

Homework 1

Homework 1


- Covers chapters 2, 4, 5, 6
- To be released today in Blackboard
- Due: 11:59pm, Mar. 23
- No late homework will be accepted!

▼ 算法设计与分析

主页

公告

Slides

Homework 

Piazza

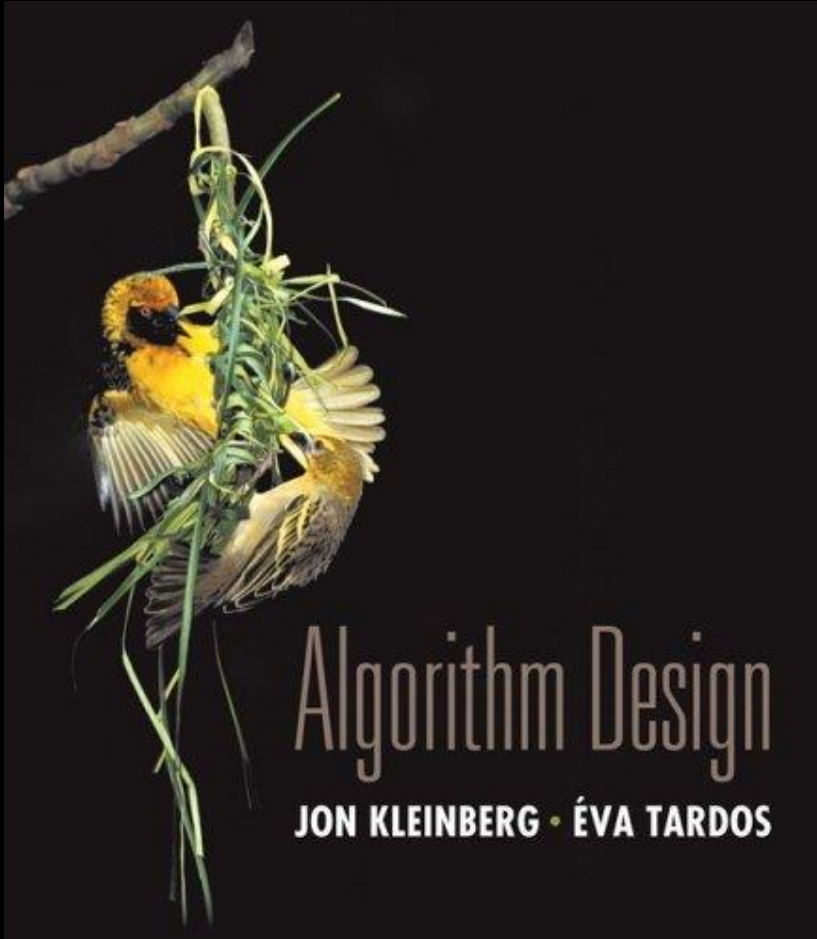


Submission

- Register at gradescope.com using your university email
 - Make sure that your FULL NAME is your Chinese name
- Enroll in this course with code [9G8Y6E](#)
- Save your homework solutions in a PDF or image file
- Upload your file to Gradescope and match your solution to each problem

Chapter 6

Dynamic Programming



Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

Algorithmic Paradigms

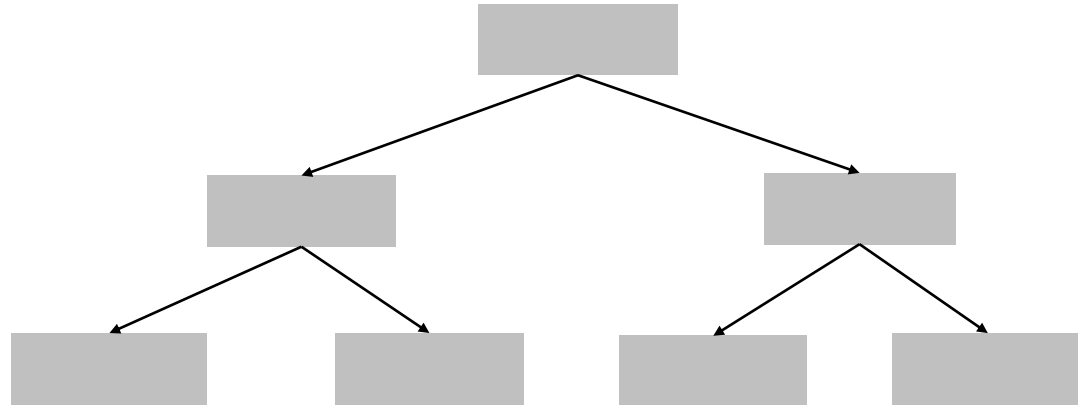
Greed. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into a few sub-problems, solve each sub-problem independently and recursively, and combine solution to sub-problems to form solution to original problem.

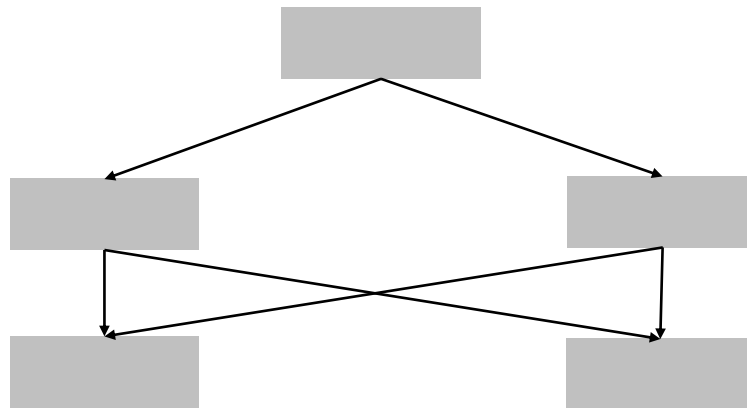
Dynamic programming. Break up a problem into a series of **overlapping** sub-problems, and build up solutions to larger and larger sub-problems.

Divide-and-conquer VS. Dynamic programming

Divide-and-conquer



Dynamic programming

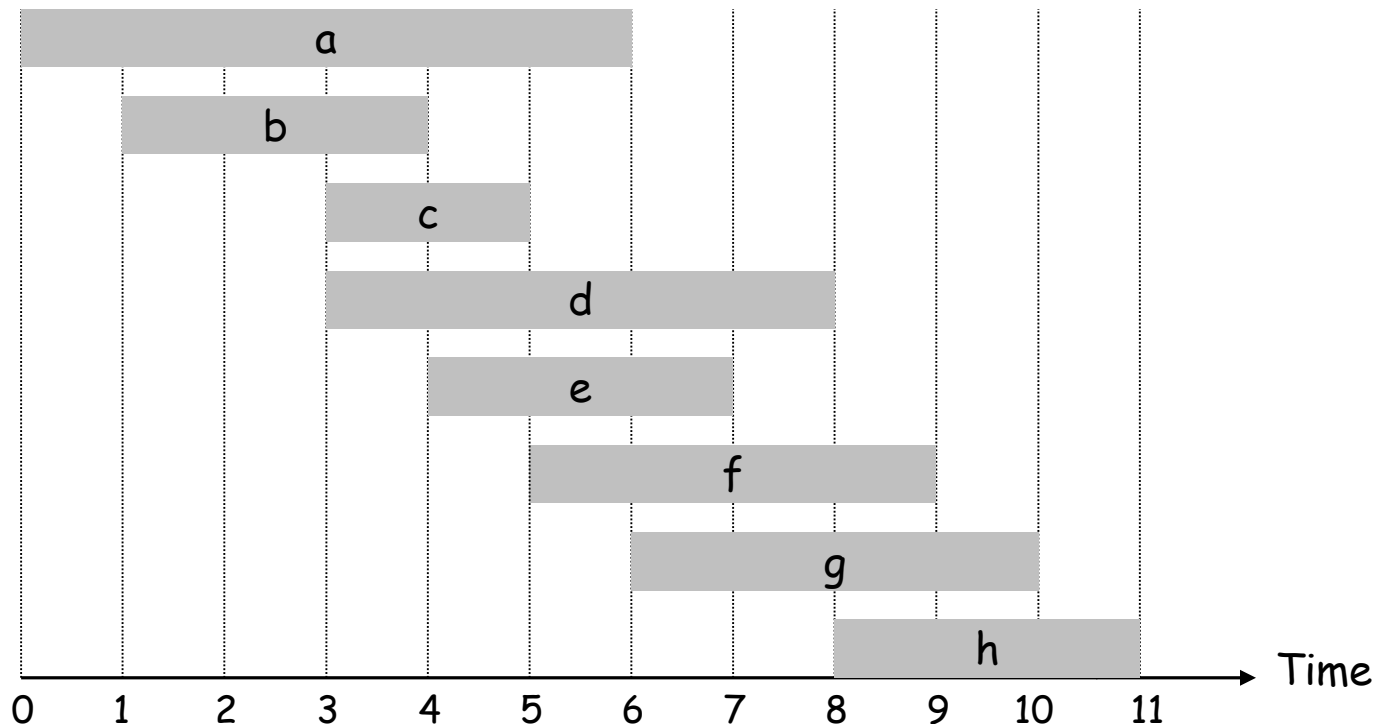


6.1 Weighted Interval Scheduling

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.

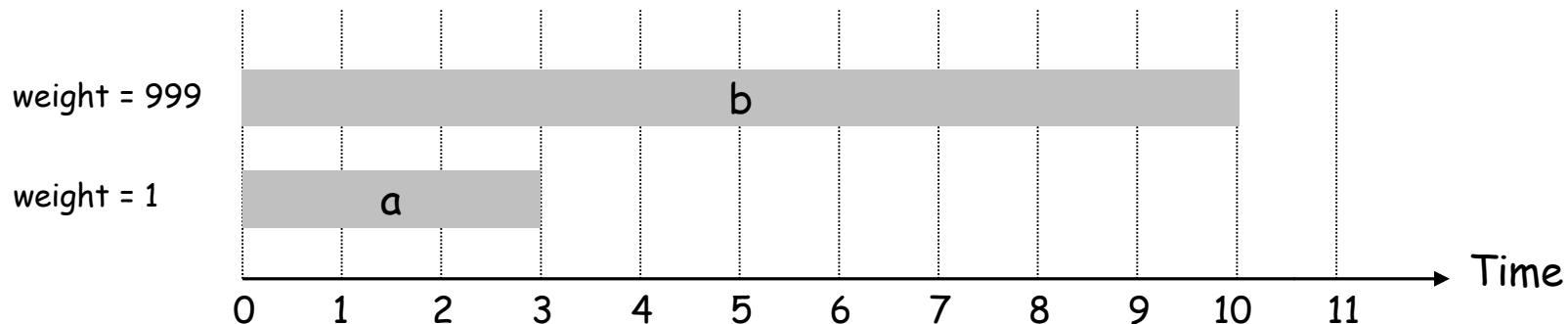


Unweighted Interval Scheduling Review

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 can fail spectacularly if arbitrary weights are allowed.

