : $O(n \log n)$ via binary search.
 - M-Compute-Opt(j): each invocation takes $O(1)$ time and either
 - 1) returns an existing value $M[j]$; or
 - 2) initializes $M[j]$ and makes two recursive calls
 - At most n uninitialized $M[]$ entries \Rightarrow at most $2n$ recursive calls.
 - Overall running time of M-Compute-Opt(n) is $O(n)$. ▪



Weighted Interval Scheduling: Finding a Solution

- Q. DP computes optimal value $M[n]$. How to find an optimal solution?
- A. Do backtrack post-processing by calling **Find-Solution(n)**.

```
Find-Solution(j) {
    if (j = 0)
        return empty set
    else if (M[j] > M[j - 1])
        return {j} ∪ Find-Solution(p(j)) ←  $M[j] = \max\{ M[j - 1], w_j + M[p(j)] \}$ 
    else
        return Find-Solution(j - 1)
}
```

- **Running time.** Number of recursive calls $\leq n \Rightarrow O(n)$
- **Note.** Such a **trace-back** technique also applies to other DP algorithms to find an optimal solution to other problems.



Weighted Interval Scheduling: Bottom-Up

- **Bottom-up DP algorithm.** Unwind recursion.

```
Input: n, s1, ..., sn, f1, ..., fn, v1, ..., vn
```

Sort jobs by finish times and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.
Compute $p(1), p(2), \dots, p(n)$ via binary search

```
M[0] = 0
for j = 1 to n
    M[j] = max{ M[j - 1], vj + M[p(j)] }
```

previously computed M[] values

```
return M[n]
```

- **Running time.** Bottom-up DP takes $O(n \log n)$ time.



A Quote that Suits Dynamic Programming

Those who cannot remember the past are condemned
to repeat it.

--- George Santayana



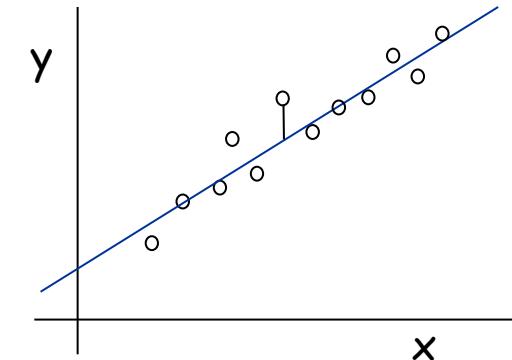
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

2. Segmented Least Squares

Least Squares

- **Least squares.** Foundational problem in statistics.
 - Given n points in the plane: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
 - Find a line $y = ax + b$ that minimizes the sum of the squared errors (SSE).

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



- **Solution.** Calculus \Rightarrow min SSE is achieved when

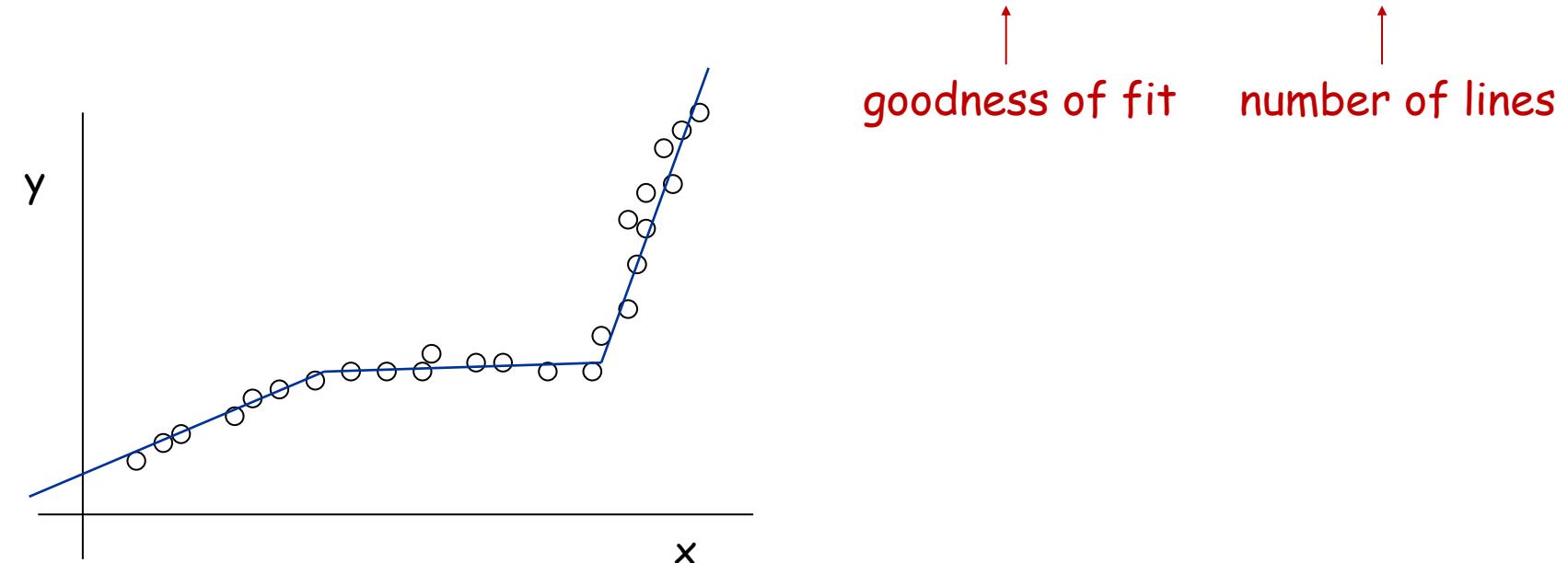
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- **Segmented least squares problem:**

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

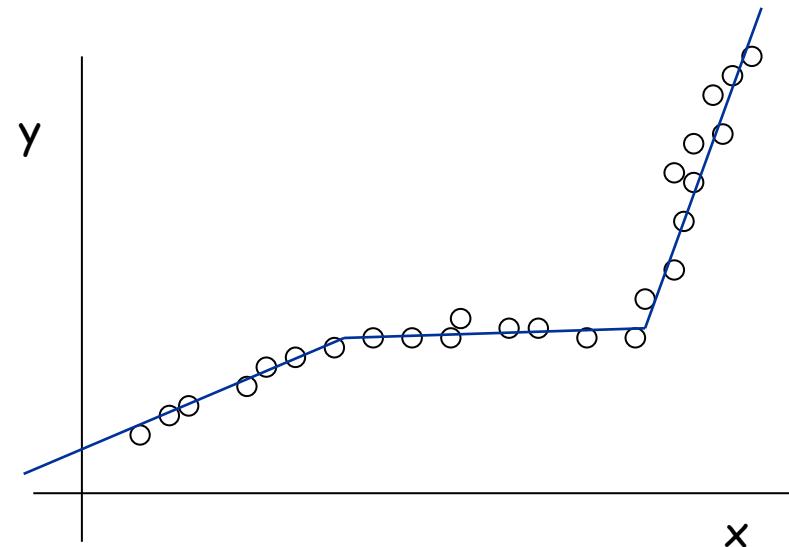
- Q. What's a reasonable choice for $f(x)$ to balance **accuracy** and **parsimony**?



Segmented Least Squares

- **Segmented least squares problem:**

- Points lie roughly on a sequence of several **line segments**.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ with $x_1 < x_2 < \dots < x_n$, find a sequence of lines that minimizes cost $f(x) = E + cL$ (for some constant $c > 0$).
 - ✓ E = sum of the sums of the squared errors (SSEs) in each segment
 - ✓ L = number of lines





Dynamic Programming: Multiway Choice

- **Notation:**

- $OPT(j)$ = minimum $f(x)$ cost for points p_1, p_2, \dots, p_j
- e_{ij} = minimum SSE for points p_i, p_{i+1}, \dots, p_j

- **Goal.** $OPT(n)$

- **To compute $OPT(j)$:**

- Last segment uses points p_i, p_{i+1}, \dots, p_j for some $i \leq j$.
- Minimum cost = $e_{ij} + c + OPT(i - 1)$.

- **Bellman equation:**

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e_{ij} + c + OPT(i - 1) \} & \text{if } j > 0 \end{cases}$$

在i的位置分出来新的一段



Segmented Least Squares: Algorithm

- Bottom-up DP algorithm:

Input: n, p_1, \dots, p_n, c

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

M[0] = 0

for $j = 1$ to n

for $i = 1$ to j

 Compute the minimum SSE e_{ij} for the segment p_i, \dots, p_j

for $j = 1$ to n

$M[j] = \min_{1 \leq i \leq j} \{ e_{ij} + c + M[i - 1] \}$

return $M[n]$

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

$n = j - i + 1$

$O(n^3)$

previously computed value

$O(n^2)$

- Running time. $O(n^3)$

Space. $O(n^2)$

- Q. Can we improve the time complexity to $O(n^2)$?



Segmented Least Squares: Algorithm

- **Bottom-up DP algorithm:**

Input: n, p_1, \dots, p_n, c

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

M[0] = 0

for $j = 1$ **to** n

for $i = 1$ **to** j

 Compute the minimum SSE e_{ij} for the segment p_i, \dots, p_j

for $j = 1$ **to** n

$M[j] = \min_{1 \leq i \leq j} \{ e_{ij} + c + M[i - 1] \}$

return $M[n]$

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

$\Theta(n^3)$ $O(n^2)$

previously computed value

$O(n^2)$

- **Remark.** Can be improved to $O(n^2)$ time.

- For each i , precompute cumulative sums:
- Compute a_{ij}, b_{ij}, e_{ij} in $O(1)$ time.

$$\sum_{k=1}^i x_k, \quad \sum_{k=1}^i y_k, \quad \sum_{k=1}^i x_k^2, \quad \sum_{k=1}^i x_k y_k$$



3. Knapsack Problem



Knapsack Problem

- **Knapsack problem:**

- Given n objects and a “knapsack”. assume all weights are integral
- Item i has value $v_i > 0$ and weighs $w_i > 0$ kilograms.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack to maximize total value of items taken.

- **Example.** What is the optimal subset for $W = 11$?

- $\{3, 4\}$ with maximum value 40

- **Greedy approaches that fail to work:**

- Repeatedly add item with maximum v_i .
- Repeatedly add item with maximum w_i .
- Repeatedly add item with maximum ratio v_i / w_i .

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7



Dynamic Programming: A First Attempt

- **Def.** $OPT(i)$ = maximum value within weight capacity W using items $1, \dots, i$.
- **Goal.** $OPT(n)$
- **To compute $OPT(i)$:**
 - Case 1: $OPT(i)$ does not select item i .
 - ✓ Select best of $\{1, 2, \dots, i - 1\}$
 - Case 2: $OPT(i)$ selects item i .
 - ✓ Which other items should be excluded?
 - ✓ How to relate to smaller subproblems $OPT(j)$ ($j < i$) ?
- **Conclusion.** Need more sub-problems!



Dynamic Programming: Adding a Variable

- **Def.** $OPT(i, w)$ = maximum value within weight limit w using items $1, \dots, i$.
- **Goal.** $OPT(n, W)$ — weights are integers from 0 to W
- **To compute $OPT(i, w)$:**
 - Case 1: $OPT(i, w)$ does not select item i .
 - ✓ Select best of $\{1, 2, \dots, i - 1\}$ within weight limit w
 - Case 2: $OPT(i)$ selects item i .
 - ✓ Collect value v_i
 - ✓ Select best of $\{1, 2, \dots, i - 1\}$ within weight limit $w - w_i$
- **Bellman equation:**

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



Knapsack Problem: Algorithm

- **Bottom-up DP algorithm:**

```
Input: n, W, w1, ..., wn, v1, ..., vn

for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wi > w)
            M[i, w] = M[i - 1, w]
        else
            M[i, w] = max { M[i - 1, w], vi + M[i - 1, w - wi] }

return M[n, W]
```

- **Running time. $O(nW)$**



Knapsack DP Algorithm: Demo

$\xleftarrow{\hspace{10em}} W + 1 \xrightarrow{\hspace{10em}}$

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

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

optimal subset: {3, 4} value = $18 + 22 = 40$

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



Knapsack DP Algorithm: Space Complexity

- Bottom-up DP algorithm:

```
Input: n, W, w1, ..., wn, v1, ..., vn

for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wi > w)
            M[i, w] = M[i - 1, w]
        else
            M[i, w] = max { M[i - 1, w], vi + M[i - 1, w - wi] }

return M[n, W]
```

only M[i - 1, .] is used!

- Space. $O(nW)$ ← can we reduce space? 10^{10} operations ($\sim 1000s$) vs 10^{10} bytes ($\sim 10G$)



Knapsack DP Algorithm: Space Complexity

- Bottom-up DP algorithm:

```
Input: n, W, w1, ..., wn, v1, ..., vn

for w = 0 to W
    M[w] = 0 ← 1-D scrolling array

for i = 1 to n
    for w = W to 1 ← updating in reverse order
        if (wi > w)
            M[w] = M[w] ← these M[.] values are from (i - 1)-th iteration!
        else
            M[w] = max { M[w], vi + M[w - wi] }

return M[W]
```

- Space. $O(W)$



Knapsack Problem: Closing Remarks

- DP algorithm depends crucially on assumption that **weights are integral**.
- Running time $O(nW)$ is **not polynomial** in input size! “**pseudo-polynomial**”
- The decision version of the knapsack problem is hard: **NP-complete**.
[Section 8 of textbook]
 - Can a value $\geq V$ be achieved under a restriction of a certain capacity W ?
- There exists a **polynomial-time** algorithm that produces an **approximate** solution to the knapsack problem that has value $\leq 0.01\%$ of optimum.
[Section 11.8 of textbook]



Exercise: Coin Changing

- **Observation.** Greedy algorithm is **not optimal** for US postal denominations: $1, 10, 21, 34, 70, 100, 350, 1225, 1500$
- **Example.** $140¢$

- Greedy: $100, 34, 1, 1, 1, 1, 1, 1$.
- Optimal: $70, 70$.



