

Dynamic Programming*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course: Shanghai Jiao Tong University

*Special thanks is given to *Prof. Kevin Wayne@Princeton* for sharing his slides, and also given to Mr. Chao Wang from CS2014@SJTU and Mr. Hongjian Cao from CS2015@SJTU for producing this lecture.

Outline

- 1 Basic Methodology
 - Introduction
 - Four Examples (Recipe of Dynamic Programming)
- 2 A Comprehensive Example
 - String Similarity
 - Sequence Alignment in Linear Space

Algorithmic Paradigms

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

Divide-and-conquer: Break up a problem into sub-problems, solve each sub-problem independently, 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.

Dynamic Programming Applications

Areas

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

Some famous dynamic programming algorithms

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

Four Examples

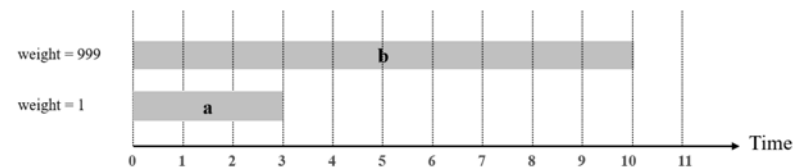
- Weighted Interval Scheduling
- Segmented Least Squares
- Knapsack Problem
- RNA Secondary Structure

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.

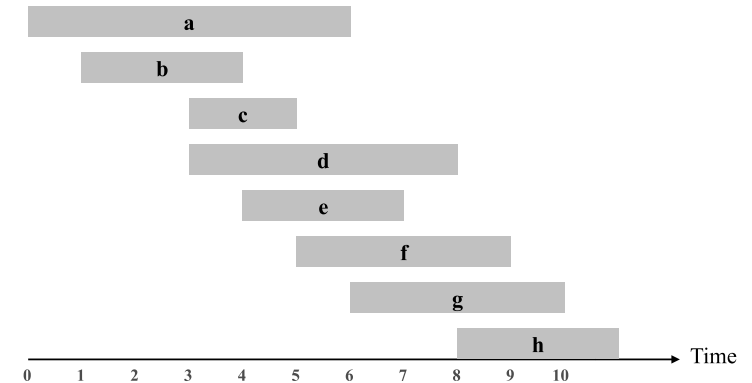


Weighted Interval Scheduling Problem

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.

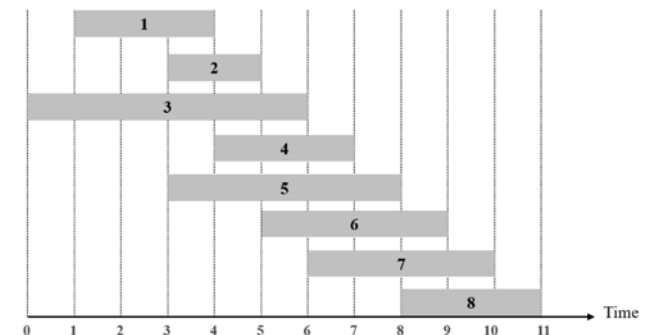


Weighted Interval Scheduling

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

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

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



Binary Choice

Greedy template: $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \dots, j$.

Optimal substructure:

Case 1: OPT selects job j .

- collect profit v_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 $1, 2, \dots, p(j)$

Case 2: OPT does not select job j .

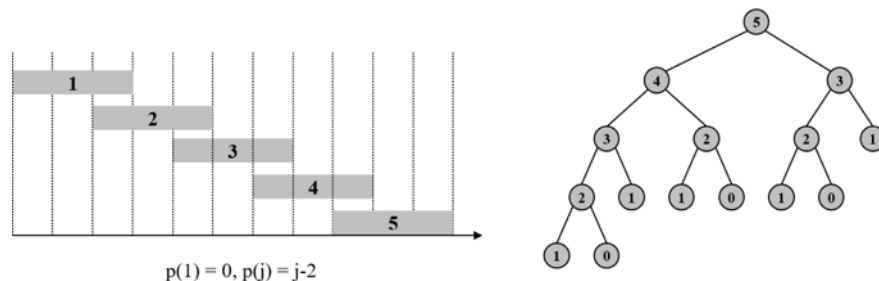
- must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, j - 1$

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

Brute Force Algorithm

Observation: Recursive algorithm fails spectacularly because of redundant sub-problems \Rightarrow [exponential algorithms](#).