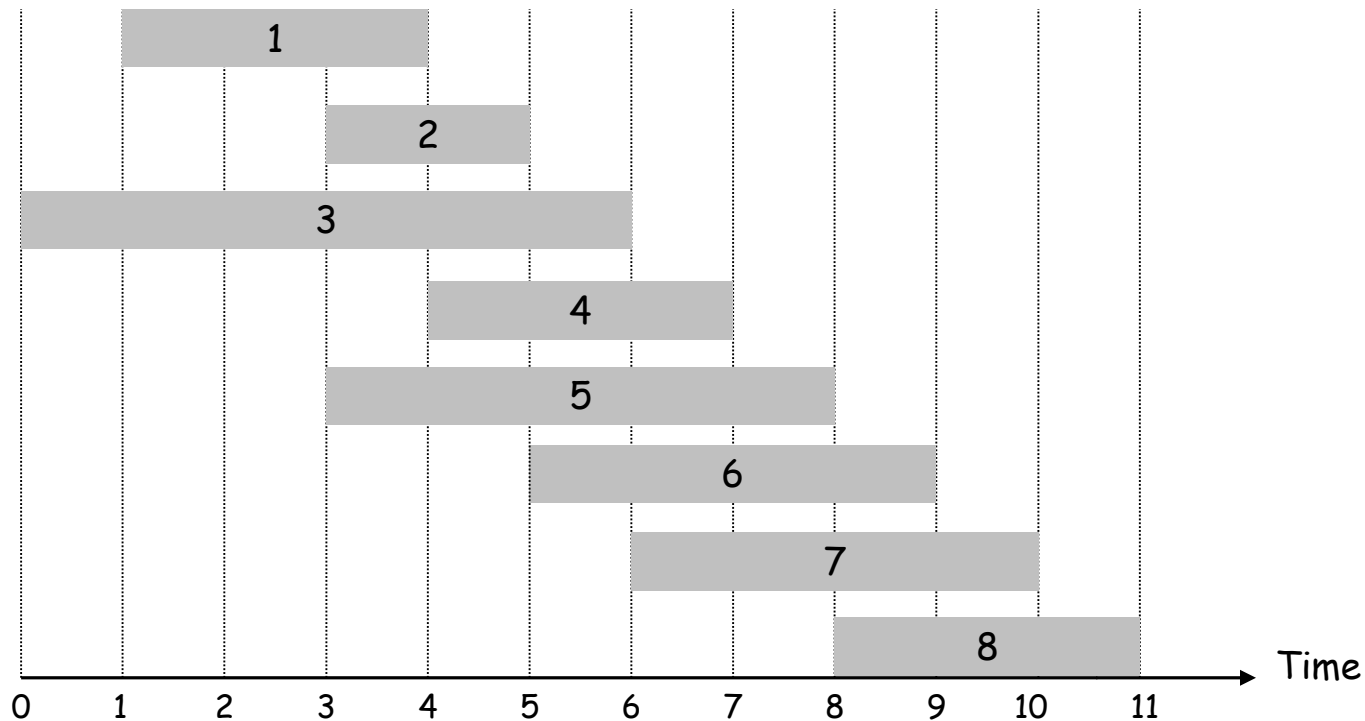


Weighted Interval Scheduling

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

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, ..., j.

- Case 1: OPT selects job j.
 - 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, ..., $p(j)$
- Case 2: OPT does not select job j.
 - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $j-1$

↖
↙
optimal substructure

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

Case 1

Case 2

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

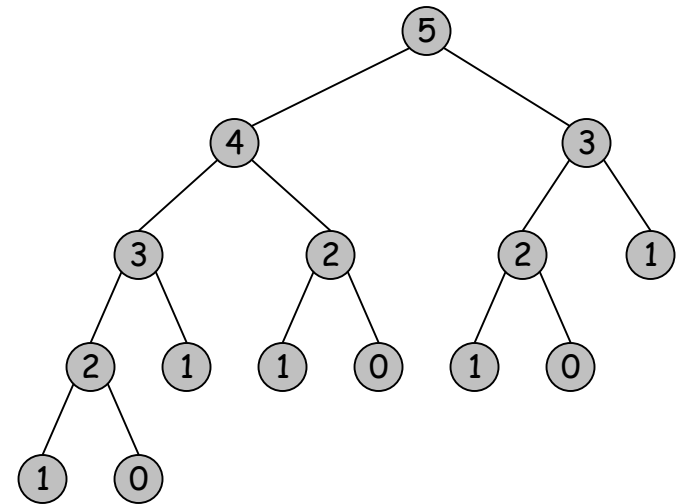
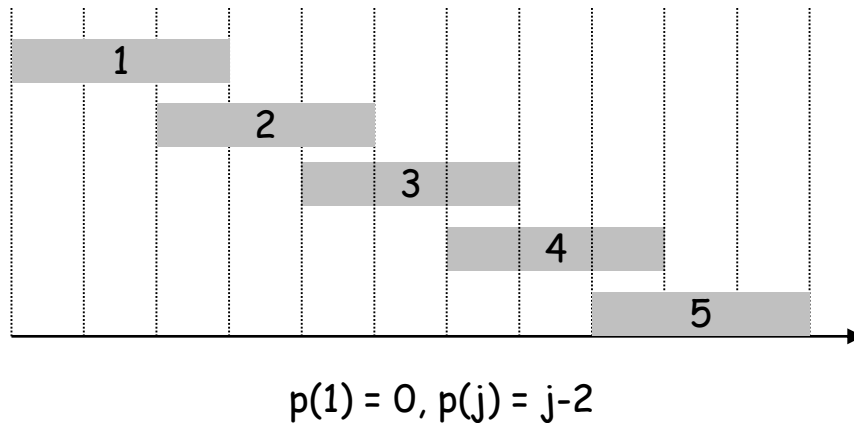
Compute $p(1), p(2), \dots, p(n)$

```
Compute-Opt(j) {  
    if (j = 0)  
        return 0  
    else  
        return max( $v_j + \text{Compute-Opt}(p(j))$ ,  $\text{Compute-Opt}(j-1)$ )  
}
```

Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems \Rightarrow exponential algorithms.

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



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p(1), p(2), \dots, p(n)$

for $j = 1$ to n

$M[j] = \text{empty}$ \leftarrow global array

$M[0] = 0$

M-Compute-Opt(j) {

if ($M[j]$ is empty)

$M[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$

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 $p(\cdot)$: $O(n)$ after sorting by start time \leftarrow how?
- $\text{M-Compute-Opt}(j)$: $O(n)$
 - Each entry $M[j]$ is computed only once
 - The computation of $M[j]$ invokes M-Compute-Opt twice

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if ( $v_j + M[p(j)] > M[j-1]$ )
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

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

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p(1), p(2), \dots, p(n)$

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

Top-down vs. bottom-up

- Top-down: May skip unnecessary sub-problems
- Bottom-up: Save the overhead in recursion

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

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



10 oz., \$1,000



Max Weight: 400 oz.



100 oz., \$2,000



300 oz., \$4,000



1 oz., \$5,000



200 oz., \$5,000

Knapsack Problem

Knapsack problem.

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

Ex: { 3, 4 } has value 40.

$W = 11$

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

Greedy:

- repeatedly add item with maximum value v_i .
- repeatedly add item with maximum weight w_i .
- repeatedly add item with maximum ratio v_i / w_i .

Greedy not optimal!

Dynamic Programming: False Start

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

- Case 1: OPT does not select item i .
 - OPT selects best of $\{1, 2, \dots, i-1\}$
- Case 2: OPT selects item i .
 - How shall we enforce the weight limit?

Conclusion. Shall specify the remaining weight capacity in OPT

Dynamic Programming: Adding a New Variable

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

- Case 1: OPT does not select item i.
 - OPT selects best of $\{ 1, 2, \dots, i-1 \}$ using weight limit w
- Case 2: OPT selects item i.
 - new weight limit = $w - w_i$
 - OPT selects best of $\{ 1, 2, \dots, i-1 \}$ using this new weight limit

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

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n -by- W array.

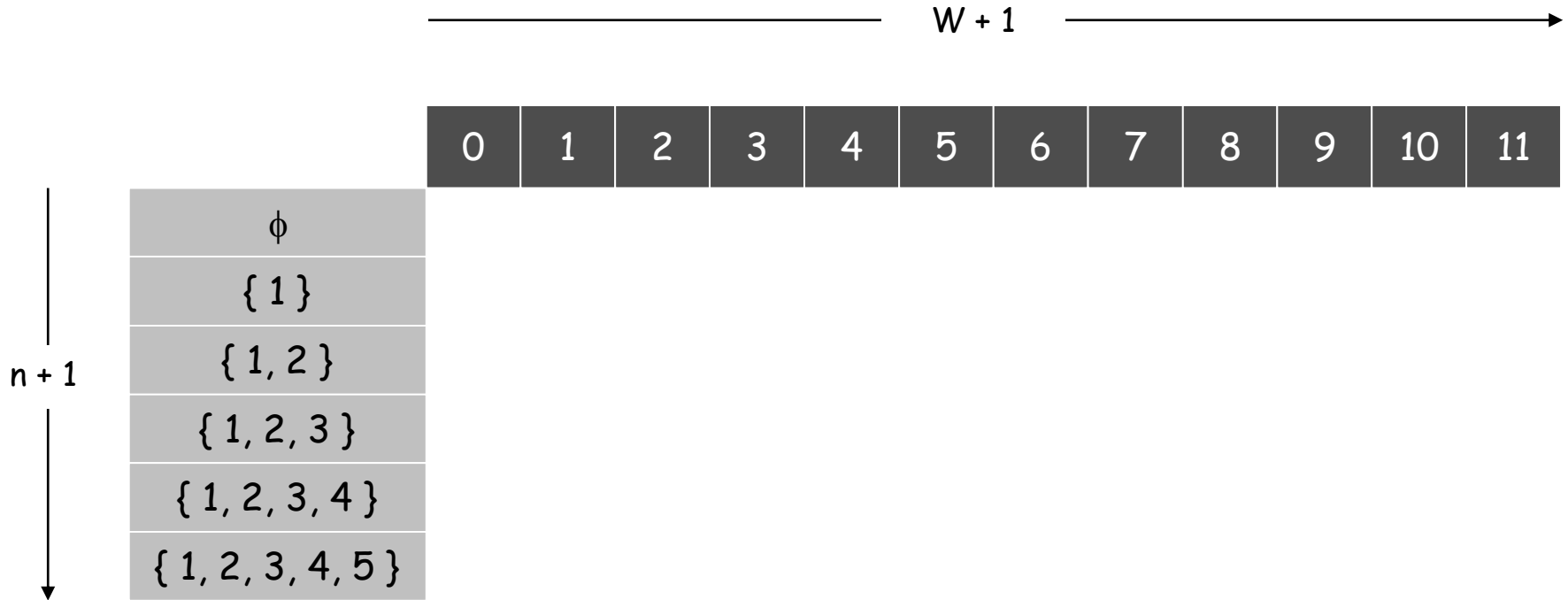
```
Input:  $n, w_1, \dots, w_N, v_1, \dots, v_N$ 

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

for  $i = 1$  to  $n$ 
    for  $w = 1$  to  $W$ 
        if  $(w_i > w)$ 
             $M[i, w] = M[i-1, w]$ 
        else
             $M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}$ 

return  $M[n, W]$ 
```