- **Q.** How to solve this problem using **DP**?
- **Problem.** Given n denominations $\{d_1, d_2, \dots, d_n\}$ and a **target value V** , find the **fewest stamps** needed to make change for V (or report impossible).



Coin Changing: Dynamic Programming

- **Def.** $OPT(v)$ = minimum number of stamps to make change for value v .
- **Goal.** $OPT(V)$
- **To compute $OPT(v)$:**
 - Case i : $OPT(v)$ selects the stamp with value d_i ,
✓ $OPT(v)$ continues to select fewest stamps to make change for $v - d_i$,
 - What if $v - d_i < 0$?
✓ set $OPT(v - d_i)$ = arbitrarily large number (i.e., the impossible case)

- **Bellman equation:**
- **Running time.** $O(nV)$

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min_{1 \leq i \leq n} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 \end{cases}$$



Announcement

- Lab 8 will be released today and the deadline is Apr 22.



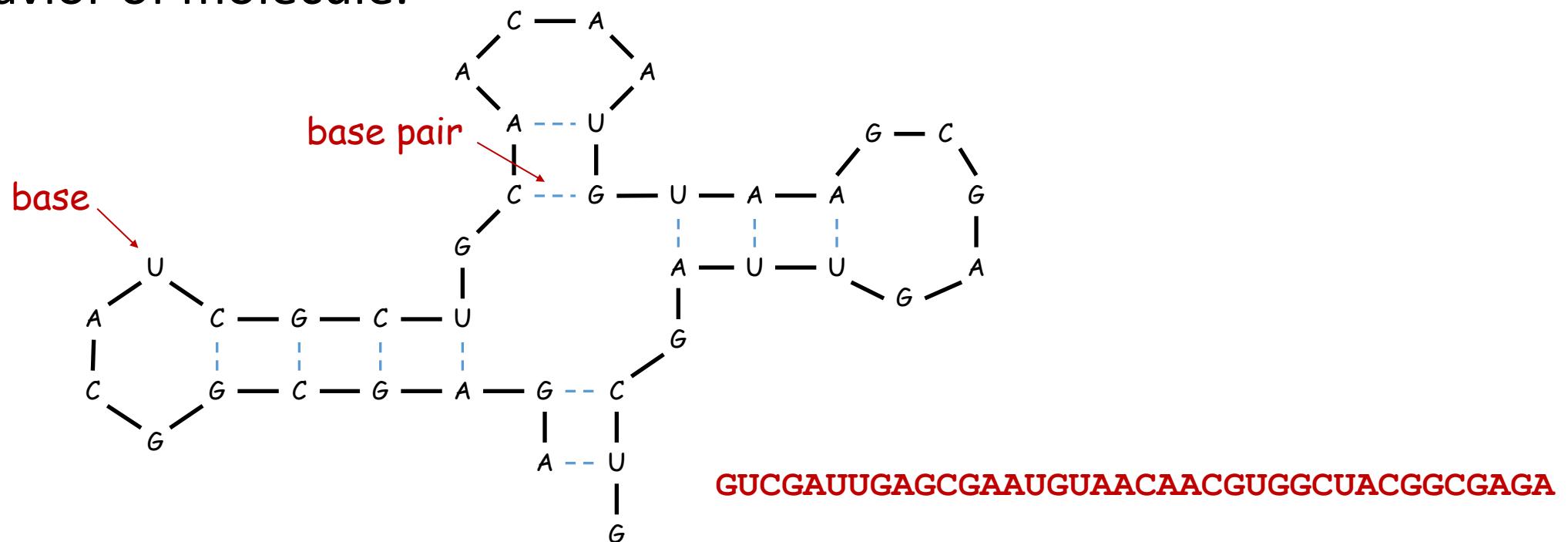
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

4. RNA Secondary Structure



RNA Secondary Structure

- **RNA.** String $B = b_1b_2\dots b_n$ over alphabet $\{A, C, G, U\}$.
- **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.

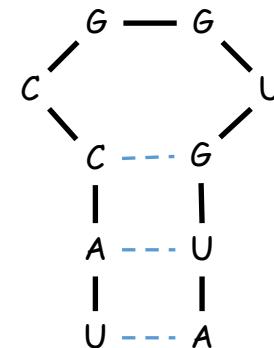




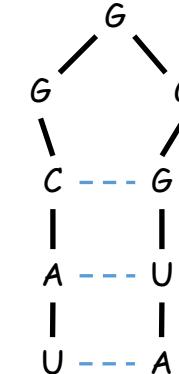
RNA Secondary Structure

- **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 > 4 intervening bases.
If $(b_i, b_j) \in S$, then $i < j - 4$.
每对配对的碱基之间要有 超过 4 个碱基的间隔
 - [Non-crossing] If (b_i, b_j) and (b_k, b_l) are two pairs in S , cannot have $i < k < j < l$.

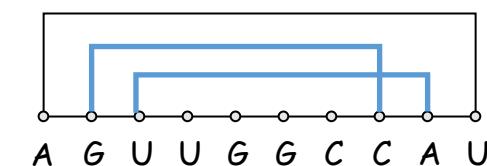
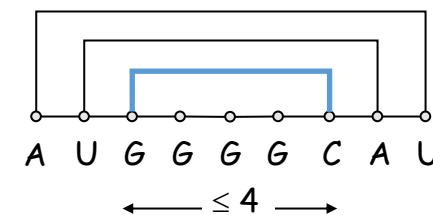
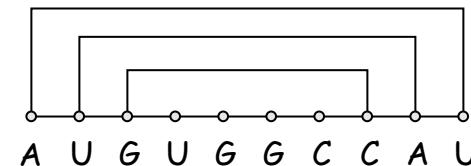
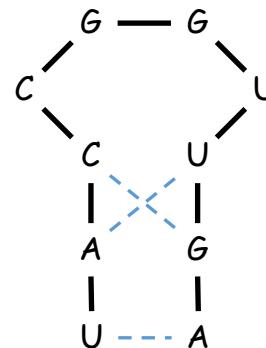
Watson-Crick
base pair



sharp turn



crossing





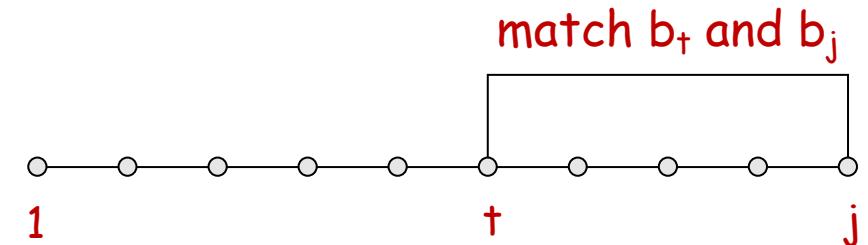
RNA Secondary Structure

- **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 > 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 , cannot have $i < k < j < l$.
- **Free-energy hypothesis.** RNA molecule will form the secondary structure with the minimum total free energy.
 - approximate by number of base pairs
more base pairs \Rightarrow lower free energy
- **Goal.** Given an RNA molecule $B = b_1 b_2 \dots b_n$, find a secondary structure S that maximizes the number of base pairs.



Dynamic Programming: A First Attempt

- **Def.** $OPT(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \dots b_j$.
- **Goal.** $OPT(n)$
- **Choices.** Match base b_t ($1 \leq t < j$) and b_j .
- **Difficulty.** Results in two sub-problems (but one of wrong form).
 - Find secondary structure in: $b_1 b_2 \dots b_{t-1}$. $\leftarrow OPT(t - 1)$
 - Find secondary structure in: $b_{t+1} b_{t+2} \dots b_{j-1}$. \leftarrow need more subproblems

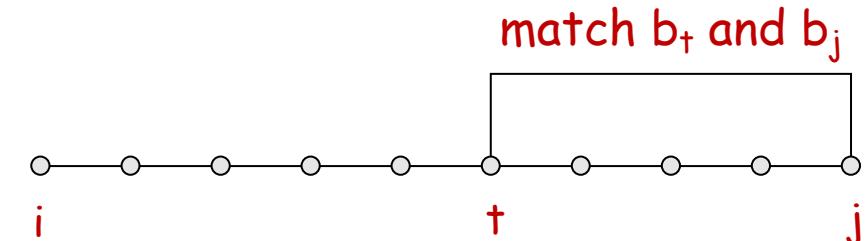


不是一个以1开头的子串，无法直接用OPT函数表



Dynamic Programming over Intervals

- **Def.** $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$.
- **Goal.** $OPT(1, n)$
- **Choices.** Match base b_t ($i \leq t < j$) and b_j .
- **To compute $OPT(i, j)$:**
 - Case 1: If $i \geq j - 4$, $OPT(i, j) = 0$ by no-sharp-turns condition.
 - Case 2: If base b_j is not involved in a pair, $OPT(i, j) = OPT(i, j - 1)$.
 - Case 3: If base b_j is paired with b_t for some $i \leq t < j - 4$, then:
 - ✓ non-crossing constraint decouples resulting sub-problems
 - ✓ $OPT(i, j) = 1 + \max_t \{OPT(i, t - 1) + OPT(t + 1, j - 1)\}$

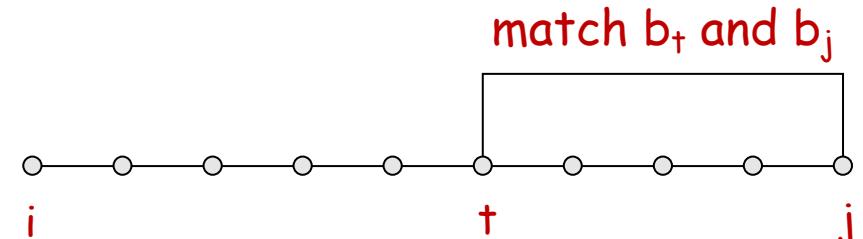


take max over t such that $i \leq t < j - 4$ and
 b_t and b_j are Watson-Crick complements



Dynamic Programming over Intervals

- **Def.** $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$.
- **Goal.** $OPT(1, n)$
- **Choices.** Match base b_t ($i \leq t < j$) and b_j .
- **Bellman equation:**



子问题的求解顺序必须保证在求 $OPT(i,j)$ 之前，所有它依赖的子问题已经被计算好。

compute in which order?

$$OPT(i, j) = \begin{cases} 0, & \text{if } i \geq j - 4 \\ OPT(i, j - 1), & b_j \text{ cannot be paired} \\ 1 + \max_t \{ OPT(i, t - 1) + OPT(t + 1, j - 1) \}, & b_j \text{ can be paired} \end{cases}$$

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



Dynamic Programming over Intervals

- Bottom-up DP algorithm:

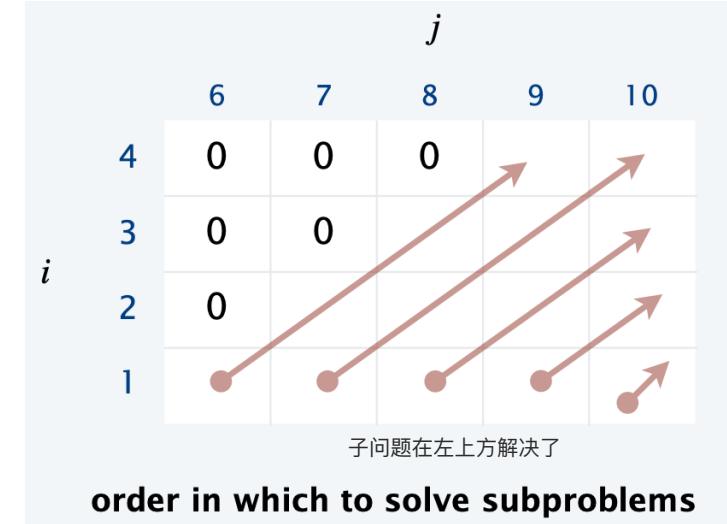
Input: n, b_1, \dots, b_n

```
for k = 5, 6, ..., n - 1      k对应子序列长度
    for i = 1, 2, ..., n - k
        j = i + k
        Compute M[i, j] using formula
            找到最优的t来裂开
```

return $M[1, n]$

all needed $M[]$ values
already computed

shortest intervals first



$$\text{OPT}(i, j) = \begin{cases} 0, & \text{if } i \geq j - 4 \\ \text{OPT}(i, j - 1), & b_j \text{ cannot be paired} \\ 1 + \max_t \{\text{OPT}(i, t - 1) + \text{OPT}(t + 1, j - 1)\}, & b_j \text{ can be paired} \end{cases}$$

指的是 b_j 和前面某个碱基 b_t 是否能配对

- Running time. $O(n^3)$ Space. $O(n^2)$

if $i \geq j - 4$

b_j cannot be paired

b_j can be paired



RNA-Secondary-Structure DP Algorithm: Demo

RNA sequence *ACCGGUAGU*

是要完成碱基配对，不是相同字母配对

4	0	0	0	
3	0	0		
2	0			
$i = 1$				
	6	7	8	9

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			
	6	7	8	9

Filling in the values
for $k = 5$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i = 1$	1	1		
	6	7	8	9

Filling in the values
for $k = 6$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	
	6	7	8	9

Filling in the values
for $k = 7$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2
	6	7	8	9

Filling in the values
for $k = 8$



Dynamic Programming: Quick Summary

- **Outline:**

- Define a collection of **subproblems**. typically only a polynomial number of subproblems
- Solution to original problem can be computed from subproblems.
- Naturally **order subproblems** from “smallest” to “largest” such that a solution to a subproblem can be **determined** from solutions to **smaller** subproblems.