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



Brute Force Algorithm

Algorithm 1: Brute Force

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

- 1 **Sort** jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$;
- 2 **Compute** $p(1), p(2), \dots, p(n)$;
- 3 **Function** Compute-Opt (j) :
 - 4 **if** $j = 0$ **then**
 - 5 **return** 0;
 - 6 **else**
 - 7 **return**
 - 8 $\max\{v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1)\};$

Memoization

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

Algorithm 2: Memoization

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

- 1 **Sort** jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$;
- 2 **Compute** $p(1), p(2), \dots, p(n)$;
- 3 **for** $j = 1 \rightarrow n$ **do**
- 4 $M[j] = \text{empty};$
- 5 $M[0] = 0;$
- 6 **Function** M-Compute-Opt (j) :
 - 7 **if** $M[j]$ *is empty* **then**
 - 8 $M[j] =$
 - 9 $\max\{v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j - 1)\};$
 - 10 **return** $M[j];$

Running Time

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

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.
- M-Compute-Opt(j): each invocation takes $O(1)$ time and either
 - ▷ returns an existing value $M[j]$
 - ▷ fills in one new entry $M[j]$ and makes two recursive calls
- Progress measure Φ = number nonempty entries of $M[\cdot]$.
 - ▷ initially $\Phi = 0$, throughout $\Phi \leq n$.
 - ▷ increases Φ by 1 \Rightarrow at most $2n$ recursive calls.
- Overall running time of M-Compute-Opt(n) is $O(n)$.

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

Bottom-Up Dynamic Programming

Algorithm 4: Tabulation

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

- 1 **Sort** jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$;
- 2 **Compute** $p(1), p(2), \dots, p(n)$;
- 3 **Function** Iterative-Compute-Opt():
 - 4 $M[0] = 0$;
 - 5 **for** $j = 1 \rightarrow n$ **do**
 - 6 $M[j] = \max\{v_j + M[p(j)], M[j-1]\}$;

Finding a Solution from the OPT Value

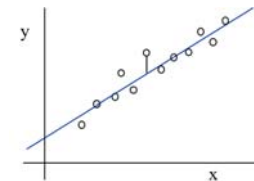
Algorithm 3: Post-Processing

- 1 **Run** M-Compute-Opt(n);
- 2 **Run** Find-Solution(n);
- 3 **Function** Find-Solution(j):
 - 4 **if** $j = 0$ **then**
 - 5 \mid output nothing;
 - 6 **else if** $v_j + M[p(j)] > M[j-1]$ **then**
 - 7 \mid print j ;
 - 8 \mid Find-Solution($p(j)$);
 - 9 **else**
 - 10 \mid Find-Solution($j-1$);

- # of recursive calls $1 \leq n \Rightarrow O(n)$;

Segmented Least Squares

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
- Find a line $y = ax + b$ to minimize the sum of the squared error:



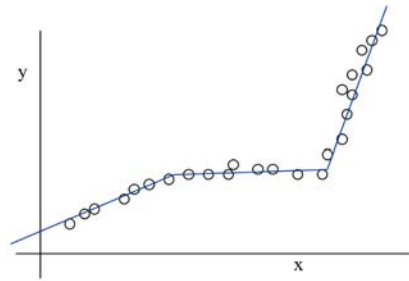
Solution: Calculus \Rightarrow min error is achieved when

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

Segmented Least Squares

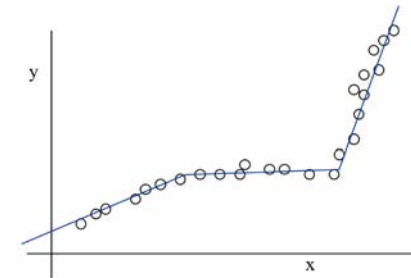
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ with $x_1 < x_2 < \dots < x_n$, find a sequence of lines that minimizes $f(x)$.

Question: What's a reasonable choice for $f(x)$ to balance accuracy (goodness of fit) and parsimony (number of lines)?



Segmented Least Squares

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



Multiway Choice

Notation:

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

Compute $OPT(j)$:

- Last segment uses points p_i, p_{i+1}, \dots, p_j for some i .
- $Cost = e(i, j) + c + OPT(i - 1)$.

$$OPT(j) = \begin{cases} 0, & j = 0, \\ \min_{1 \leq i \leq j} \{e(i, j) + c + OPT(i - 1)\}, & \text{otherwise} \end{cases}$$

Segmented Least Squares

Algorithm 5: Tabulation

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

```

1 Function Iterative-Compute-Opt () :
2    $M[0] = 0;$ 
3   for  $j = 1 \rightarrow n$  do
4     for  $i = 1 \rightarrow j$  do
5       compute the least square error  $e_{ij}$  for the segment
          $p_i, \dots, p_j;$ 
6   for  $j = 1 \rightarrow n$  do
7      $M[j] = \min_{1 \leq i \leq j} \{e_{ij} + c + M[i - 1]\};$ 
8   return  $M[n];$ 
```

Running time: $O(n^3)$ (can be improved to $O(n^2)$ by pre-computing.)