Knapsack Algorithm



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

$W = 11$

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

Knapsack Algorithm

		W + 1											
		0	1	2	3	4	5	6	7	8	9	10	11
n + 1	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }												
	{ 1, 2 }												
	{ 1, 2, 3 }												
	{ 1, 2, 3, 4 }												
	{ 1, 2, 3, 4, 5 }												

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

W = 11

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

Knapsack Algorithm

		W + 1 →											
		0	1	2	3	4	5	6	7	8	9	10	11
n + 1 ↓	ϕ	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

Case 2 (from {1,2,3} to {1,2,3,4})

Case 2 (from {1,2,3,4} to {1,2,3,4,5})

Case 1 (from {1,2,3,4,5} to {1,2,3,4,5})

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

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

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Knapsack Algorithm: Top-down

		W + 1 →											
		0	1	2	3	4	5	6	7	8	9	10	11
<div style="display: flex; align-items: center;"> <div style="border-left: 1px solid black; height: 100px; margin-right: 5px;"></div> <div style="display: flex; flex-direction: column; align-items: center;"> <div>n + 1</div> <div style="border-bottom: 1px solid black; width: 10px; height: 10px;"></div> <div>↓</div> </div> </div>	∅		0	0	0	0	0	0		0	0	0	0
	{ 1 }			1	1	1	1	1			1		1
	{ 1, 2 }	0				7	7	7					7
	{ 1, 2, 3 }					7	18						25
	{ 1, 2, 3, 4 }					7							40
	{ 1, 2, 3, 4, 5 }												40

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

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

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Knapsack Problem: Running Time

Running time. $\Theta(n W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

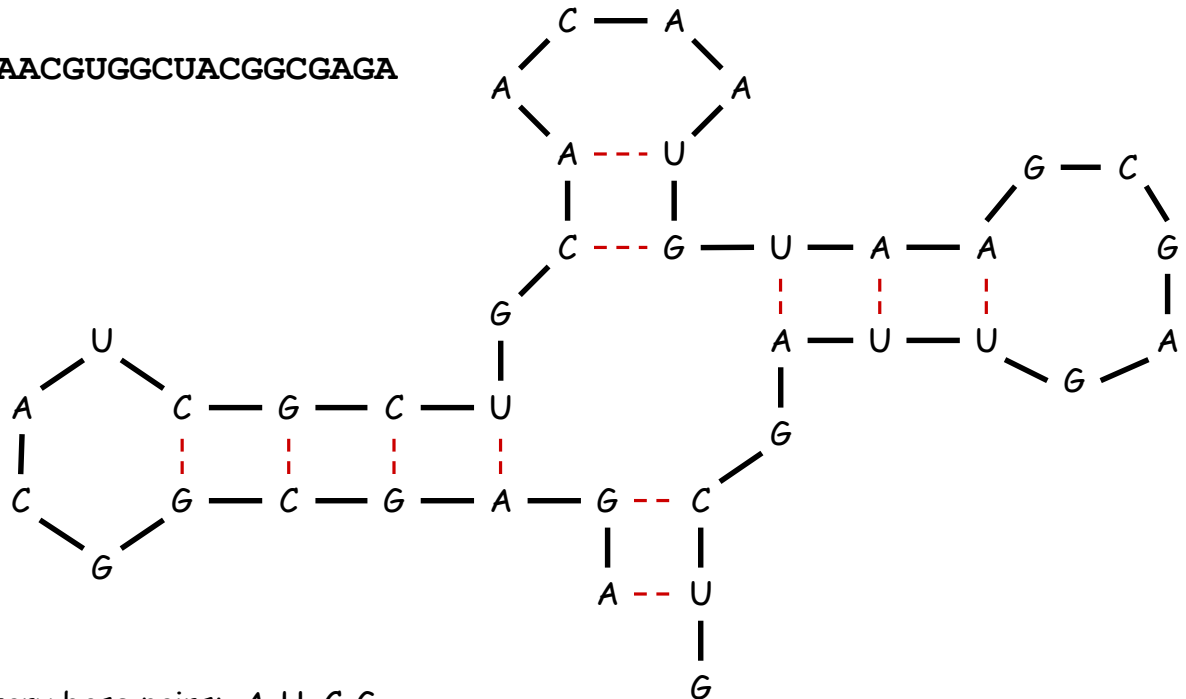
6.5 RNA Secondary Structure

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.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA



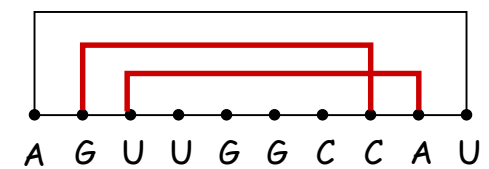
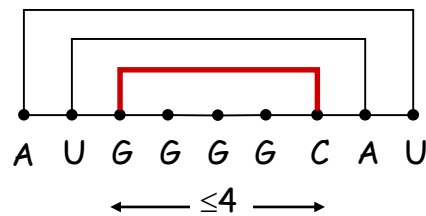
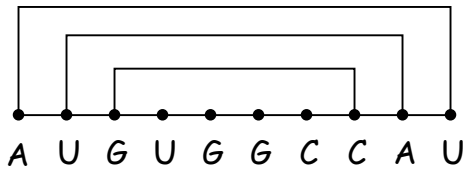
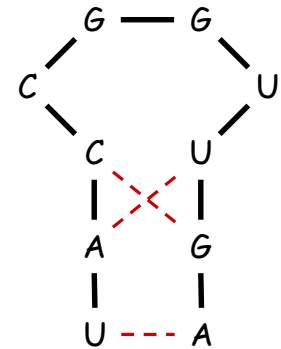
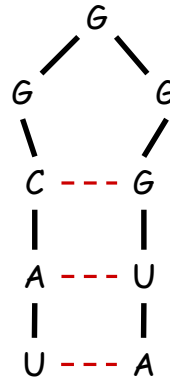
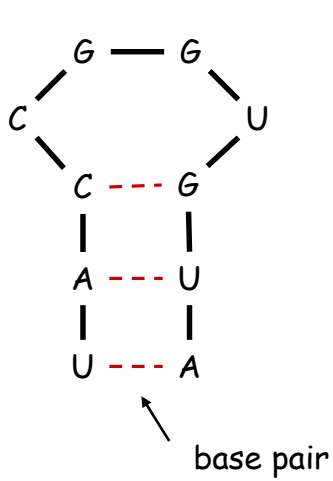
RNA Secondary Structure

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 $i < k < j < l$.

RNA Secondary Structure: Examples

Examples.



ok

sharp turn

crossing

RNA Secondary Structure

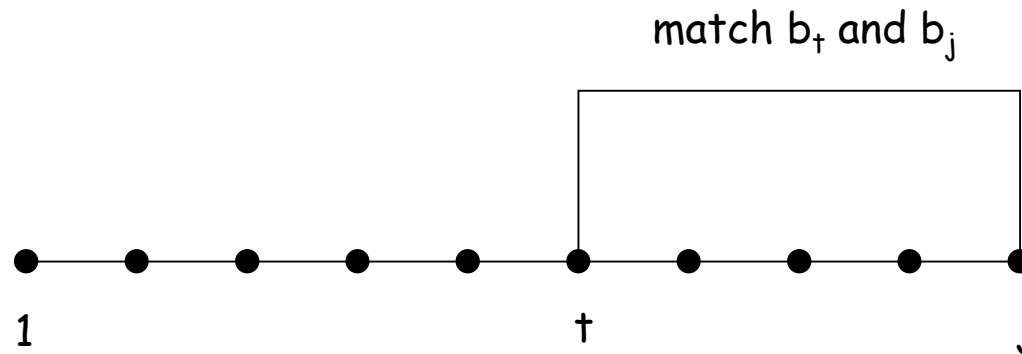
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

↑
approximate by number of base pairs

Goal. Given an RNA molecule $B = b_1b_2\dots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Subproblems

First attempt. $\text{OPT}(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1b_2\dots b_j$.



Difficulty. Results in two sub-problems.

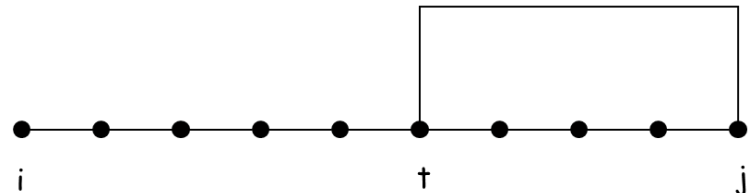
- Finding secondary structure in: $b_1b_2\dots b_{t-1}$. $\leftarrow \text{OPT}(t-1)$
- Finding secondary structure in: $b_{t+1}b_{t+2}\dots b_{j-1}$. \leftarrow need more sub-problems

Dynamic Programming Over Intervals

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

- If $i \geq j - 4$.
 - $\text{OPT}(i, j) = 0$ by no-sharp turns condition.
- If $i < j - 4$: take max of two cases
 - Case 1. Base b_j is not involved in a pair.
 $\text{OPT}(i, j-1)$
 - Case 2. Base b_j pairs with b_t for some $i \leq t < j - 4$.
Non-crossing constraint decouples resulting sub-problems
 $1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \}$

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

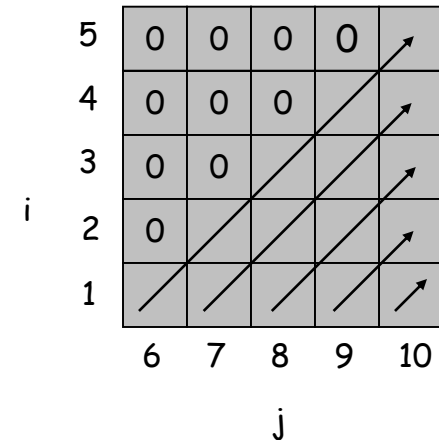


Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do shortest intervals first.

```
RNA( $b_1, \dots, b_n$ ) {  
    for  $k = 5, 6, \dots, n-1$   
        for  $i = 1, 2, \dots, n-k$   
             $j = i + k$   
            Compute  $M[i, j]$   
  
    return  $M[1, n]$   
}
```



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

6.6 Sequence Alignment

String Similarity

How similar are two strings?