- **Techniques:**

- **Binary choice:** weighted interval scheduling
- **Multiway choice:** segmented least squares
- **Adding a new variable:** knapsack problem
- **Intervals:** RNA secondary structure

- **Top-down vs. bottom-up DP.** Opinions differ.

- E.g., simpler logic/implementation vs better performance/parallelization



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

5. Sequence Alignment



Spell Correction

Hi,

This is a test for the typo/grammar checker. Can I get your respones?

陈杉

Shan Chen

Computer Science and Engineering

Southern University of Science and Technology



QuillBot

Correct the spelling error

response

Ignore



Rewrite sentence



Sciantists discovered a substance that could prove that there is life on Venus.

• SPELLING

~~Sciantists~~ → **Scientists**

The word **Sciantists** is not in our dictionary. If you're sure this spelling is correct, you can add it to your personal dictionary to prevent future alerts.

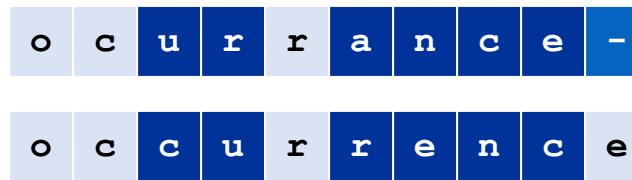
Add to dictionary



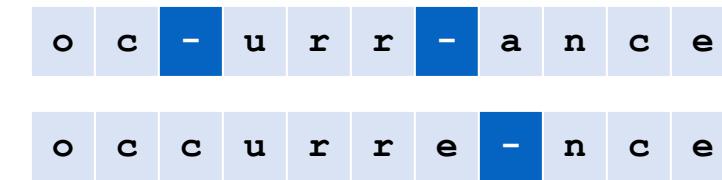


String Similarity

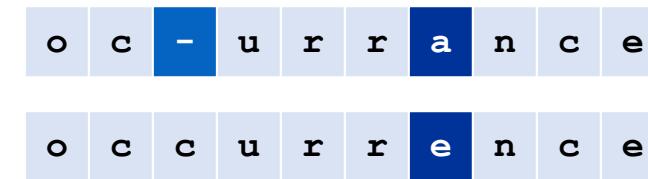
- **Q.** How similar are two strings?
- **Example.** **ocurrance** and **occurrence**.



6 mismatches, 1 gap



0 mismatches, 3 gaps



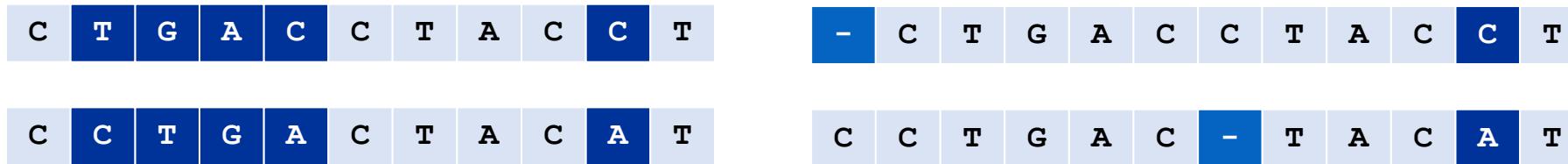
1 mismatch, 1 gap



Edit Distance

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

- Gap penalty δ ; mismatch penalty α_{pq} . (Typically, $\alpha_{pp} = 0$.)
- Cost = sum of gap and mismatch penalties.
- Edit distance is the **minimum cost**.



$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

$$2\delta + \alpha_{CA}$$

- **Applications.** Bioinformatics, spell correction, machine translation, speech recognition, information extraction, ...



BLOSUM62 Score Matrix for Aligning Proteins

	C	S	T	A	G	P	D	E	Q	N	H	R	K	M	I	L	V	W	Y	F	
C	9																			C	
S	-1	4																		S	
T	-1	1	5																	T	
A	0	1	0	4																A	
G	-3	0	-2	0	6															G	
P	-3	-1	-1	-1	-2	7														P	
D	-3	0	-1	-2	-1	-1	6													D	
E	-4	0	-1	-1	-2	-1	2	5												E	
Q	-3	0	-1	-1	-2	-1	0	2	5											Q	
N	-3	1	0	-2	0	-2	1	0	0	6										N	
H	-3	-1	-2	-2	-2	-2	-1	0	0	1	8									H	
R	-3	-1	-1	-1	-2	-2	-2	0	1	0	0	5								R	
K	-3	0	-1	-1	-2	-1	-1	1	1	0	-1	2	5							K	
M	-1	-1	-1	-1	-3	-2	-3	-2	0	-2	-2	-1	-1	5						M	
I	-1	-2	-1	-1	-4	-3	-3	-3	-3	-3	-3	-3	-3	1	4					I	
L	-1	-2	-1	-1	-4	-3	-4	-3	-2	-3	-3	-2	-2	2	2	4				L	
V	-1	-2	0	0	-3	-2	-3	-2	-2	-3	-3	-3	-2	1	3	1	4			V	
W	-2	-3	-2	-3	-2	-4	-4	-3	-2	-4	-2	-3	-3	-1	-3	-2	-3	11		W	
Y	-2	-2	-2	-2	-3	-3	-3	-2	-1	-2	2	-2	-2	-1	-1	-1	-1	2	7	Y	
F	-2	-2	-2	-2	-3	-4	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	1	3	F	
	C	S	T	A	G	P	D	E	Q	N	H	R	K	M	I	L	V	W	Y	F	

Reference: <https://en.wikipedia.org/wiki/BLOSUM>



Sequence Alignment

- **Goal.** Given two strings $x_1x_2\dots x_m$ and $y_1y_2\dots y_n$ find a **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 there are no crossings.

 x_i-y_j and $x_{i'}-y_{j'}$ cross if $i < i'$ but $j > j'$
- **Example.** CTACCG vs TACATG (gap penalty $\delta = 2$; mismatch penalty $\alpha_{pq} = 1$)
- **Solution.** $M = x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6$.

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



Sequence Alignment: Dynamic Programming

- **Def.** $OPT(i, j)$ = min cost of aligning prefix strings $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$
- **Goal.** $OPT(m, n)$
- **To compute $OPT(i, j)$:**
 - Case 1: $OPT(i, j)$ matches x_i-y_j .
 - ✓ Pay mismatch for x_i-y_j + min cost of aligning $x_1x_2 \dots x_{i-1}$ and $y_1y_2 \dots y_{j-1}$.
 - Case 2a: $OPT(i, j)$ leaves x_i unmatched.
 - ✓ Pay gap for x_i + min cost of aligning $x_1x_2 \dots x_{i-1}$ and $y_1y_2 \dots y_j$.
 - Case 2b: $OPT(i, j)$ leaves y_j unmatched.
 - ✓ Pay gap for y_j + min cost of aligning $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_{j-1}$.



Sequence Alignment: Dynamic Programming

- **Def.** $OPT(i, j) = \min \text{ cost}$ of aligning prefix strings $\underline{x_1 x_2 \dots x_i}$ and $y_1 y_2 \dots y_j$
- **Goal.** $OPT(m, n)$
- **Bellman equation:** [Needleman-Wunsch]

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

动态规划的核心是：我们按比对到当前位置 (i, j) 的所有可能路径递推，每一步都考虑是否引入 gap



Sequence Alignment: Algorithm

- Bottom-up DP algorithm:

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {  
    for i = 0 to m  
        M[i, 0] = iδ  
    for j = 0 to n  
        M[0, j] = jδ  
  
    for i = 1 to m  
        for j = 1 to n  
            M[i, j] = min{ α[xi, yj] + M[i - 1, j - 1],  
                            δ + M[i - 1, j],  
                            δ + M[i, j - 1] }  
    return M[m, n]  
}
```

- Running time. $O(mn)$



Sequence Alignment: Impossibility Result

编辑距离 (Edit Distance) 基本不可能有真正比 $O(n^2)$ 更快的算法, 除非 P=NP 级别的奇迹发生。

- **Theorem.** [Backurs-Indyk 2015] If can compute **edit distance** of two strings of length n in $O(n^{2-\varepsilon})$ time for some constant $\varepsilon > 0$, then can solve **SAT** with n variables and m clauses in $\text{poly}(m)2^{(1-\delta)n}$ time for some constant $\delta > 0$.

Edit Distance Cannot Be Computed
in Strongly Subquadratic Time
(unless SETH is false)*

SETH: strong exponential time hypothesis

Arturs Backurs[†]
MIT

Piotr Indyk[‡]
MIT

- **Takeaway.** It is very difficult (if not impossible) to improve the $O(mn)$ running time of computing the edit distance of two strings.



Sequence Alignment: Algorithm

- Bottom-up DP algorithm:

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {  
    for i = 0 to m  
        M[i, 0] = iδ  
    for j = 0 to n  
        M[0, j] = jδ  
  
    for i = 1 to m          M[i, j] depends only on M[·, j - 1] and M[i - 1, ·]  
        for j = 1 to n  
            M[i, j] = min{ α[xi, yj] + M[i - 1, j - 1],  
                            δ + M[i - 1, j],  
                            δ + M[i, j - 1] }  
    return M[m, n]  
}
```

- Running time. $O(mn)$ Space. $O(\min\{m, n\})$ ← only $\text{OPT}(m, n)$ is computed
cannot find an optimal alignment



Sequence Alignment: Finding a Solution

- Q. How to find an **optimal alignment**?
- A. Apply a **trace-back** algorithm.
- Example trace-back:
 - PALETTE
 - PAL-ATE
- Q. How much **space** is required?
- A. $O(mn)$: store the $m \times n M[\cdot, \cdot]$ matrix

\uparrow
 $m, n \leq 10$ for English words
 $m, n \approx 10^5$ for biology applications

	P	A	L	A	T	E	
P	0	2	4	6	8	10	12
A	2	0	2	4	6	8	10
L	4	2	0	2	4	6	8
E	6	4	2	0	2	4	6
T	8	6	4	2	1	3	4
T	10	8	6	4	3	1	3
E	12	10	8	6	5	3	2
	14	12	10	8	7	5	3

- Q. Can we **reduce** the required **space** for sequence alignment?



Sequence Alignment in Linear Space

- **Theorem.** [Hirschberg] There exists 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