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

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.

Example: {3, 4} has value 40.

W = 11

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

Greedy: repeatedly add item with maximum ratio v_i/w_i .

Example: {5, 2, 1} achieves only value = 35 \Rightarrow greedy not optimal.

Adding a New Variable

Definition: $OPT(i, w) = \max$ profit subset of items $1, \dots, i$ **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 using $\{1, 2, \dots, i-1\}$ this new weight limit

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

False Start

Definition: $OPT(i) = \max$ 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 .

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

Conclusion: Need more sub-problems!

Bottom-Up Algorithm (Fill up an n -by- W array)

Algorithm 6: Knapsack Problem Algorithm using n -by- W Array

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

```

1 for  $w = 0 \rightarrow W$  do
2    $M[0, w] = 0$ ;
3 for  $i = 1 \rightarrow n$  do
4   for  $w = 1 \rightarrow W$  do
5     if  $w_i > w$  then
6        $M[i, w] = M[i-1, w]$ ;
7     else
8        $M[i, w] = \max\{M[i-1, w], v_i + M[i-1, w - w_i]\}$ 
9 return  $M[n, W]$ ;
```

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	35	40

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

W = 11

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

Running Time

Running time: $\Theta(nW)$.

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

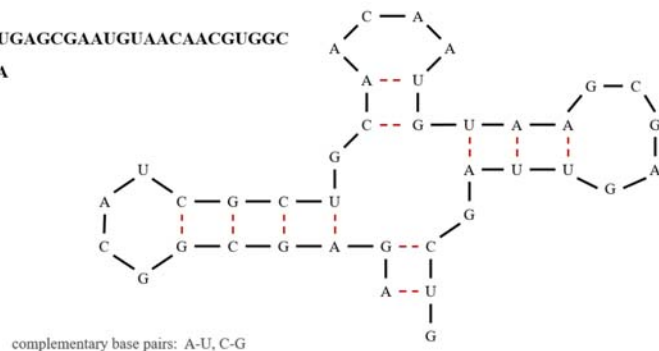
Knapsack approximation algorithm: There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum.

RNA Secondary Structure

RNA: String $B = b_1b_2 \cdots b_n$ 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.

Example: GUCGAUUGAGCGAAUGUACAACGUGGC
UACGCGCAGA



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.

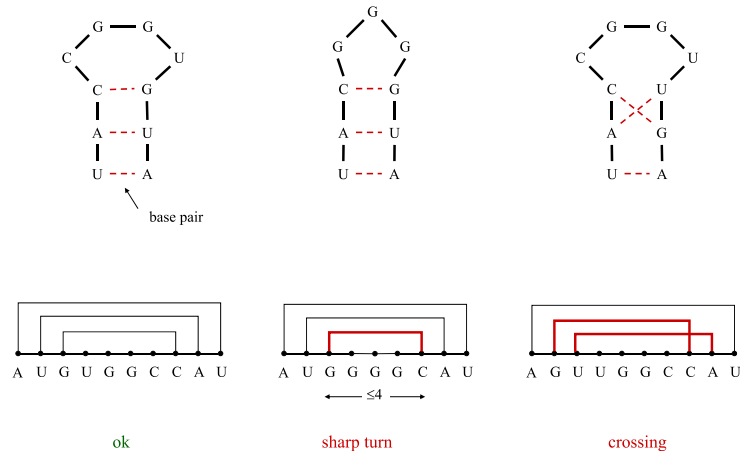
[No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.

[Non-crossing] If (b_i, b_j) and (b_k, b_l) are two pairs in S , then we cannot have $i < k < j < l$.

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

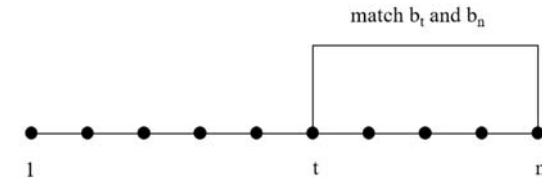
Goal: Given an RNA molecule $B = b_1b_2 \cdots b_n$, find a secondary structure S that maximizes the number of base pairs

Examples



Subproblems

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



Difficulty: Results in two sub-problems.