- **ocurrance**
- **occurrence**

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

6 mismatches, 1 gap

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

1 mismatch, 1 gap

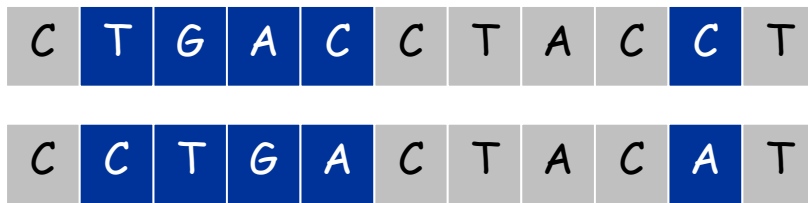
o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

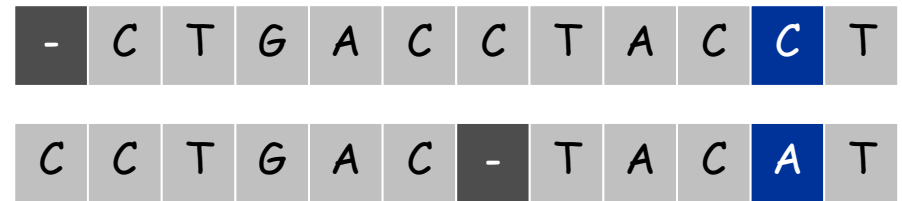
Edit Distance

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

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



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



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

Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ find alignment of minimum cost.

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.

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

Def. The **cost** of an alignment

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

Ex: An alignment of CTACCG vs. TACATG.

$$M = x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6.$$

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: Problem Structure

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

- Case 1: OPT matches x_i - y_j .
 - pay mismatch for x_i - y_j + min cost of aligning two strings $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
 - pay gap for x_i and min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$
- Case 2b: OPT leaves y_j unmatched.
 - pay gap for y_j and min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$

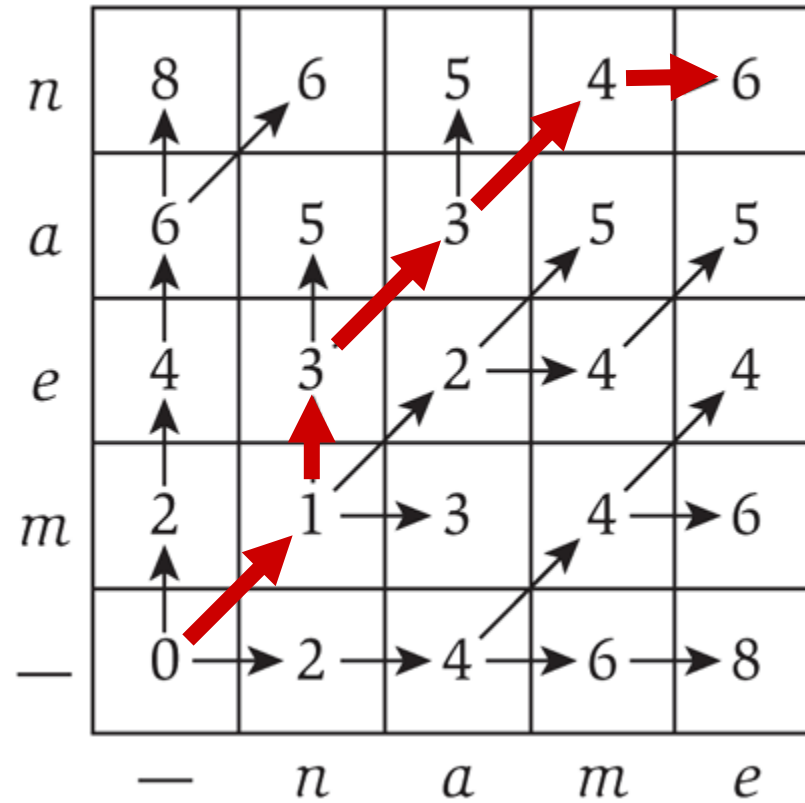
$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment( $m, n, x_1x_2\dots x_m, y_1y_2\dots y_n, \delta, \alpha$ ) {  
  for  $i = 0$  to  $m$   
     $M[0, i] = i\delta$   
  for  $j = 0$  to  $n$   
     $M[j, 0] = j\delta$   
  
  for  $i = 1$  to  $m$   
    for  $j = 1$  to  $n$   
       $M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1],$   
                     $\delta + M[i-1, j],$   
                     $\delta + M[i, j-1])$   
  