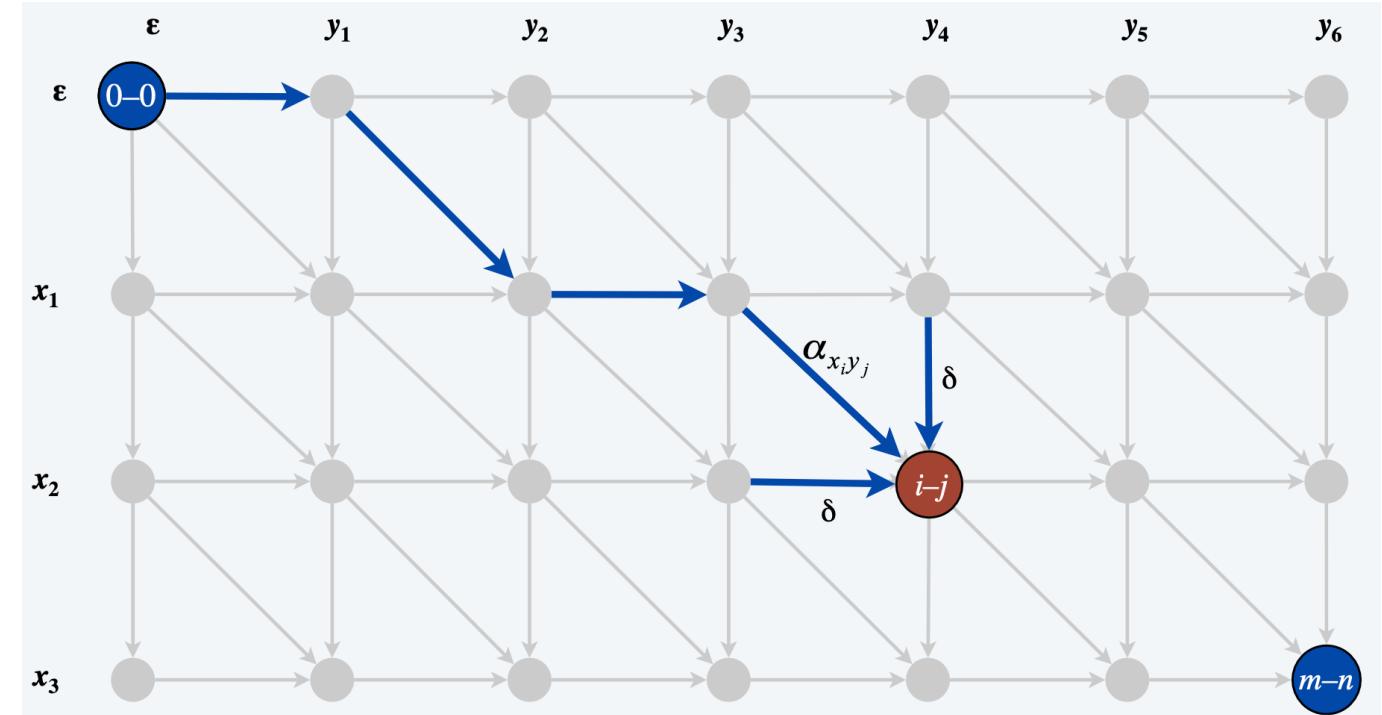
Edit Distance Graph

- **Edit distance graph:**

- rows: x
- columns: y
- path: alignment

- **Example path:**

- $-x_1-x_2 \dots x_m$
- $y_1 y_2 y_3 y_4 \dots y_n$

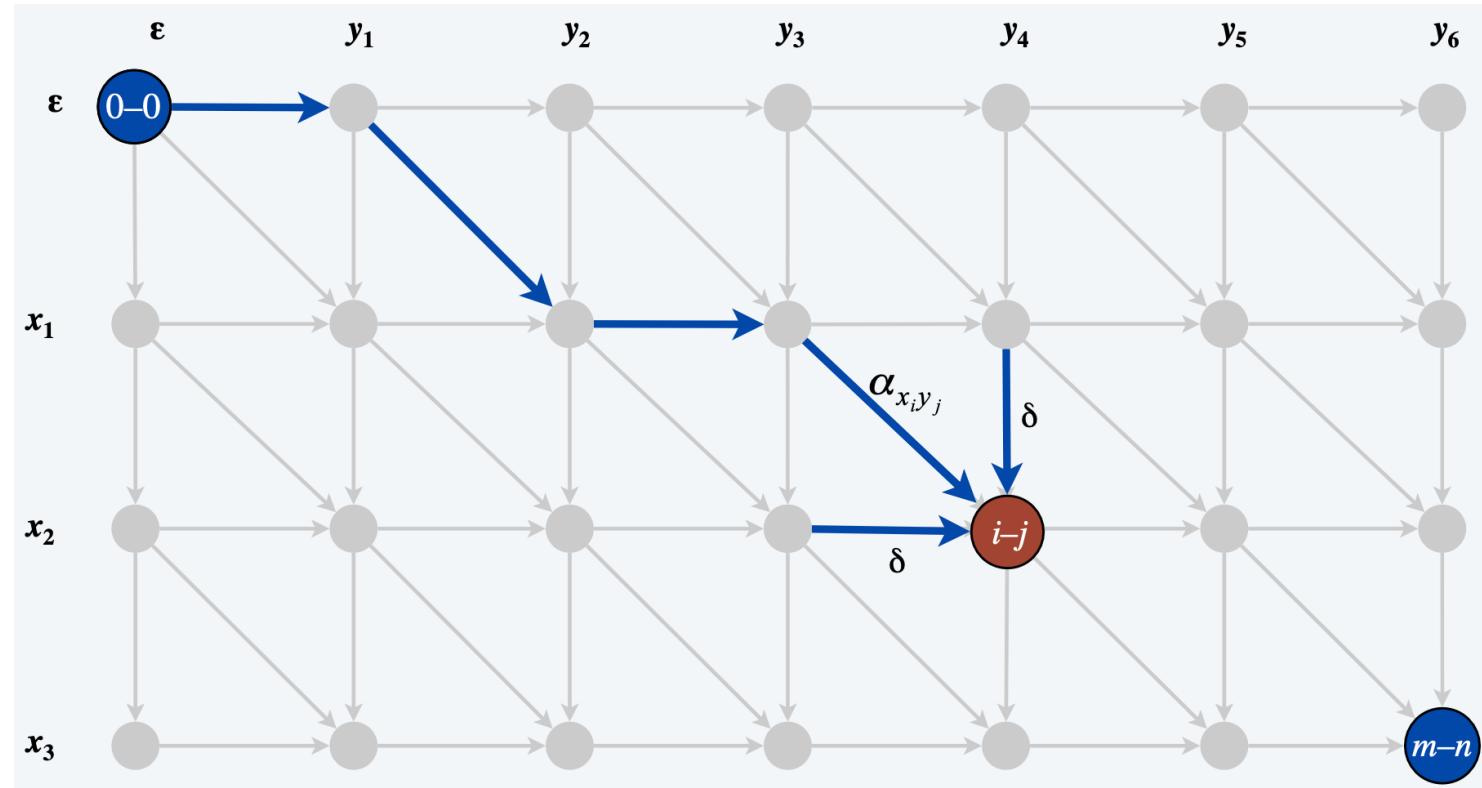




Shortest Path = Optimal Alignment

- **Lemma.** $f(i, j) = OPT(i, j)$ for all i and j .

$f(i, j) = \text{length of shortest path from } (0, 0) \text{ to } (i, j)$



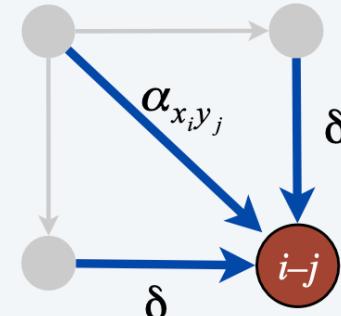
Shortest Path = Optimal Alignment

- **Lemma.** $f(i, j) = OPT(i, j)$ for all i and j .
- **Pf.** (by strong induction on $i + j$)
 - Basis 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 $(i - 1, j - 1)$ or $(i - 1, j)$ or $(i, j - 1)$.

$$f(i, j) = \min\{\alpha_{x_i y_j} + f(i - 1, j - 1), \delta + f(i - 1, j), \delta + f(i, j - 1)\}$$

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

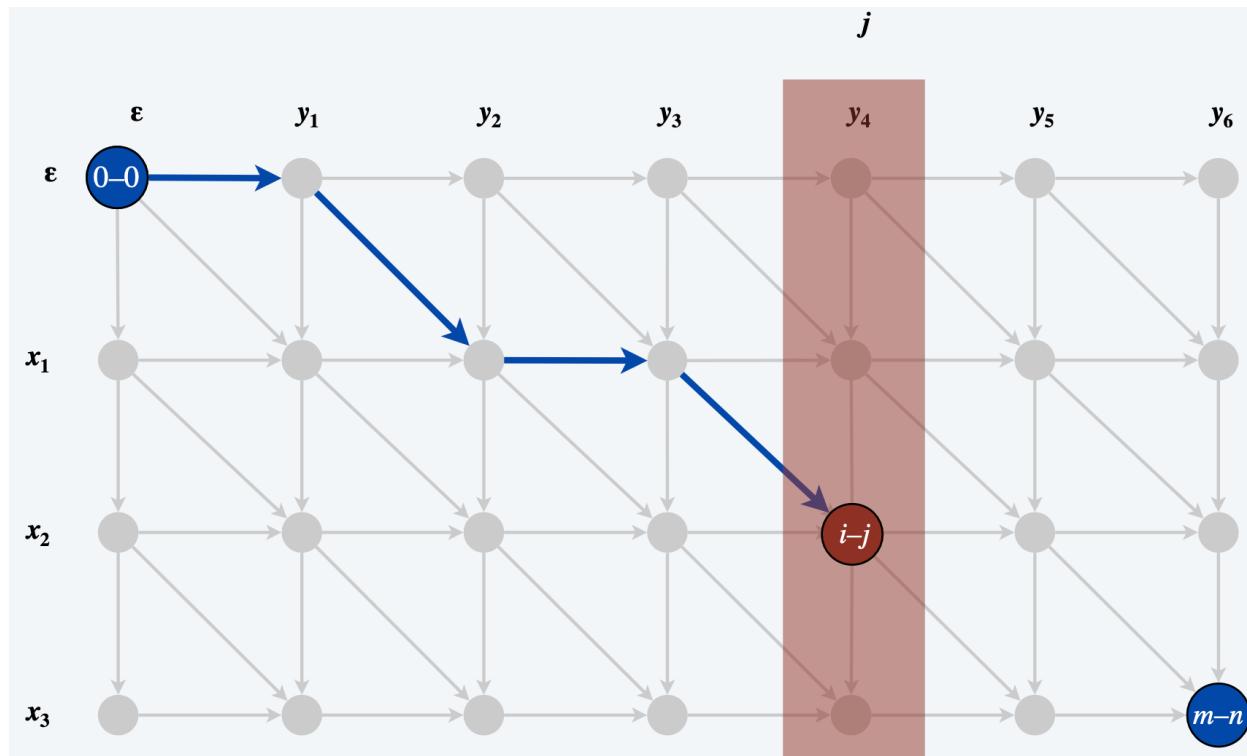
Bellman equation



Hirschberg's Algorithm

- To compute $OPT(m, n)$:

- Let $f(i, j)$ denote length of shortest path from $(0, 0)$ to (i, j) .
- Lemma: $f(i, j) = OPT(i, j)$ for all i and j .
- Can compute $f(\cdot, j)$ for any specific j in $O(mn)$ time and $O(m)$ space.

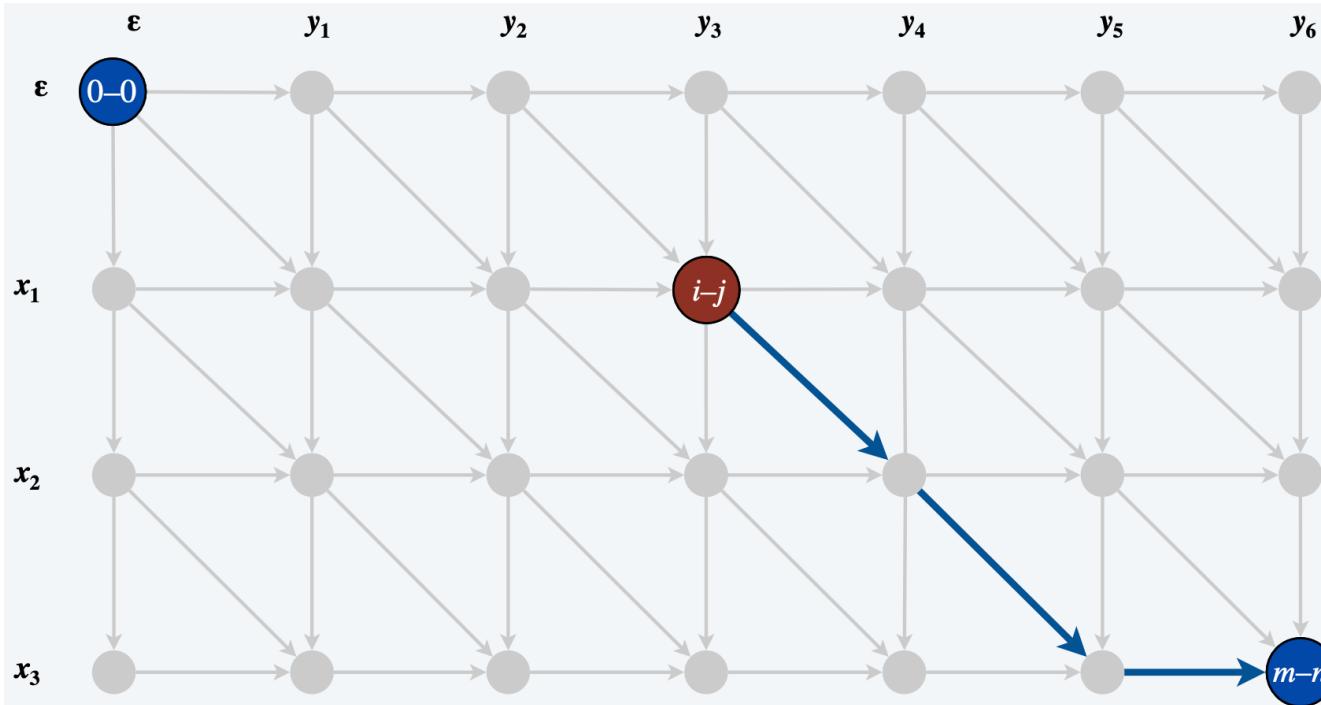


$f(\cdot, j)$ depends only on $f(\cdot, j - 1)$

Hirschberg's Algorithm

- To compute $OPT(m, n)$:

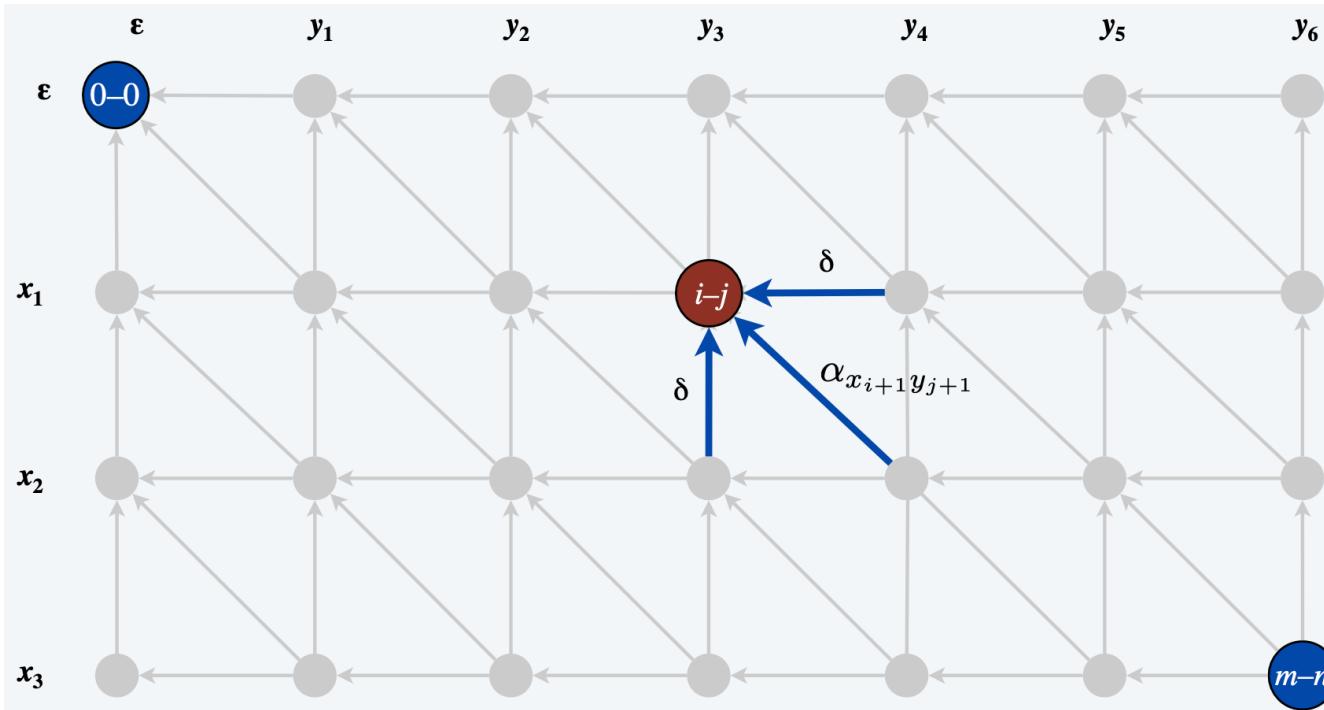
➤ Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .



Hirschberg's Algorithm

- To compute $OPT(m, n)$:

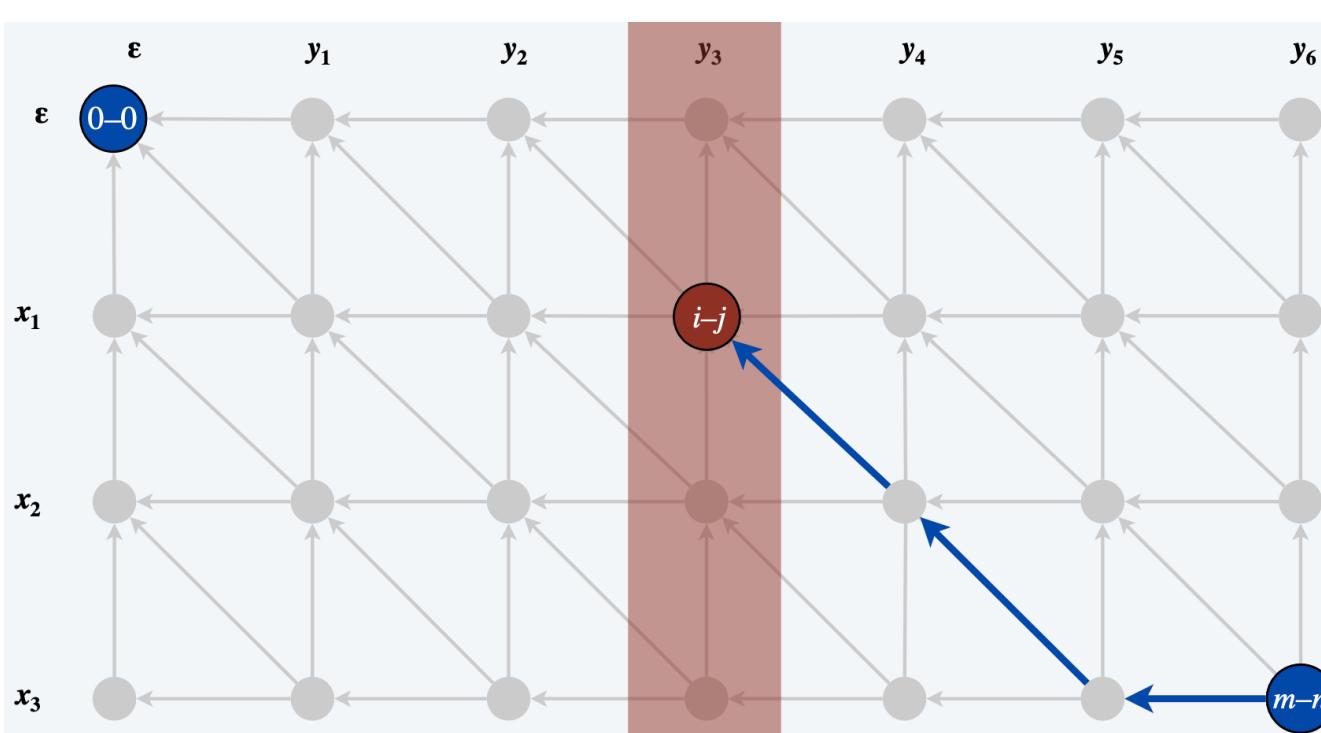
- Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .
- Can compute $g(i, j)$ by reversing edges and the roles of $(0, 0)$ and (m, n) .



Hirschberg's Algorithm

- To compute $OPT(m, n)$:

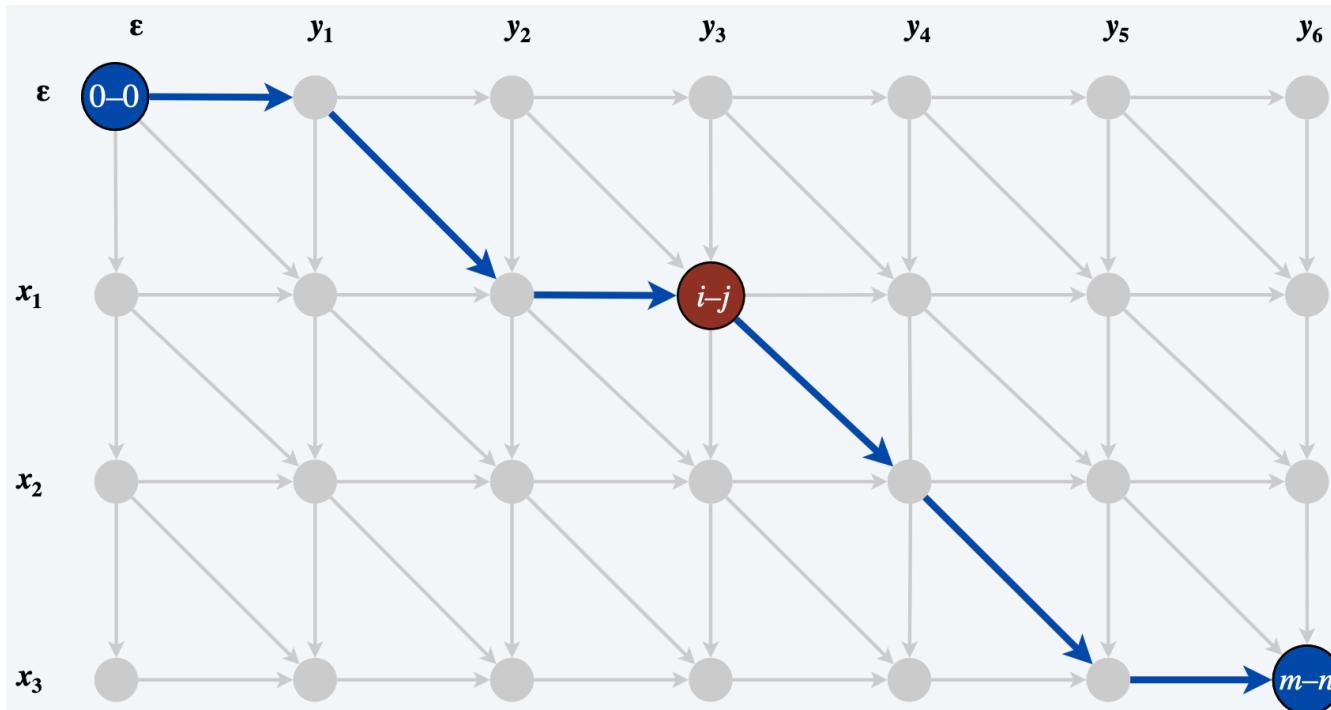
- Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .
- Can compute $g(i, j)$ by reversing edges and the roles of $(0, 0)$ and (m, n) .
- Can compute $g(\cdot, j)$ for any specific j in $O(mn)$ time and $O(m)$ space.



$g(\cdot, j)$ depends only on $g(\cdot, j + 1)$

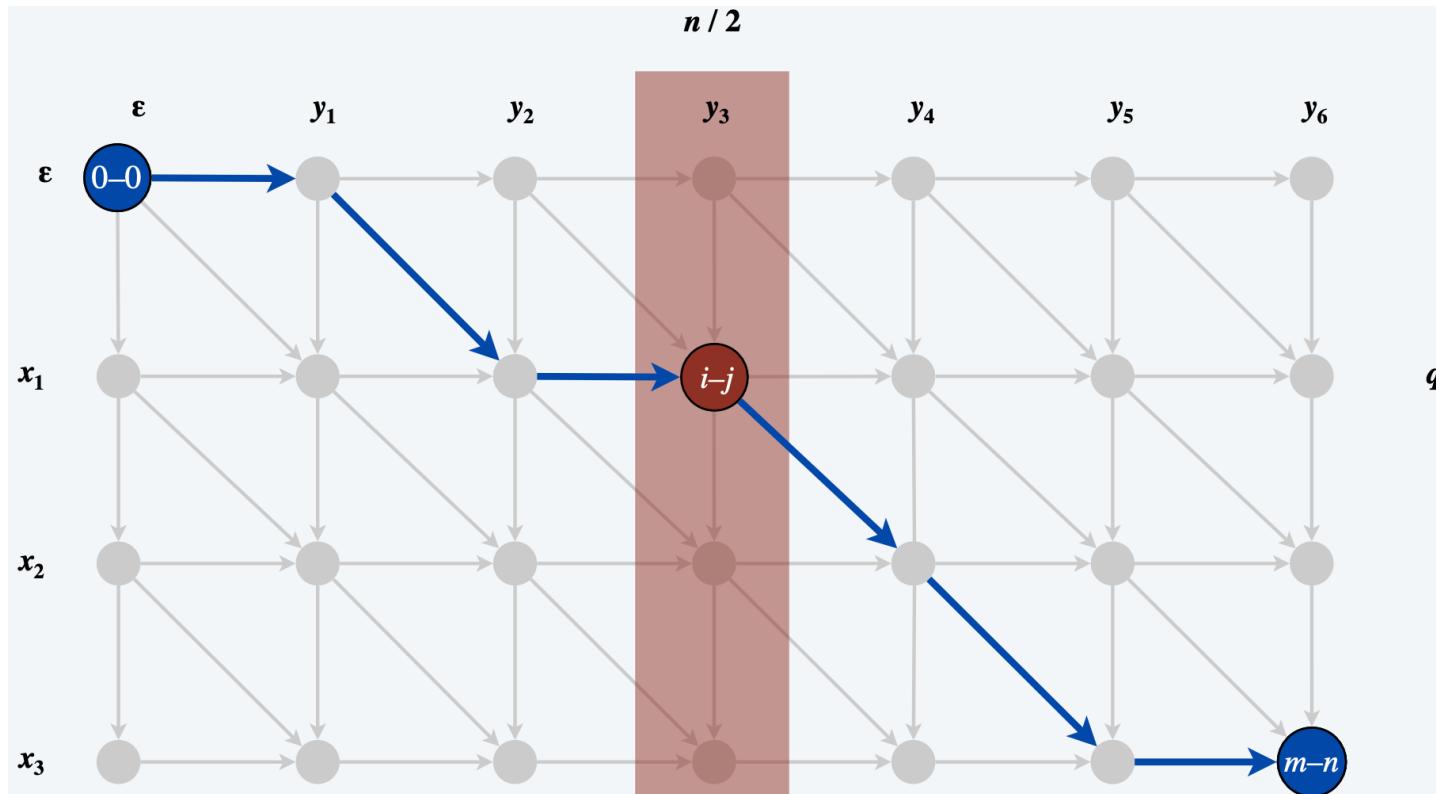
Hirschberg's Algorithm

- **Observation.** The length of a shortest path via (i, j) is $f(i, j) + g(i, j)$.