- Finding secondary structure in: $b_1 b_2 \cdots b_{t-1}$.
- Finding secondary structure in: $b_{t+1} b_{t+2} \cdots b_{n-1}$.

Dynamic Programming Over Intervals

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

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

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

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

- $OPT(i, j) = OPT(i, j - 1)$

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

- non-crossing constraint decouples resulting sub-problems
- $OPT(i, j) = 1 + \max_t \{OPT(i, t - 1) + OPT(t + 1, j - 1)\}$

Remark: Same core idea in CKY algorithm to parse context-free grammars.

Bottom Up Dynamic Programming Over Intervals

Question: What order to solve the sub-problems?

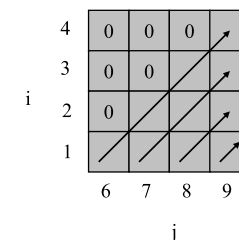
Answer: Do shortest intervals first.

Algorithm 7: RNA

```

1 Function RNA ( $b_1, b_2, \dots, b_n$ ) :
2   for  $k = 5 \rightarrow n - 1$  do
3     for  $i = 1 \rightarrow n - k$  do
4        $j = i + k$ ;
5       Compute  $M[i, j]$ 
6   return  $M[1, n]$ ;

```



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

Dynamic Programming Summary

Recipe

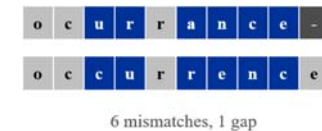
- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over interval

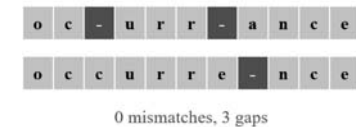
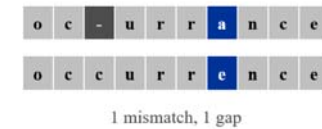
Top-down vs. bottom-up: different people have different intuitions.

String Similarity: How similar are two strings?



How similar are two strings?

- occurrence
- occurrence



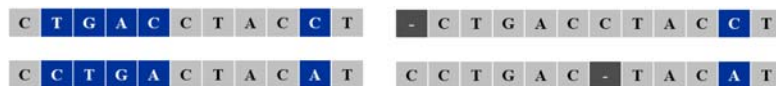
Edit Distance

Applications.

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

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

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



Sequence Alignment

Goal: Given two strings $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$ find alignment of minimum cost.

Definition: 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.

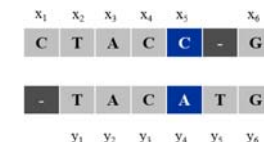
Definition: The pair x_i-y_j and $x_{i'}-y_{j'}$ **cross** if $i < i'$, but $j > j'$.

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

mismatch gap

Example: CTACCG vs. TACATG.

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



Problem Structure

Definition: $OPT(i, j) = \min$ cost of aligning strings $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$.

Case 1: OPT matches x_i - y_j .

pay mismatch for x_i - y_j + min cost of aligning two strings $x_1x_2 \cdots x_{i-1}$ and $y_1y_2 \cdots y_{j-1}$

Case 2a: OPT leaves x_i unmatched.

pay gap for x_i and min cost of aligning $x_1x_2 \cdots x_{i-1}$ and $y_1y_2 \cdots y_j$

Case 2b: OPT leaves y_j unmatched.

pay gap for y_j and min cost of aligning $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_{j-1}$

Sequence Alignment

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

English words or sentences: $m, n \leq 10$.

Computational biology: $m = n = 100,000$. 10 billions OK, but 10GB array?

Sequence Alignment

Algorithm 8: Sequence Alignment

1 Function

Sequence-Alignment ($m, n, x_1x_2 \cdots x_m, y_1y_2 \cdots y_n, \delta, \alpha$) :

```

2   for  $i = 0 \rightarrow m$  do
3        $M[i, 0] = i\delta$ 
4   for  $j = 0 \rightarrow n$  do
5        $M[0, j] = j\delta$ 
6   for  $i = 1 \rightarrow m$  do
7       for  $j = 1 \rightarrow n$  do
8            $M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1],$ 
9                        $\delta + M[i-1, j],$ 
10                       $\delta + M[i, j-1])$ 
11  return  $M[m, n]$ ;
```

Linear Space

Question: Can we avoid using quadratic **space**?

Easy. Optimal **value** in $O(m)$ space and $O(mn)$ time.