  return  $M[m, n]$   
}
```

Analysis. $\Theta(mn)$ time and space.

Sequence Alignment: Example



Sequence Alignment: Algorithm

Analysis. $\Theta(mn)$ time and space.

English words or sentences:

- $m, n \leq 30$. \leftarrow OK

Computational biology:

- $m = n = 100,000$
- 10 billions ops is OK, but 10GB array is quite large

6.7 Sequence Alignment in Linear Space

Sequence Alignment: Linear Space

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

Easy. Optimal **cost** in $O(m + n)$ space and $O(mn)$ time.

- Compute $\text{OPT}(i, \cdot)$ from $\text{OPT}(i-1, \cdot)$.
- No longer a simple way to recover alignment itself.

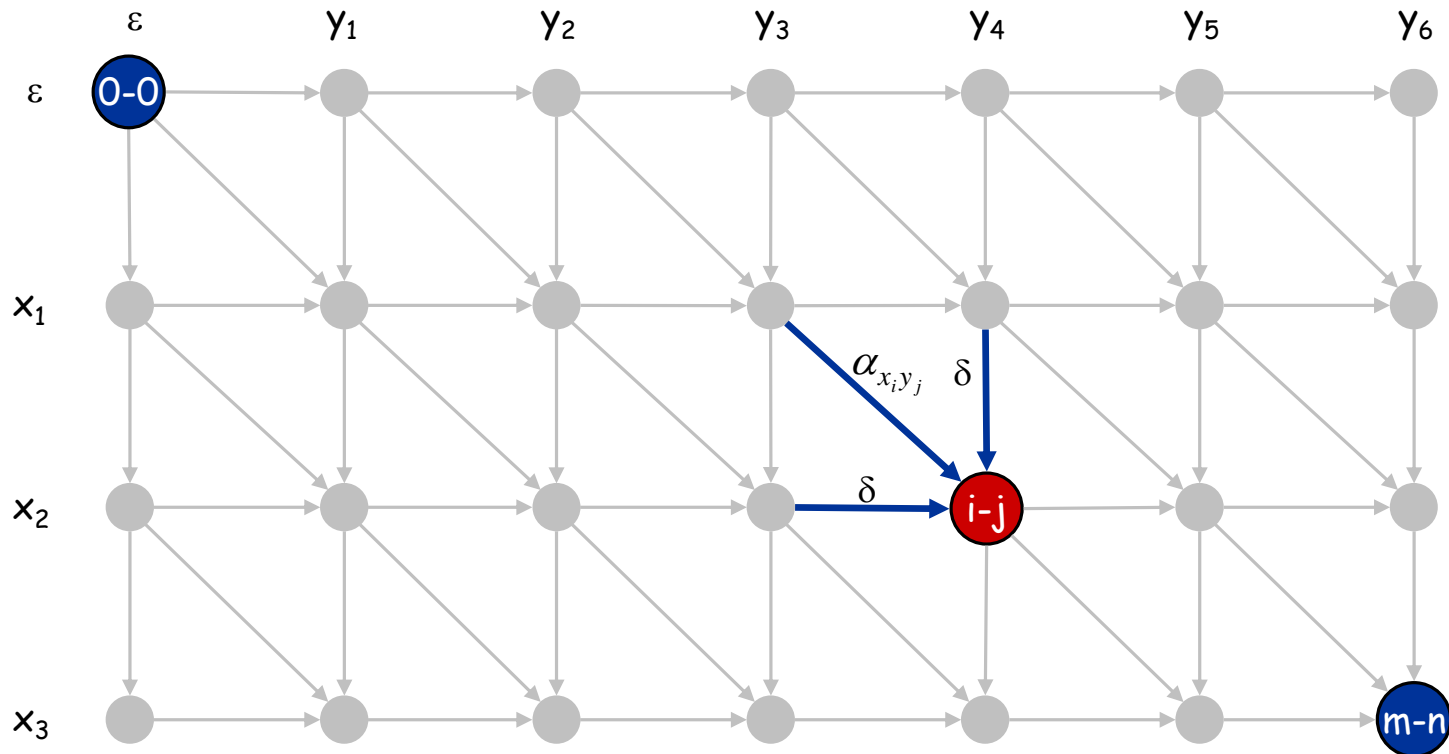
Theorem. [Hirschberg 1975] Optimal **alignment** in $O(m + n)$ space and $O(mn)$ time.

- Clever combination of divide-and-conquer and dynamic programming.

Sequence Alignment: Linear Space

Edit distance graph.

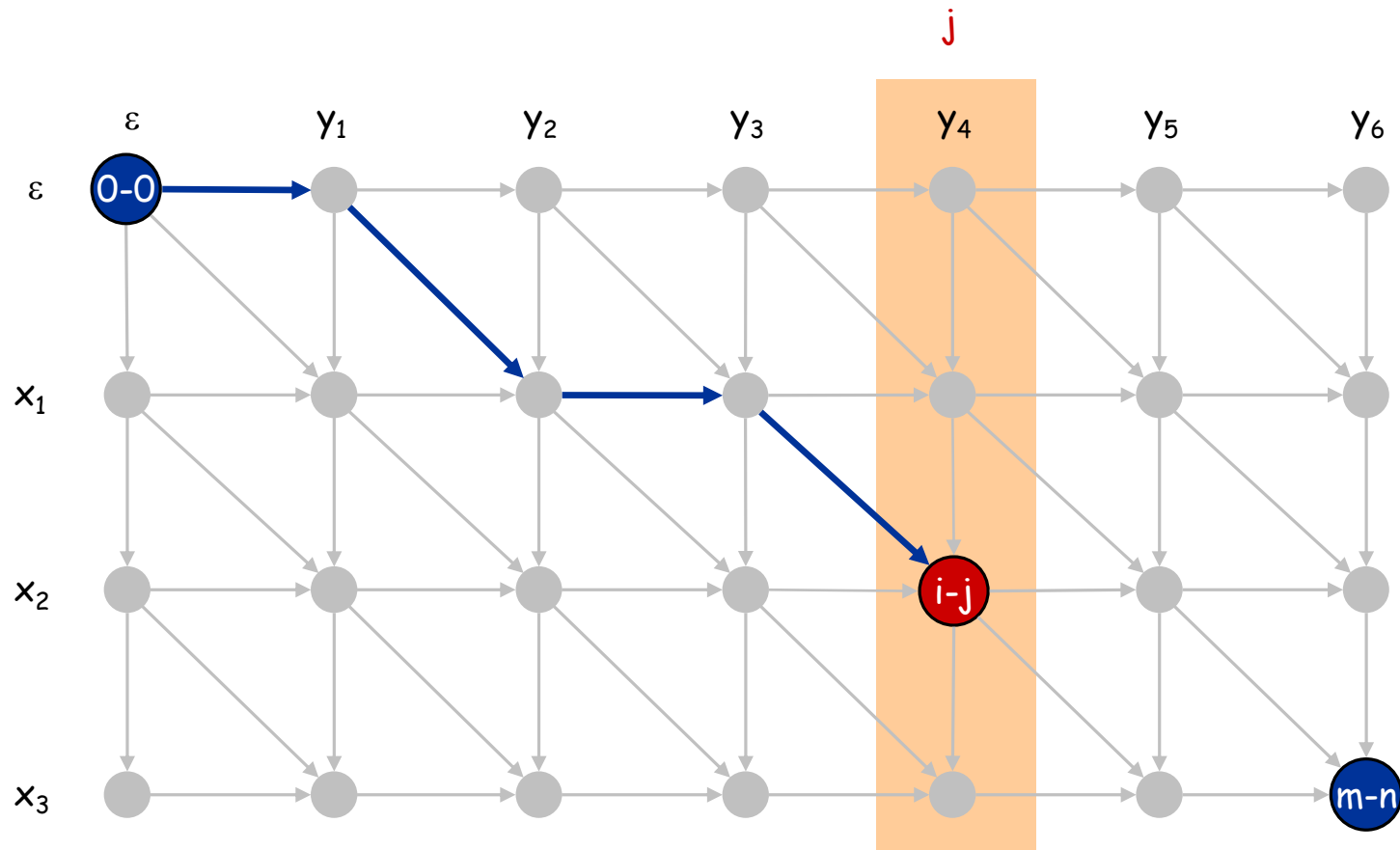
- Let $f(i, j)$ be shortest path from $(0,0)$ to (i, j) .
- Observation: $f(i, j) = \text{OPT}(i, j)$.



Sequence Alignment: Linear Space

Edit distance graph.

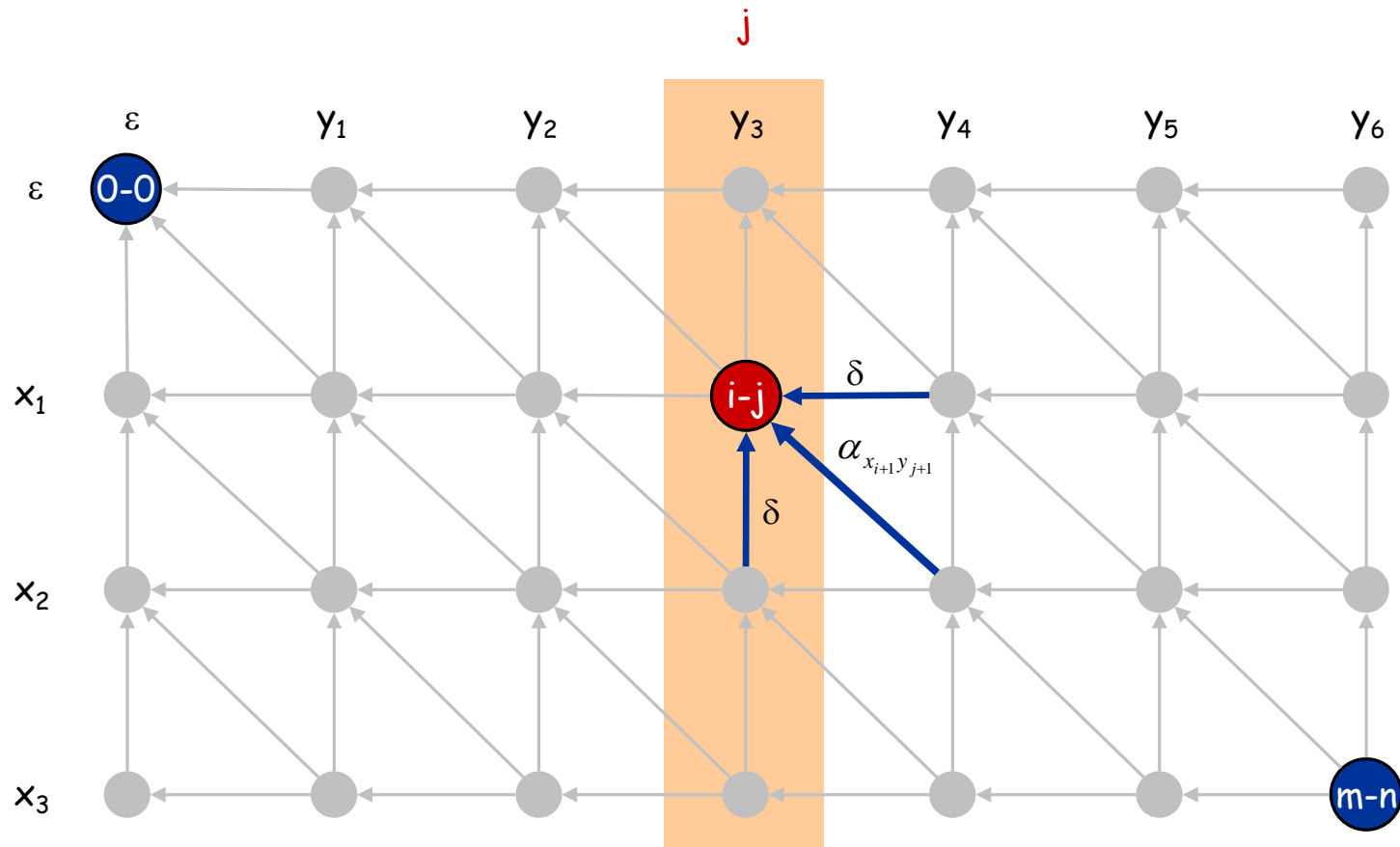
- Let $f(i, j)$ be shortest path from $(0,0)$ to (i, j) .
- Can compute $f(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



Sequence Alignment: Linear Space

Edit distance graph.

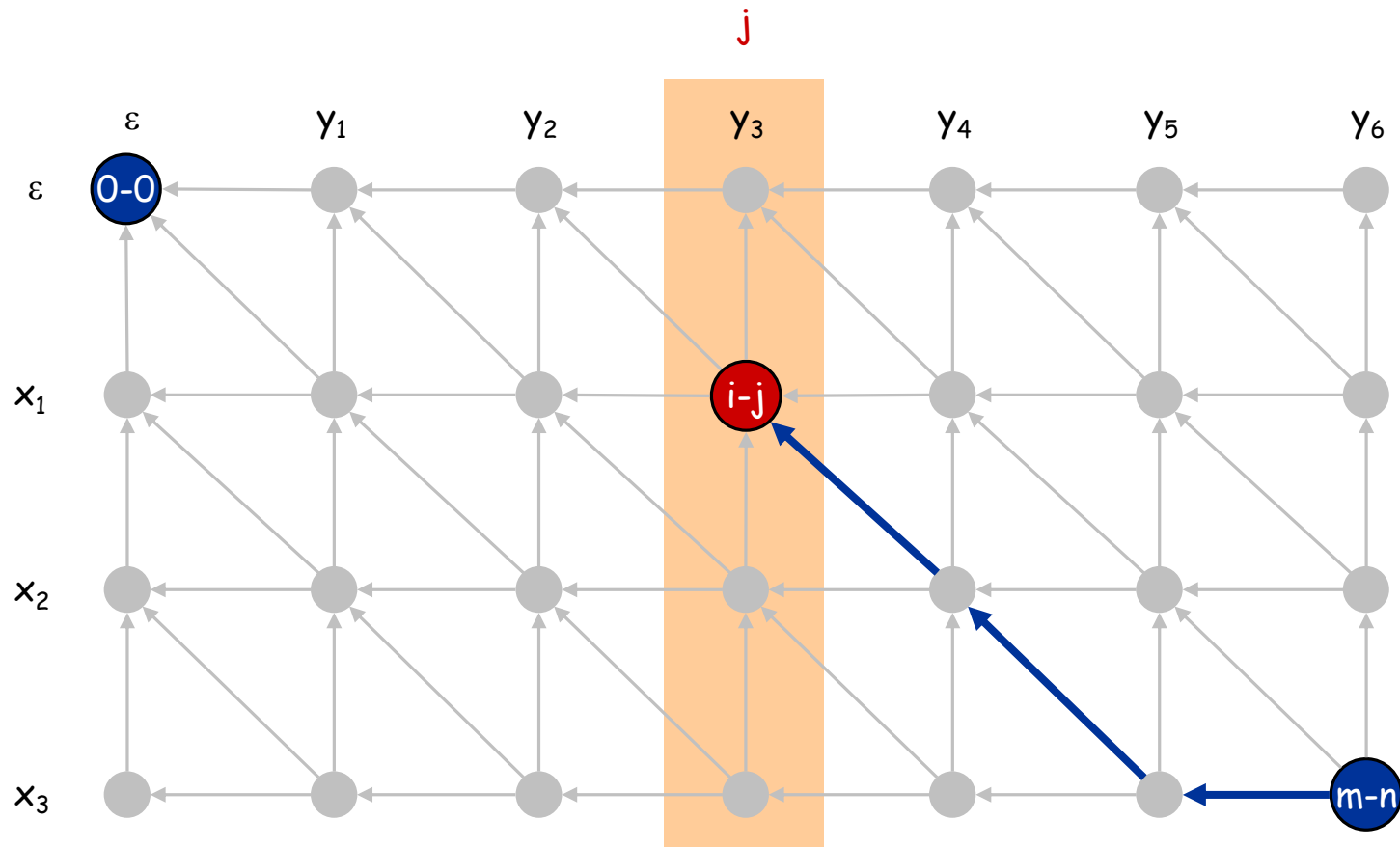
- Let $g(i, j)$ be shortest path from (i, j) to (m, n) .
- Can compute $g(\cdot, j)$ by reversing the edge orientations and inverting the roles of $(0, 0)$ and (m, n)



Sequence Alignment: Linear Space

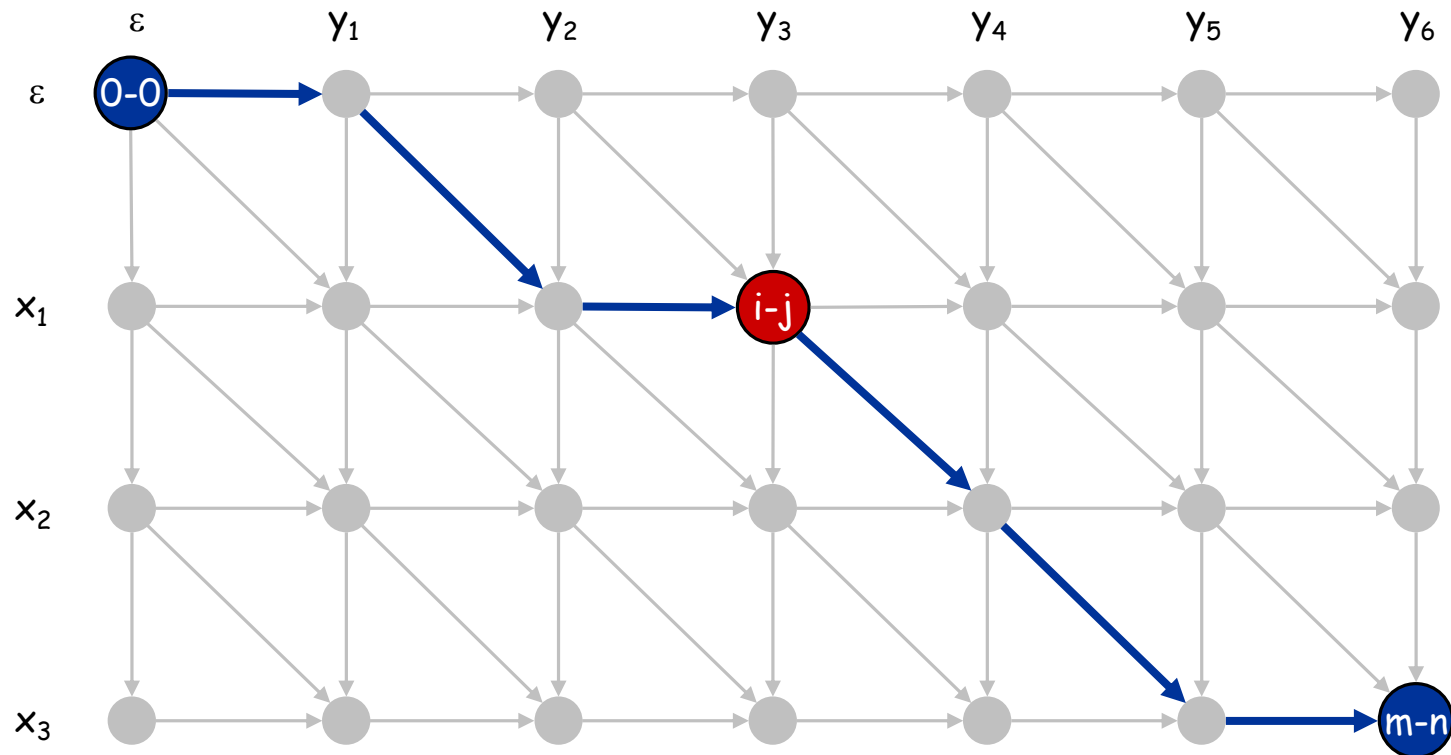
Edit distance graph.

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



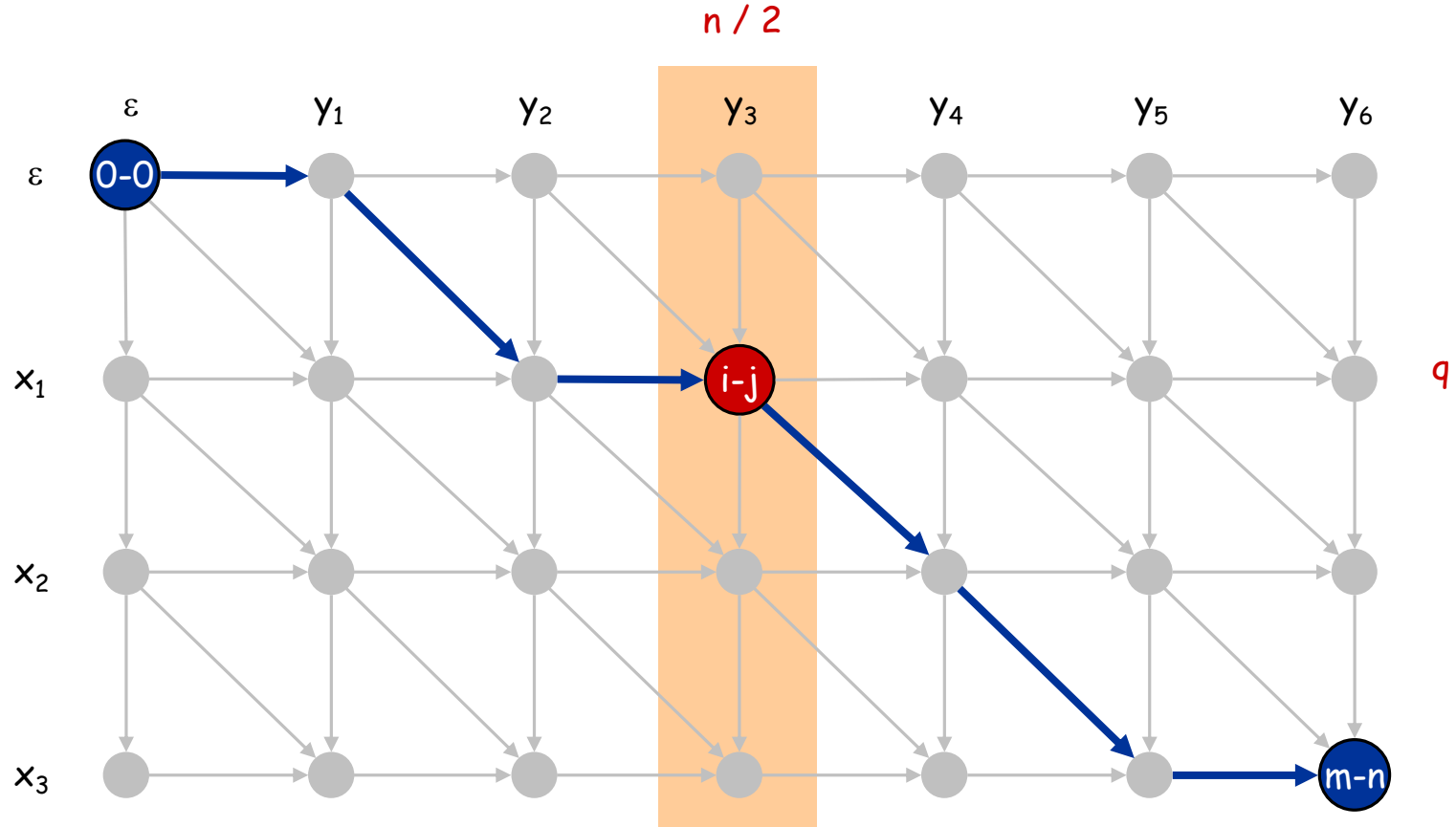
Sequence Alignment: Linear Space

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



Sequence Alignment: Linear Space

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

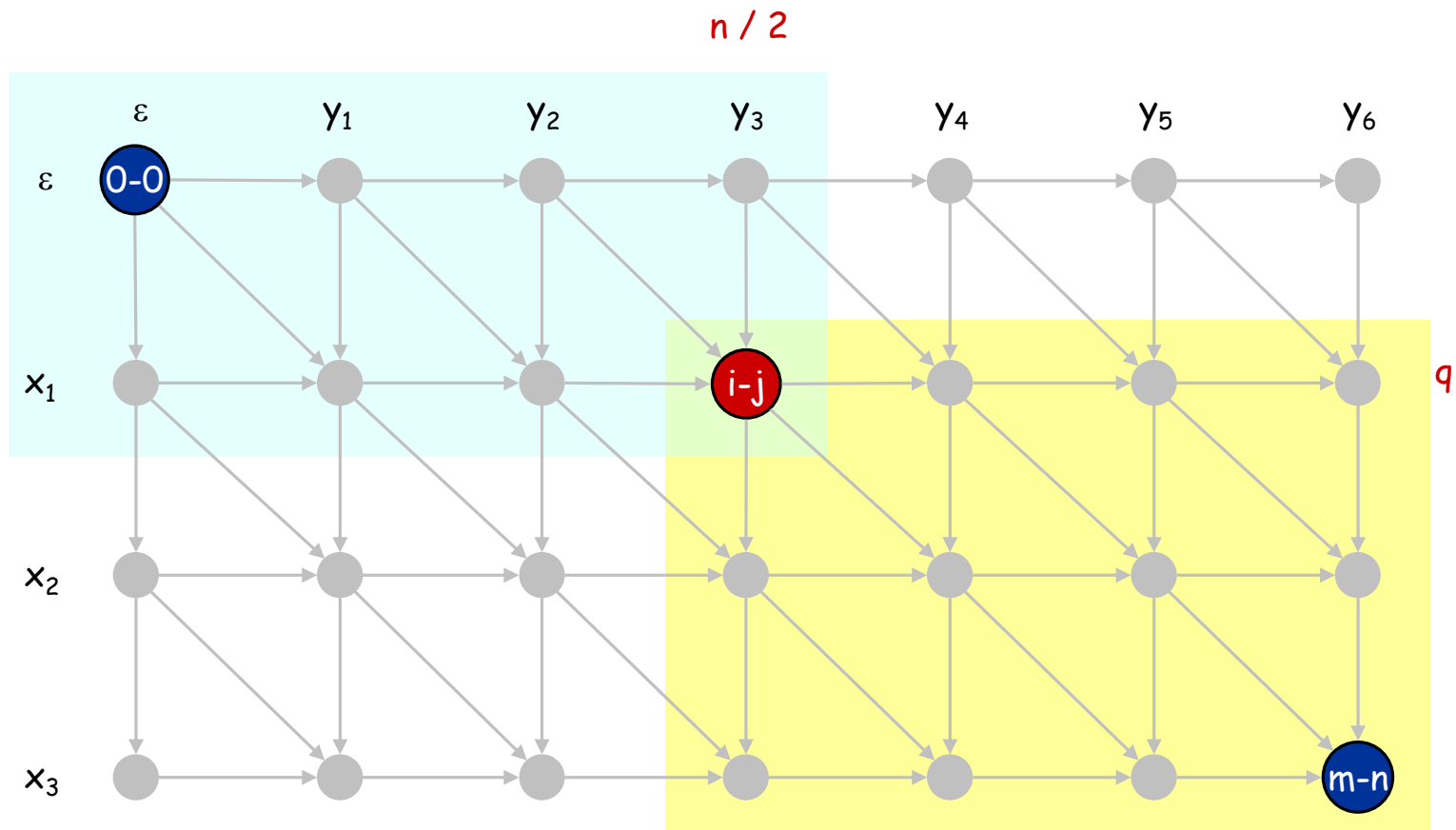


Sequence Alignment: Linear Space

Divide: find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Do alignment at $(x_q, y_{n/2})$.