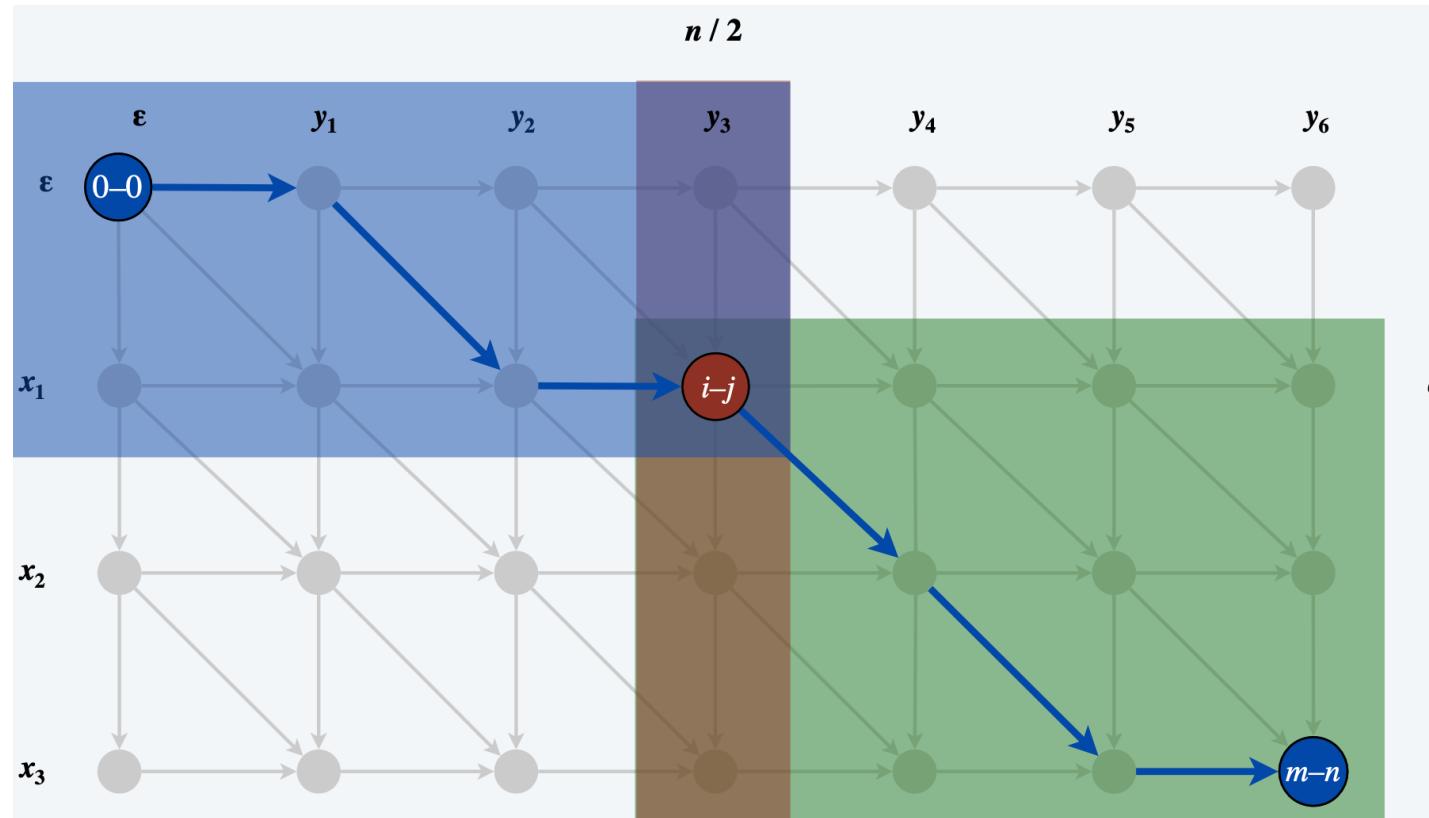
Hirschberg's Algorithm

- **Observation.** The length of a **shortest path** via (i, j) is $f(i, j) + g(i, j)$.
- **Observation.** 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) that uses $(q, n/2)$.



Hirschberg's Algorithm: Divide and Conquer

- **Divide.** Find index q that minimizes $f(q, n/2) + g(q, n/2)$.
 - Save this dividing node $(q, n/2)$ as part of solution.
- **Conquer.** Recursively compute optimal alignment in each piece.



每次递归只用存储中间列（前一列和当前列），之后可以释放空间

total problem size halved
after each recursion



Hirschberg's Algorithm: Space

- **Theorem.** Hirschberg's algorithm uses $\Theta(m + n)$ space.
- **Pf. (direct proof)**
 - Each recursive call uses $\Theta(m)$ space to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$.
 - $\Theta(1)$ space needs to be maintained per recursive call to save the dividing node.
 - Recursion stops when input size becomes 1.
 - Number of recursive calls $\leq 1 + 2 + 4 + 8 + \dots + n \leq 2n$.
 - So, $O(n)$ space in total. ▀
- **Q.** Does Hirschberg's algorithm also run in $O(mn)$ time?



Hirschberg's Algorithm: Running Time

- **Theorem.** Let $T(m, n) = \max$ running time of Hirschberg's algorithm on strings of lengths at most m and n . Then, $T(m, n) = O(mn \log n)$.
- **Pf. (direct proof)**
 - $T(m, n)$ is monotone nondecreasing in both m and n .
 - $T(m, n) \leq 2T(m, n/2) + O(mn) \Rightarrow T(m, n) = O(mn \log n)$ ▀
- **Q.** Can the above analysis of $T(m, n)$ be improved?
- **A.** Yes, because two subproblems are of size $(q, n/2)$ and $(m - q, n/2)$.



Hirschberg's Algorithm: Running Time

- **Theorem.** Hirschberg's algorithm runs in $T(m, n) = O(mn)$ time.
- **Pf.** (by strong induction on $m + n$)
 - $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find dividing node $(q, n/2)$.
 - $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
 - Choose constant $c > 0$ such that:
 - ✓ $T(m, 1) \leq cm$, $T(1, n) \leq cn$, $T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$
 - **Claim.** $T(m, n) \leq 2cmn$
 - ✓ Basis cases: $m = 1$ and $n = 1$.
 - ✓ Inductive hypothesis: $T(m, n) \leq 2cmn$ for all (m', n') with $m' + n' < m + n$.
 - ✓ Inductive case:
$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\ &\leq 2cqn/2 + 2c(m - q)n/2 + cmn = cqn + cmn - cqn + cmn = 2cmn \end{aligned}$$

inductive hypothesis



Exercise: Longest Common Subsequence

- **Problem.** Given two strings $X = x_1x_2\dots x_m$ and $Y = y_1y_2\dots y_n$, find a common subsequence that is as long as possible.
- **Alternative viewpoint.** Delete minimum number of characters from string X and string Y such that the resulting strings are the same.
- **Example.** $\text{LCS}(\text{GGCACCAACG}, \text{ACGGCGGATACG}) = \text{GGCAACG}$
- **Applications.** Unix diff, git, bioinformatics, ...
- **Q.** How to solve this problem via DP (in linear space)?



Longest Common Subsequence: Solution 1

- **Def.** $OPT(i, j)$ = length of **LCS** of prefix strings $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$
- **Goal.** $OPT(m, n)$
- **To compute $OPT(i, j)$:**
 - **Case 1:** $x_i = y_j$
 - ✓ 1 + length of **LCS** of prefix strings $x_1x_2 \dots x_{i-1}$ and $y_1y_2 \dots y_{j-1}$
 - **Case 2:** $x_i \neq y_j$
 - ✓ Delete x_i : length of **LCS** of prefix strings $x_1x_2 \dots x_{i-1}$ and $y_1y_2 \dots y_j$
 - ✓ Delete y_j : length of **LCS** of prefix strings $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_{j-1}$
- **Bellman equation.**

$$OPT(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + OPT(i - 1, j - 1) & \text{if } x_i = y_j \\ \max\{OPT(i - 1, j), OPT(i, j - 1)\} & \text{if } x_i \neq y_j \end{cases}$$



Longest Common Subsequence: Solution 2

- **Alternatively.** Reduce this **LCS** problem to finding an optimal alignment of $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$.
 - Gap penalty $\delta = 1$; mismatch penalty $\alpha_{pp} = 0$ and $\alpha_{pq} = \infty$.
 - Edit distance = min number of gaps = min number of deletions from X and Y .
 - Length of **LCS** = $(m + n - \text{edit distance}) / 2$.
- **Running time.** $O(mn)$ **Space.** $O(m + n)$ (e.g., run Hirschberg)
- **Lower bound.** [Abboud-Backurs-Williams 2015] There exists no algorithm that runs in $O(n^{2-\varepsilon})$ to find LCS of two length- n strings unless SETH is false.



Announcement

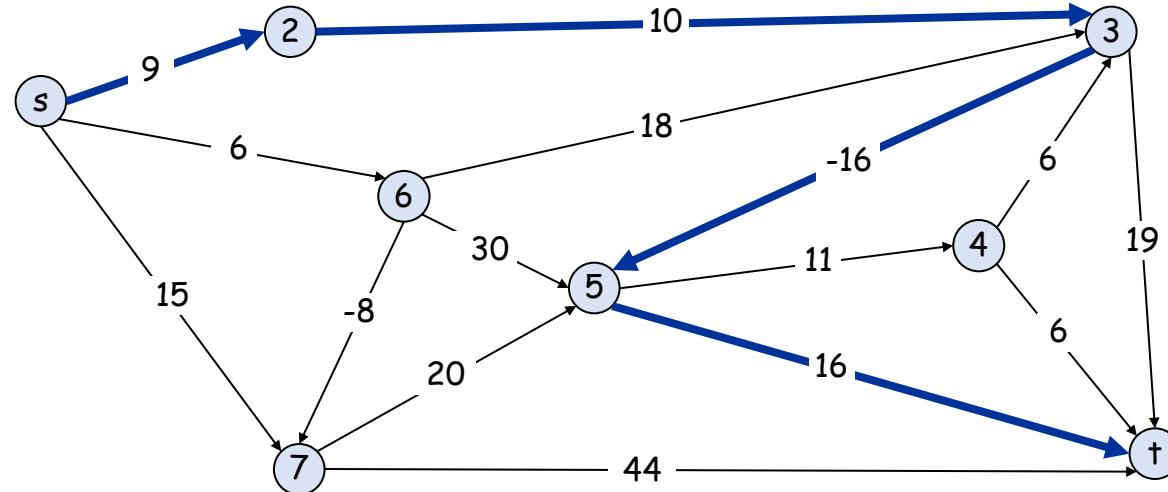
- Lab 9 will be released today and the deadline is Apr 29.



6. Shortest Paths with Negative Weights

Shortest Paths with Negative Weights

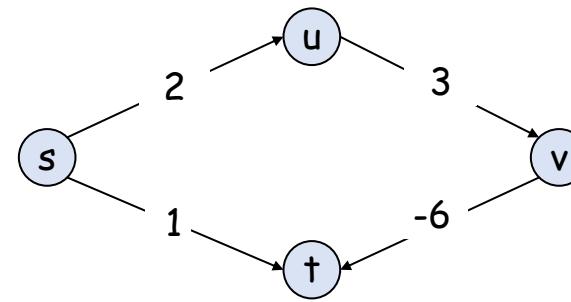
- **Shortest-path problem.** Given a digraph $G = (V, E)$, with **arbitrary** edge weights ℓ_{vw} , find **shortest path** from source node s to destination node t .
- **Example.** Nodes represent agents in a financial setting and ℓ_{vw} is cost of transaction in which we **buy** from agent v and **sell** immediately to w .



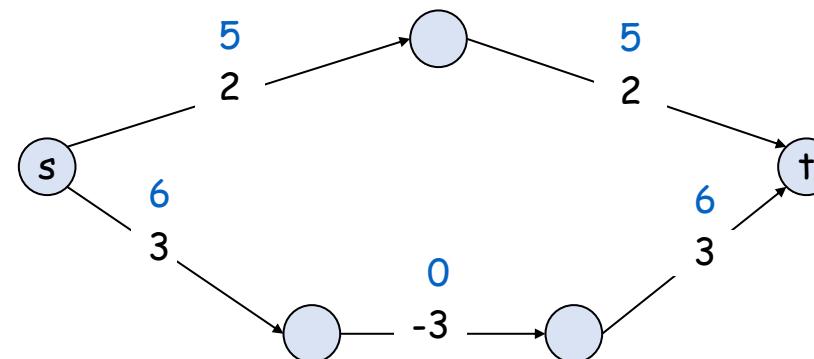


Shortest Paths: Failed Attempts

- **Dijkstra.** Can fail if there exist **negative** edge weights.

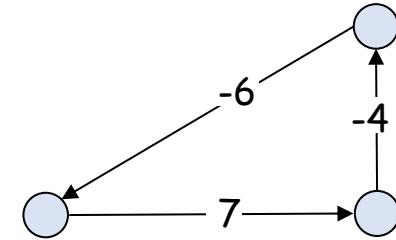


- **Reweighting.** Adding a **constant** to every edge weight can still fail.

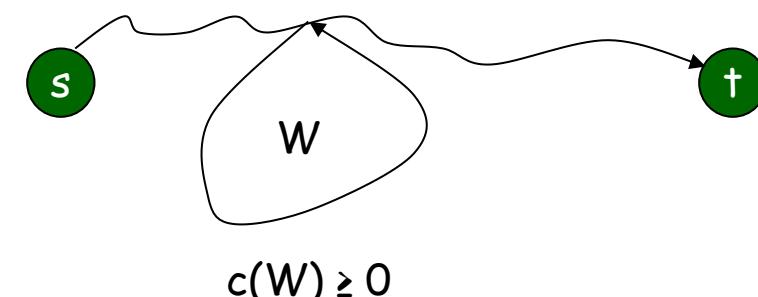
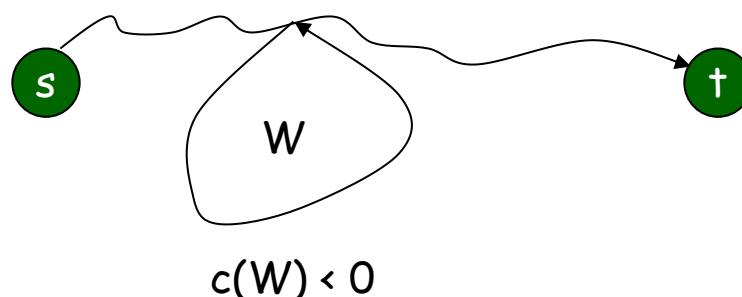


Negative Cycles

- **Def.** A **negative cycle** is a directed cycle for which the sum of its edge lengths is negative.



- **Observation 1.** If some s - t path contains a negative cycle, then there does **not exist** a shortest s - t path.
- **Observation 2.** If there exists **no negative cycle**, there **exists** a shortest s - t path that is simple (and has $\leq n - 1$ edges).





Shortest-Paths Problem with Negative Weights

- **Single-destination shortest-paths problem.** Given a digraph $G = (V, E)$, with arbitrary edge weights ℓ_{vw} (but no negative cycles) and a destination node t , find a shortest $v-t$ path from every node v to t .

assume there exists a path from every node to t
- **Remark.** This single-destination shortest-paths problem is equivalent to single-source shortest-paths problem with edge directions reversed.



Shortest Paths: Dynamic Programming

- **Def.** $OPT(i, v) =$ length of shortest $v-t$ path P using $\leq i$ edges.
- **Goal.** $OPT(n - 1, v)$ if no neg cycles, there exists a simple shortest path of length $\leq n - 1$
- **To compute $OPT(i, v)$:** 从v这个点走到终点t, 用不超过i条边
 - Case 1: P uses $\leq i - 1$ edges.
✓ $OPT(i, v) = OPT(i - 1, v)$
 - Case 2: P uses exactly i edges.
✓ Let (v, w) denote the first edge in P : pay the cost of e_{vw} , then select the shortest $w-t$ path using $\leq i - 1$ edges. 所有可能的w



Shortest Paths: Dynamic Programming

- **Def.** $OPT(i, v) =$ length of shortest $v-t$ path P using $\leq i$ edges.
- **Goal.** $OPT(n - 1, v)$ if no neg cycles, there exists a simple shortest path of length $\leq n - 1$
- **Bellman equation:**

$$OPT(i, v) = \begin{cases} 0, & \text{if } i = 0 \text{ and } v = t \\ \infty, & \text{if } i = 0 \text{ and } v \neq t \\ \min\{OPT(i - 1, v), \min_{w \in V}(OPT(i - 1, w) + l_{vw})\} & \text{if } i > 0 \end{cases}$$

最后得到所有点到终点的最短路径



Shortest Paths: Algorithm

- Bottom-up DP algorithm:

```
Shortest-Path(G, t) {
    foreach node v ∈ V
        M[0, v] = ∞
    M[0, t] = 0

    for i = 1 to n - 1
        foreach node v ∈ V
            M[i, v] = M[i - 1, v]
            foreach edge (v, w) ∈ E
                M[i, v] = min{ M[i, v], M[i - 1, w] + ℓvw }
}
```

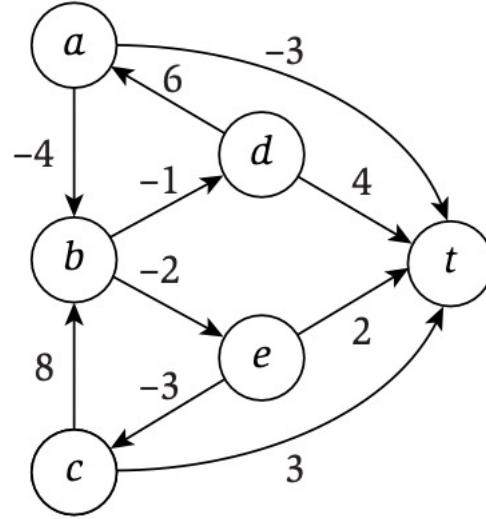
pull every neighbor w
even if w not updated

存在冗余操作：因为只有上一轮这个更新过，当前轮才需要判断是否要更新

- Running time. $O(mn)$ Space. $O(n^2)$



Shortest-Paths DP Algorithm: Demo



(a)

	0	1	2	3	4	5
<i>t</i>	0	0	0	0	0	0
<i>a</i>	∞	-3	-3	-4	-6	-6
<i>b</i>	∞	∞	0	-2	-2	-2
<i>c</i>	∞	3	3	3	3	3
<i>d</i>	∞	4	3	3	2	0
<i>e</i>	∞	2	0	0	0	0

(b)

$$OPT(i, v) = \begin{cases} 0, & \text{if } i = 0 \text{ and } v = t \\ \infty, & \text{if } i = 0 \text{ and } v \neq t \\ \min\{OPT(i - 1, v), \min_{w \in V}(OPT(i - 1, w) + l_{vw})\} & \text{if } i > 0 \end{cases}$$



Shortest Paths: Finding Shortest Paths

- **Finding the shortest paths:**

- Approach 1: Maintain 2D array $\text{successor}[i, v]$ that points to the next node on a shortest $v-t$ path using $\leq i$ edges.
- Approach 2: Compute optimal lengths $M[i, v]$ and consider only edges with $M[i, v] = M[i - 1, w] + \ell_{vw}$. Any directed path in this subgraph is a shortest path.

两种都需要 n^2 的空间

- Q. Can we reduce the required space?

$$OPT(i, v) = \begin{cases} 0, & \text{if } i = 0 \text{ and } v = t \\ \infty, & \text{if } i = 0 \text{ and } v \neq t \\ \min\{OPT(i - 1, v), \min_{w \in V}(OPT(i - 1, w) + l_{vw})\} & \text{if } i > 0 \end{cases}$$

only $OPT[i - 1, .]$ is used!

找到最短距离可以只靠上一次迭代，空间上也只需要O



Shortest Paths: Practical Improvements

- **Space optimization.** Maintain two **1D** arrays (instead of **2D** arrays).
 - $d[v]$ = length of a **shortest $v-t$ path** that we have found so far
 - $\text{successor}[v]$ = next node on a **$v-t$ path** (used to find shortest paths)
- **Remark.** This space-efficient algorithm is called **Bellman-Ford-Moore**.
- **Performance optimization.** If $d[w]$ was **not updated** in iteration $i-1$, then no need to consider edges entering w in iteration i .
- **Remark.** This time-efficient implementation of Bellman-Ford-Moore is also known as the **shortest path faster algorithm (SPFA)**.



Bellman-Ford-Moore: Efficient Implementation

- Push-based DP algorithm with linear extra space:

```
Bellman-Ford-Moore(G, s, t) {
    foreach node v ∈ V
        d[v] = ∞
        successor[v] = null
    d[t] = 0

    for i = 1 to n - 1
        foreach node w ∈ V
            if (d[w] has been updated in previous iteration)
                foreach node v such that (v, w) ∈ E
                    if (d[v] > d[w] + ℓvw)
                        d[v] = d[w] + ℓvw
                        successor[v] = w
                        push each v that has the
                        updated w as a neighbor
            if (no d[w] value changed in iteration i)
                break
}
```

SPFA optimizes this for loop by keeping nodes updated in previous pass in a queue

一个点只有当它的邻居更新过，才可能需要更新

把更新过的节点放在队列中

terminate when no d[] update occurs

O(n) extra space

O(m) for each pass



Bellman-Ford-Moore: Analysis

- **Lemma 1.** After pass i , $d[v] \leq$ length of a shortest $v\text{-}t$ path using $\leq i$ edges.
- **Pf. (by induction on i)**
 - Basis case: $i = 0$.
 - Inductive hypothesis: assume true after pass i .
 - **Inductive case:** Let P be any $v\text{-}t$ path with $\leq i + 1$ edges.
 - ✓ Let (v, w) be first edge in P and let P' be subpath from w to t .
 - ✓ By **inductive hypothesis**, because P' is a $w\text{-}t$ path with $\leq i$ edges, at the end of pass i we have $d[w] \leq \ell(P')$.
 - ✓ After considering edge (v, w) in pass $i + 1$ (here w.l.o.g. consider pull-based):
 $d[v] \leq \ell_{vw} + d[w]$. Then, since $d[w] \leq \ell(P')$ after pass i and $d[w]$ never increases during the algorithm, we have $d[v] \leq \ell_{vw} + \ell(P') = \ell(P)$. ▀

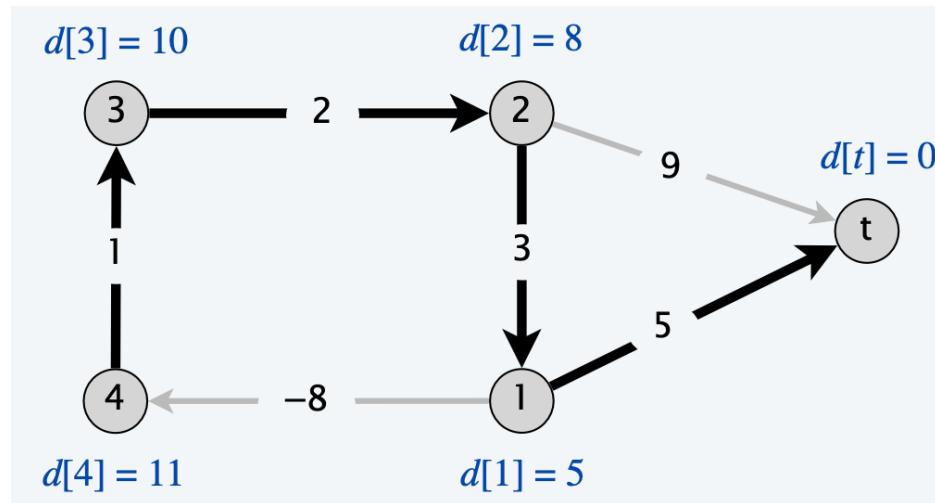


Bellman-Ford-Moore: Analysis

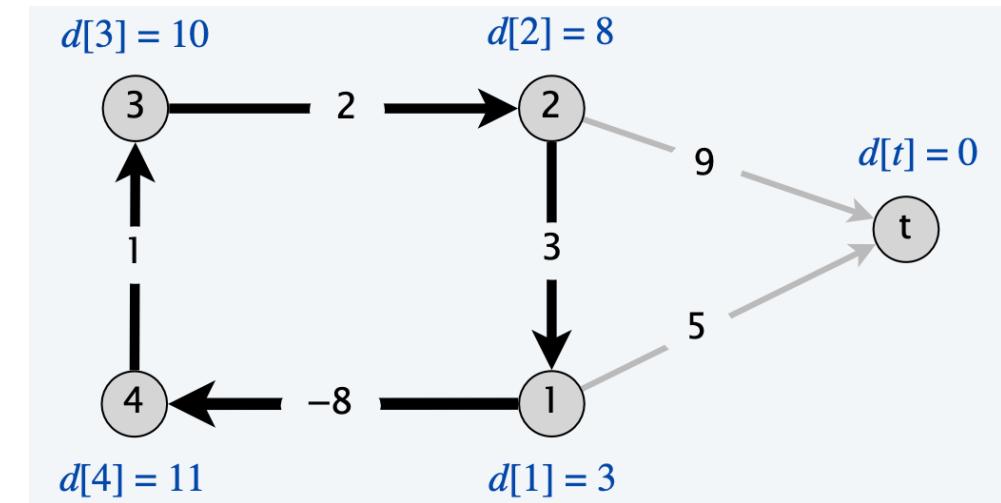
- **Theorem.** If no negative cycles, Bellman-Ford-Moore computes the lengths of the shortest $v-t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.
- **Pf.** (only left to show $d[v] = \text{shortest } v-t \text{ path length}$)
 - **Observation 2:** If no negative cycles, shortest path exists and has $\leq n - 1$ edges.
 - **Lemma 1:** After pass $n - 1$, $d[v] \leq$ length of shortest $v-t$ path using $\leq n - 1$ edges.
- **Remark.** Bellman-Ford-Moore is typically faster in practice.
 - Edge (v, w) is considered in pass $i + 1$ only if $d[w]$ was updated in pass i .
 - If shortest path has k edges, then algorithm finds it after $\leq k$ passes.
- **Q.** How to find a shortest $v-t$ path of length $d[v]$ for every node v ?

Bellman-Ford-Moore: Finding Shortest Paths

- **Def.** A successor graph consists of only edges $\{(v, \text{successor}[v])\}$.
- **Claim.** [This claim is false!] After pass $n - 1$ of Bellman-Ford-Moore, the successor graph gives a directed path from v to t of length $d[v]$.
- **Counterexample.** A successor graph may have **directed cycles**.
 - E.g., consider nodes updated in order: $t, 1, 2, 3, 4$



before updating 1



after updating 1



Bellman-Ford-Moore: Finding Shortest Paths

- **Lemma 2.** Any directed cycle W in successor graph is a negative cycle.
- **Pf. (direct proof)**
 - If $\text{successor}[v] = w$, we have $d[v] \geq d[w] + \ell_{vw}$. (They are equal when $\text{successor}[v]$ is set; $d[w]$ can only decrease; $d[v]$ decreases only when $\text{successor}[v]$ is reset.)
 - Let $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$ be the sequence of nodes in a directed cycle W .
 - Assume that (v_k, v_1) is the last edge in W added to the successor graph.
 - Just prior to that:

$d[v_1]$	\geq	$d[v_2]$	$+ \ell(v_1, v_2)$
$d[v_2]$	\geq	$d[v_3]$	$+ \ell(v_2, v_3)$
\vdots	\vdots	\vdots	
$d[v_{k-1}]$	\geq	$d[v_k]$	$+ \ell(v_{k-1}, v_k)$
$d[v_k]$	$>$	$d[v_1]$	$+ \ell(v_k, v_1)$

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



Bellman-Ford-Moore: Finding Shortest Paths

- **Theorem.** Assuming no negative cycles, Bellman-Ford-Moore finds a shortest $v-t$ path for every node v in $O(mn)$ time and $\Theta(n)$ extra space.
- **Pf. (direct proof)**
 - Lemma 2 + no neg cycles \Rightarrow the successor graph cannot have a directed cycle.
 - Thus, following the successor pointers from v yields a directed path to t .
 - Let $v = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = t$ be the nodes along this path P .
 - Upon termination, if $\text{successor}[v] = w$, then $d[v] = d[w] + \ell_{vw}$. (LHS and RHS are equal when $\text{successor}[v]$ is set; $d[\cdot]$ did not change since algorithm terminated.)
 - Thus,
$$\begin{aligned}d[v_1] &= d[v_2] + \ell(v_1, v_2) \\d[v_2] &= d[v_3] + \ell(v_2, v_3) \\&\vdots &&\vdots \\d[v_{k-1}] &= d[v_k] + \ell(v_{k-1}, v_k)\end{aligned}$$

P is a shortest path of length $d[v]$

$d[t] = 0$

$d[v] = d[t] + \ell(v_1, v_2) + \ell(v_2, v_3) + \dots + \ell(v_{k-1}, v_k)$
 - Adding above equations yields $d[v] = d[t] + \ell(v_1, v_2) + \ell(v_2, v_3) + \dots + \ell(v_{k-1}, v_k)$.