Conquer: recursively compute optimal alignment in each piece.



Sequence Alignment: Running Time Analysis

Theorem. Let $T(m, n)$ = max running time of algorithm on strings of length m and n . $T(m, n) = O(mn)$.

Pf. (by induction on n)

- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index q .
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant c so that:

$$T(m, 2) \leq cm$$

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

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

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

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

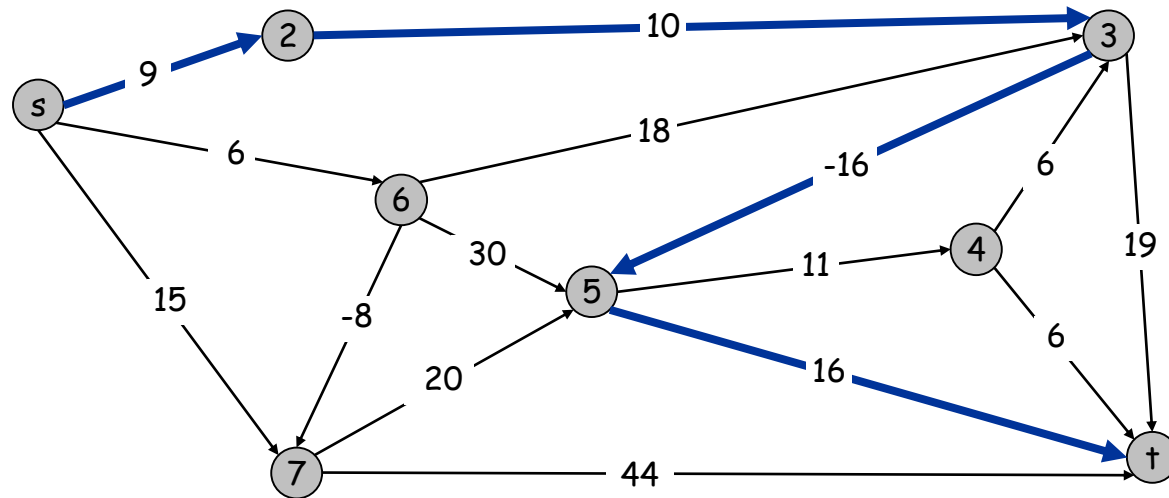
6.8 Shortest Paths

Shortest Paths

Shortest path problem. Given a directed graph $G = (V, E)$, with edge weights c_{vw} , find shortest path from node s to node t .

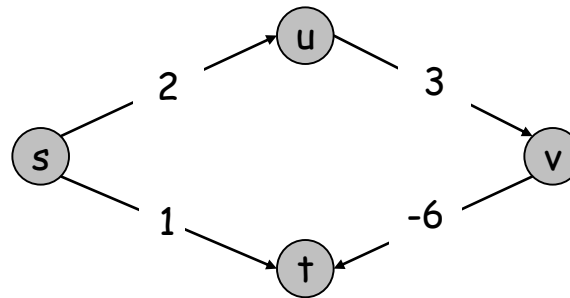
↖ allow negative weights

Ex. Nodes represent agents in a financial setting and c_{vw} is cost of transaction in which we buy from agent v and sell immediately to w .

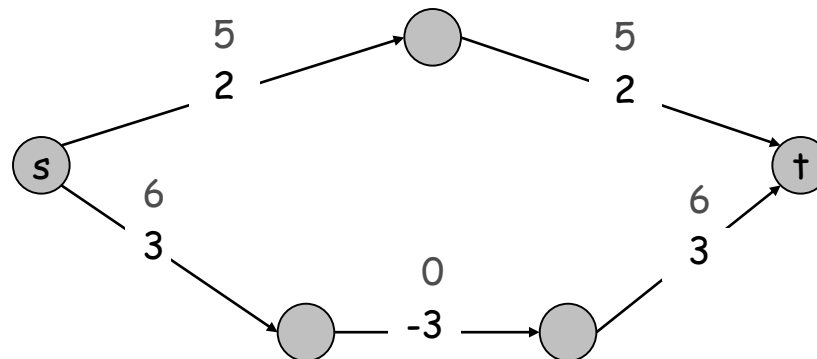


Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs.

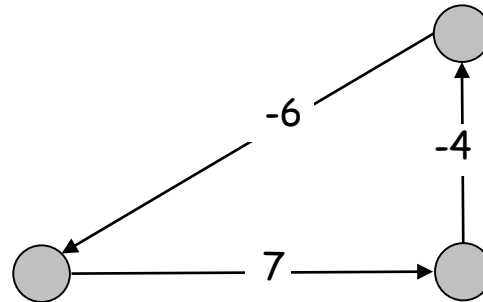


Re-weighting. Adding a constant to every edge weight can fail.

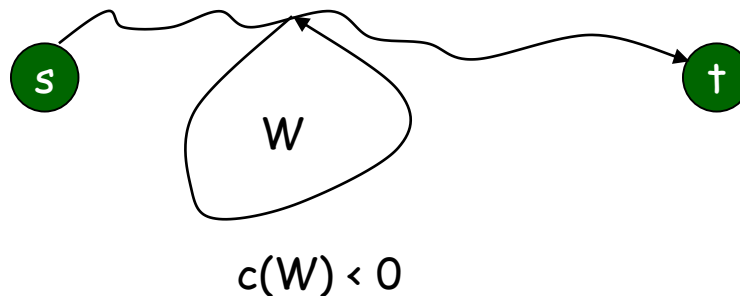


Shortest Paths: Negative Cost Cycles

Negative cost cycle.



Observation. If some path from s to t contains a negative cost cycle, there does not exist a shortest s - t path; otherwise, there exists one that is simple.



Shortest Paths: Dynamic Programming

Def. $OPT(i, v)$ = length of shortest v - t path P using at most i edges.

- Case 1: P uses at most $i-1$ edges.
 - $OPT(i, v) = OPT(i-1, v)$
- Case 2: P may use i edges.
 - if (v, w) is first edge, then OPT uses (v, w) , and then selects best w - t path using at most $i-1$ edges

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

Remark. By previous observation, if no negative cycles, then $OPT(n-1, v)$ = length of shortest v - t path.

Shortest Paths: Implementation

```
Shortest-Path( $G, s, t$ ) {  
    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  $(v, w) \in E$   
                 $M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + c_{vw} \}$   
  
    return  $M[n-1, s]$   
}
```

Analysis. $\Theta(mn)$ time, $\Theta(n^2)$ space.

Finding the shortest paths. Maintain a "successor" for each table entry.

Shortest Paths: Improvements

```
Shortest-Path( $G, s, t$ ) {  
    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  $(v, w) \in E$   
                 $M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + c_{vw} \}$   
  
    return  $M[n-1, s]$   
}
```

Practical improvements.

- Maintain only one array $M[v]$ = shortest v - t path that we have found so far.

Shortest Paths: Improvements

```
Shortest-Path( $G, s, t$ ) {  
    foreach node  $v \in V$   
         $M[v] \leftarrow \infty$   
     $M[t] \leftarrow 0$   
  
    for  $i = 1$  to  $n-1$   
        foreach node  $v \in V$   
         $M[v] \leftarrow M[v]$   
        foreach edge  $(v, w) \in E$   
             $M[v] \leftarrow \min \{ M[v], M[w] + c_{vw} \}$   
  
    return  $M[s]$   
}
```

Practical improvements.

- Maintain only one array $M[v]$ = shortest v - t path that we have found so far.
- No need to check edges of the form (v, w) unless $M[w]$ changed in previous iteration.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path( $G, s, t$ ) {  
    foreach node  $v \in V$   
         $M[v] \leftarrow \infty$   
     $M[t] \leftarrow 0$   
  
    for  $i = 1$  to  $n-1$  {  
        foreach node  $w \in V$   
            if ( $M[w]$  has been updated in previous iteration)  
                foreach node  $v$  such that  $(v, w) \in E$   
                    if ( $M[v] > M[w] + c_{vw}$ )  
                         $M[v] \leftarrow M[w] + c_{vw}$   
                If no  $M[w]$  value changed in iteration  $i$ , stop.  
    }  
  
    return  $M[s]$   
}
```

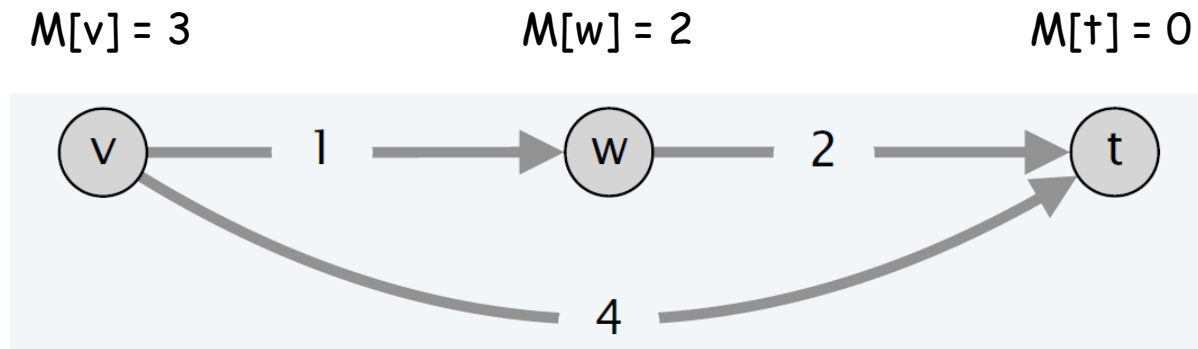
Analysis.

- $O(n)$ extra space
- Time: $O(mn)$ worst case, but substantially faster in practice.

Bellman-Ford: Efficient Implementation

Claim. Throughout the algorithm, $M[v]$ is length of some v - t path, and after i rounds of updates, the value $M[v]$ is the length of shortest v - t path using $\leq i$ edges.