Shortest Paths: Asymptotic Complexity

year	worst case	discovered by
1955	$O(n^4)$	Shimbel
1956	$O(m n^2 W)$	Ford
1958	$O(m n)$	Bellman, Moore
1983	$O(n^{3/4} m \log W)$	Gabow
1989	$O(m n^{1/2} \log(nW))$	Gabow–Tarjan
1993	$O(m n^{1/2} \log W)$	Goldberg
2005	$O(n^{2.38} W)$	Sankowski, Yuster–Zwick
2016	$\tilde{O}(n^{10/7} \log W)$	Cohen–Mądry–Sankowski–Vladu
20xx	???	

single-source shortest paths with weights between $-W$ and W



7. Distance-Vector Protocols



Shortest-Paths Algorithms for Routing

- **Routing in a communication network.**
 - Node \approx router
 - Edge \approx direct communication link
 - Cost of edge \approx latency of link
- **Dijkstra's algorithm.** Requires **global** information of network.
每个路由器只要知道周围路由器的路径长度就行
- **Bellman-Ford-Moore.** Uses only **local** knowledge of neighboring nodes.
- **Synchronization.** We don't expect routers to run in lockstep. The **order** in which each edges are processed in **Bellman-Ford-Moore** is **not important**. Moreover, algorithm converges even if updates are **asynchronous**.



Asynchronous Shortest-Paths Algorithm

- **Asynchronous DP algorithm:**

```
Asynchronous-Bellman-Ford-Moore(G, s, t) {
    foreach node v ∈ V
        d[v] = ∞
        successor[v] = null
    d[t] = 0
    Declare t to be active and all other nodes inactive

    while there exists an active node ← no global in-order for loop
        Choose an active node w for asynchronous version
        foreach node v such that (v, w) ∈ E
            if (d[v] > d[w] + lvw)
                d[v] = d[w] + lvw
                successor[v] = w
                Declare v to be active
                变得更短就广播信息
        Declare w to be inactive
}
```

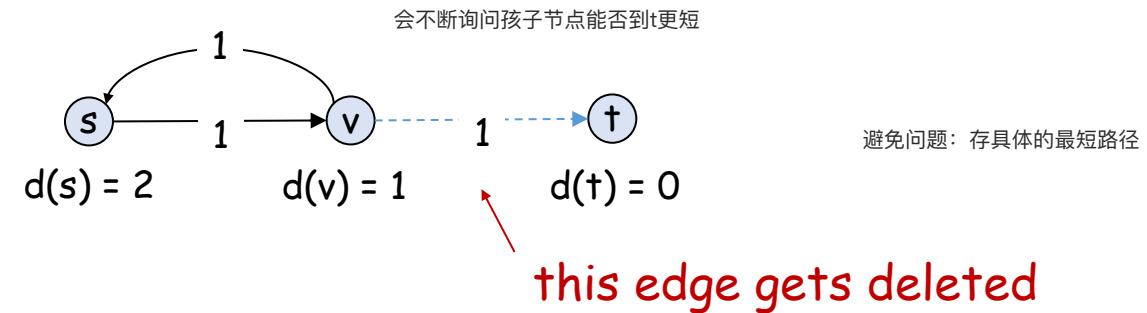
只需要知道周围的点 每个节点都是单独运行这个算法的



Distance-Vector Routing Protocols

- **Distance-vector routing protocols.** “routing by rumor”
 - Each router maintains a **vector** of shortest path lengths (**distances**) to **every other node** and the **first hops** (directions) on each path.
 - Algorithm: each router performs ***n* separate** computations, one for each potential destination node, using information **pushed** by neighboring routers.
- **Example applications.** 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 and Link-State Routing Protocols

- **Path-vector routing protocols:**

- Each router also stores the **entire path.**
- Requires **significantly more storage.**
- Avoids the “counting-to-infinity” problem and related difficulties.
- Application: Border Gateway Protocol (BGP)

not just the distance and first hop

- **Link-state routing protocols:**

- Each router stores the **entire network topology.**
- Requires **even more storage.**
- Independently calculates shortest paths based on Dijkstra’s algorithm.
- Applications: Open Shortest Path First (OSPF), Intermediate System to Intermediate System (IS-IS)

跨自治系统

这个可以直接dijkstra了

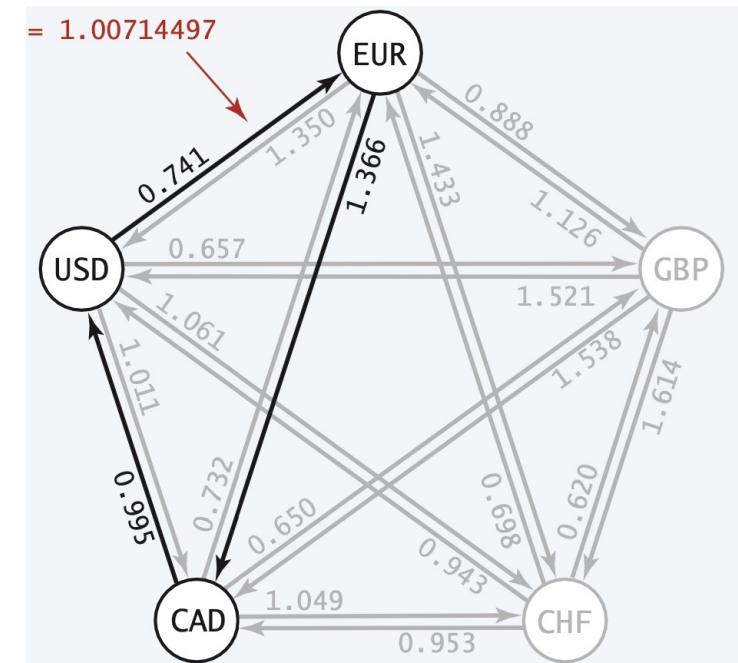
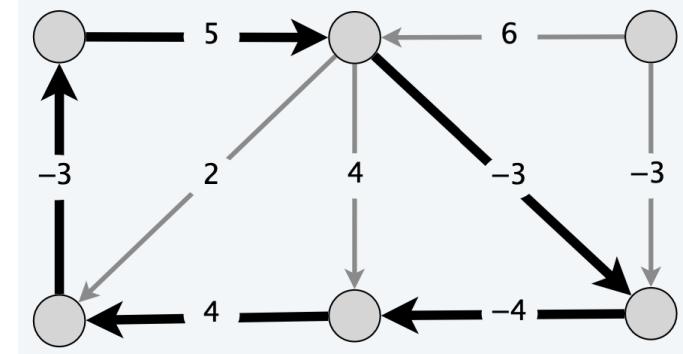


南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

8. Negative Cycles

Detecting Negative Cycles

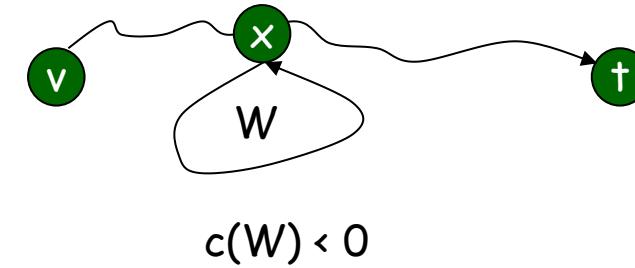
- **Negative cycle detection problem.**
 Given a directed graph $G = (V, E)$, with arbitrary edge weights ℓ_{vw} , find a negative cycle (if one exists).
- **Example. [Currency conversion]** Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?
 - Fastest algorithm literally very valuable!
- **Q.** Can you convert this problem to a negative cycle detection problem?





Detecting Negative Cycles

- **Lemma 3.** If $OPT(n, v) = OPT(n - 1, v)$ for every v , then no negative cycles.
 - Note: Lemma 3 assumes there exists a directed path from every node to t .
- **Pf.** The $OPT(n, v)$ values have converged \Rightarrow shortest $v-t$ path exists. ▀
- **Lemma 4.** If $OPT(n, v) < OPT(n - 1, v)$ for some v , then (any) shortest $v-t$ path of length n contains a cycle W . Moreover, W is a negative cycle.
- **Pf. (by contradiction)**
 - $OPT(n, v) < OPT(n - 1, v) \Rightarrow$ shortest $v-t$ path P has exactly n edges.
 - By pigeonhole principle, the path P must contain a repeated node x .
 - Let W be any cycle in P .
 - Deleting W yields a $v-t$ path with $< n$ edges.
 - Therefore, W is a negative cycle. ▀





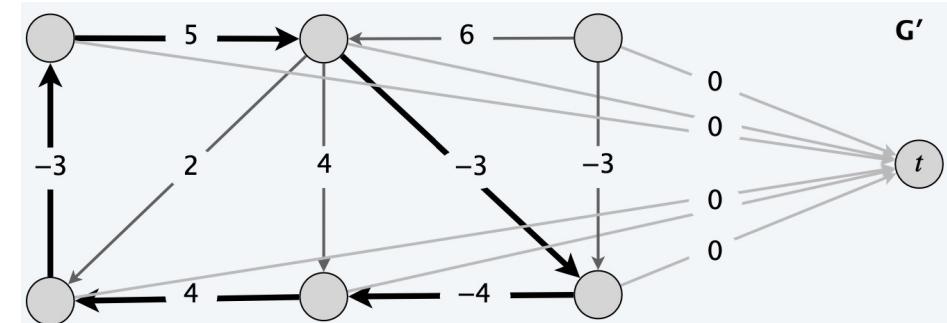
Detecting Negative Cycles

- **Theorem 1.** Can find a negative cycle in $O(mn)$ time and $O(n^2)$ space.

- **Pf. (constructive proof)**

- Construct graph G' by adding a new sink node t and connect all nodes to t with 0-length edges.
 - ✓ G' has $n' = n + 1$ nodes.
 - ✓ Every node has a directed path to t .
 - ✓ G has a negative cycle if and only if G' has a negative cycle.
- Case 1: $OPT(n', v) = OPT(n' - 1, v)$ for every node v .
 - ✓ Lemma 3 + every node can reach $t \Rightarrow$ there exist no negative cycles.
- Case 2: $OPT(n', v) < OPT(n' - 1, v)$ for some node v .
 - ✓ Lemma 4 \Rightarrow can extract negative cycle from the $v-t$ path of length exactly n' (cycle cannot contain t since no edge leaves t). ▀

多跑一轮找负环



负环不能包含t



Detecting Negative Cycles

- **Theorem 2.** Can find a **negative cycle** in $O(mn)$ time and $O(n)$ extra space.
- **Pf. (constructive proof)**
 - Construct graph G' as before.
 - Run **Bellman-Ford-Moore** on G' for $n' = n + 1$ passes (instead of n).
 - If no $d[v]$ values updated in pass n' , then no negative cycles.
 - Otherwise, suppose $d[s]$ was updated in pass n' .
 - Let $\text{pass}(v) = \text{last pass } (\leq n') \text{ in which } d[v] \text{ was updated}$.
 - Observe that $\text{pass}(s) = n'$, $\text{pass}(t) = 0$, and $\text{pass}(\text{successor}[v]) \geq \text{pass}(v) - 1$ for each $v \neq t$ (here w.l.o.g. consider push-based).
 - Following **successor pointers** ($\geq n'$ edges), we must eventually **repeat** a node.
 - **Lemma 2** \Rightarrow the corresponding cycle is a **negative cycle**. ▀



More on Detecting Negative Cycles

- **Performance optimization:**
 - SPFA can be improved by using **stack (DFS)** instead of **queue (BFS)**.
 - Tarjan's subtree disassembly trick: a practical speedup for **shortest paths** and **negative cycle detection** algorithms. [Page 304-307 of textbook]
- **Q.** Can we find **negative cycles** in (simple) **undirected** graphs?
 - A negative cycle in simple undirected graphs contains at least **3** edges.
- **A.** Yes, but **more difficult** (e.g., using **T-joins**). This same technique can be used to find **simple shortest paths** in undirected graphs that does not contain negative cycles.
- **Remark.** Some graph problems could be more challenging to solve in **undirected** graphs than in **directed** graphs.

很难在无向图和有向图之间做等价转换



Announcements

- Assignment 4 has been released and the deadline is May 13.
- Lab 10 will be released today and the deadline is May 13.