Counter-example:



if nodes w considered before node v ,
then $M[v] = 3$ after 1 pass

Bellman-Ford: Efficient Implementation

Claim. Throughout the algorithm, $M[v]$ is length of some v - t path, and after i rounds of updates, the value $M[v]$ is **no larger than** the length of shortest v - t path using $\leq i$ edges.

6.9 Distance Vector Protocol

Distance Vector Protocol

Communication network.

- Nodes \approx routers.
- Edges \approx direct communication link.
- Cost of edge \approx delay on link. \leftarrow naturally nonnegative

Dijkstra's algorithm. Requires global information of network.

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

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

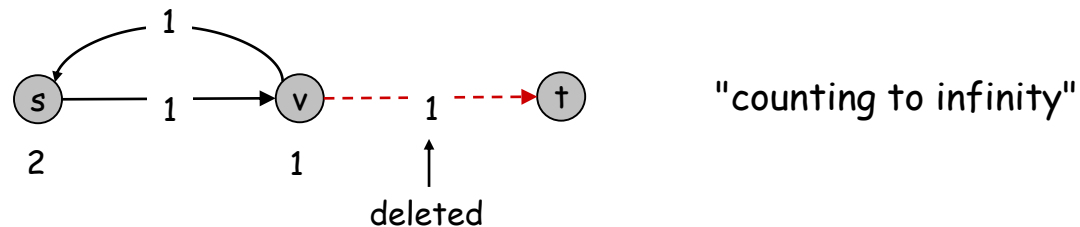
Distance Vector Protocol

Distance vector protocol.

- 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.
- "Routing by rumor."

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).



Path Vector Protocols

Link state routing.

- Each router also stores the entire path.
- Avoids "counting-to-infinity" problem and related difficulties.
- Requires significantly more storage.

not just the distance and first hop

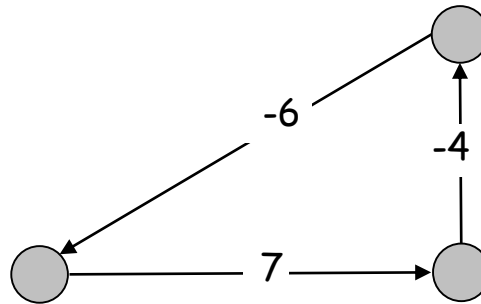


Ex. Border Gateway Protocol (BGP), Open Shortest Path First (OSPF).

6.10 Negative Cycles in a Graph

Detecting Negative Cycles

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

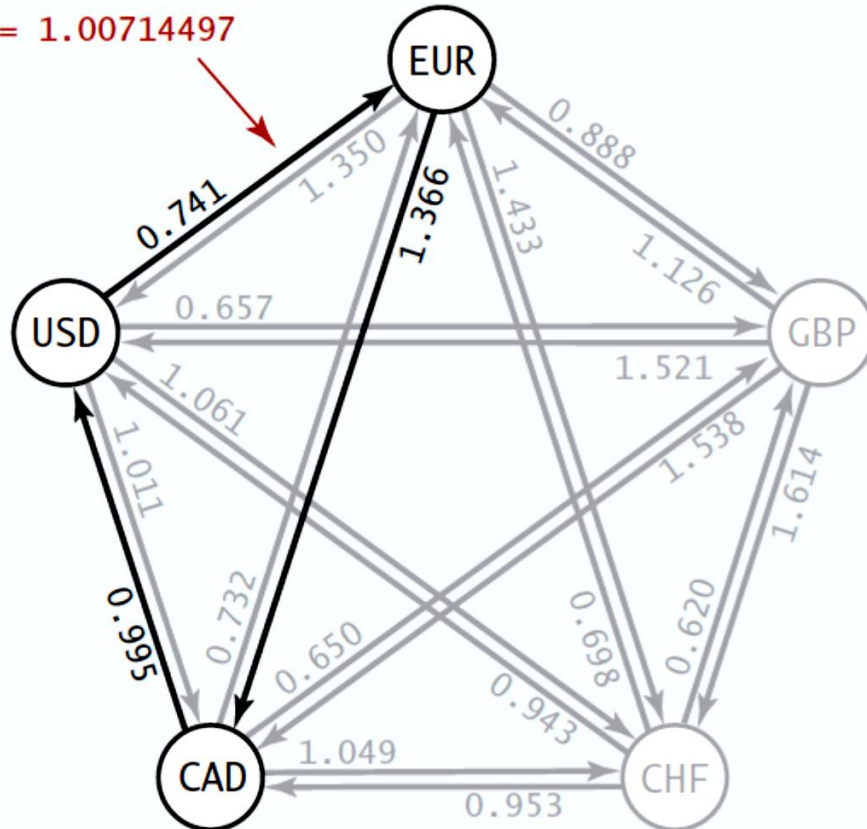


Detecting Negative Cycles: Application

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

Remark. Fastest algorithm very valuable!

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

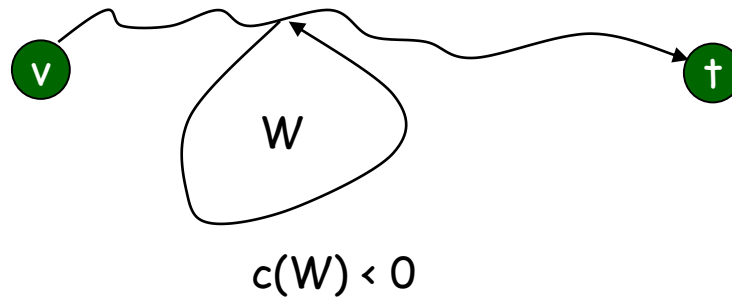


Detecting Negative Cycles

Lemma. If $\text{OPT}(n,v) = \text{OPT}(n-1,v)$ for all v , then there is no negative cycle with a path to t .

Pf. (by contradiction)

- $\text{OPT}(n,v) = \text{OPT}(n-1,v) \Rightarrow \text{OPT}(i,v) = \text{OPT}(n-1,v)$ for $i \geq n$
- But negative cycle in a path implies that $\text{OPT}(i,v)$ always decreases as i increases

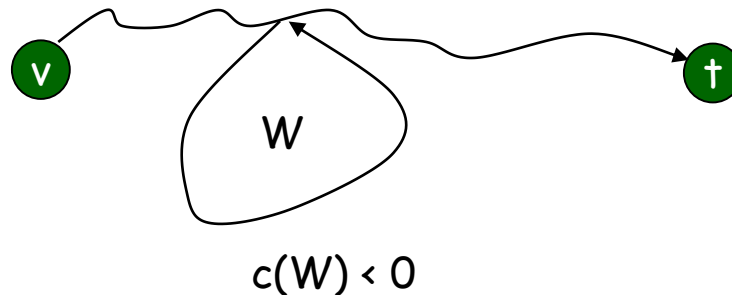


Detecting Negative Cycles

Lemma. If $\text{OPT}(n,v) < \text{OPT}(n-1,v)$ for some node v , then (any) shortest path from v to t contains a cycle W . Moreover W has negative cost.

Pf. (by contradiction)

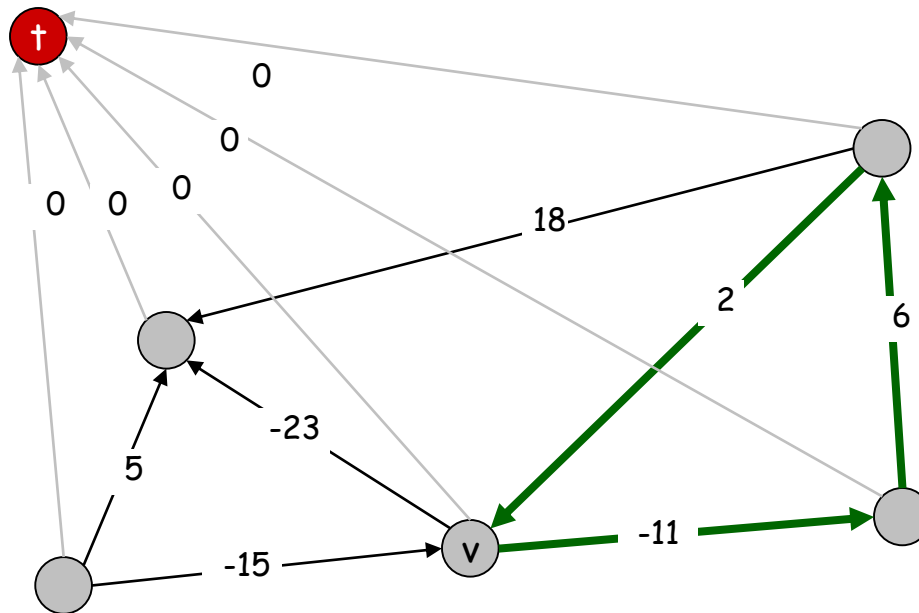
- Since $\text{OPT}(n,v) < \text{OPT}(n-1,v)$, we know the shortest v - t path P has exactly n edges.
- By pigeonhole principle, P must contain a directed cycle W .
- Deleting W yields a v - t path with $< n$ edges $\Rightarrow W$ has negative cost.



Detecting Negative Cycles

Theorem. Can detect negative cost cycle in $O(mn)$ time.

- Add new node t and connect all nodes to t with 0-cost edge.
- Check if $\text{OPT}(n, v) = \text{OPT}(n-1, v)$ for all nodes v .
 - if yes, then no negative cycles
 - if no, then extract cycle from shortest path from v to t



Dynamic Programming: Chapter Summary

Dynamic Programming

Basic idea

- Polynomial number of sub-problems with a natural ordering from smallest to largest.
- Optimal solution to a sub-problem can be constructed from optimal solutions of smaller sub-problems.
- Sub-problems are overlapping!

Guideline

- Define the sub-problems
 - $\text{OPT}(\dots)$
- Write down the recursive formulas
 - Ex: $\text{OPT}(i) = \max(f(\text{OPT}(j)), g(\text{OPT}(k)), \dots), j, k < i$
- Compute the formulas either bottom-up or top-down

Dynamic Programming

Algorithms

- Weighted interval scheduling
 - 1D array; binary choice
- Knapsack
 - 2D array; adding a new variable (weight limit)
- RNA secondary structure
 - 2D array: intervals
- Sequence Alignment
 - 2D array: prefix alignment
- Sequence Alignment in Linear Space
 - Combination of divide-and-conquer and dynamic programming
- Shortest path with negative edges
 - (Bellman-Ford) 2D array: shortest path with edge number $\leq i$
- Distance Vector Protocol
- Negative Cycle Detection