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

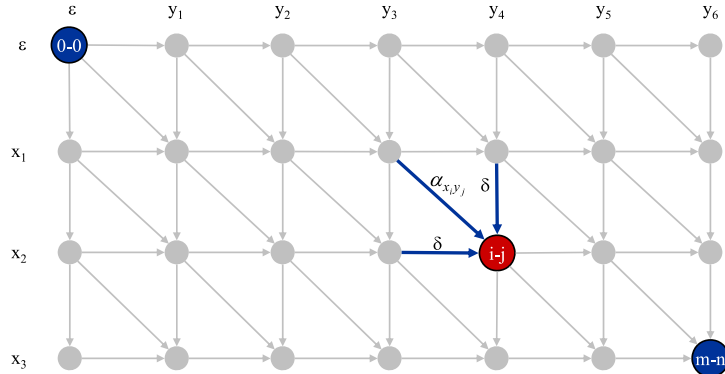
Theorem. [Hirschberg 1975] Optimal **alignment** in $O(m+n)$ space and $O(mn)$ time.

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



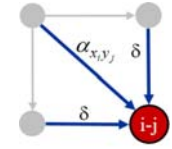
Edit Distance Graph

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



Edit Distance Graph

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



Proof: (by strong induction on $i + j$)

Base case: $f(0, 0) = OPT(0, 0) = 0$

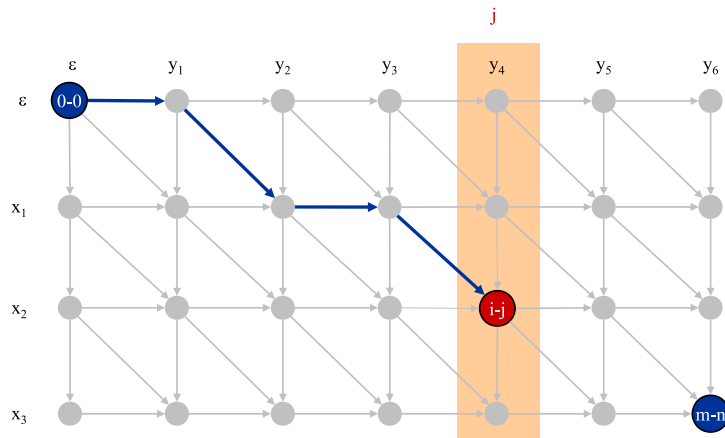
Inductive hypothesis: assume true for all (i', j') with $i' + j' < i + j$.

Induction: Last edge on shortest path to (i, j) is from $(i - 1, j - 1)$, $(i - 1, j)$, or $(i, j - 1)$.

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

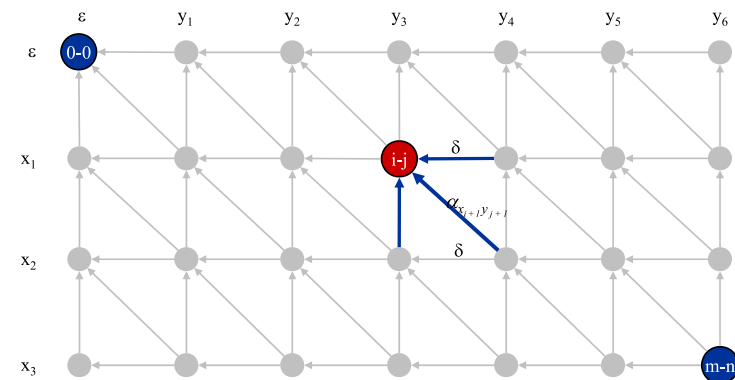
Edit Distance Graph

- 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 $O(m)$ space.



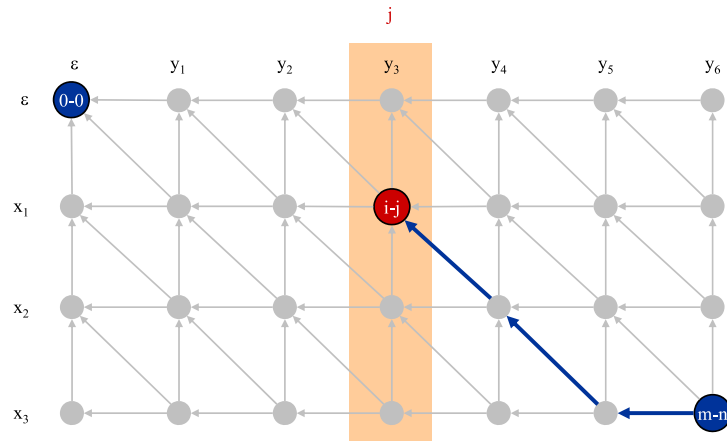
Edit Distance Graph

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



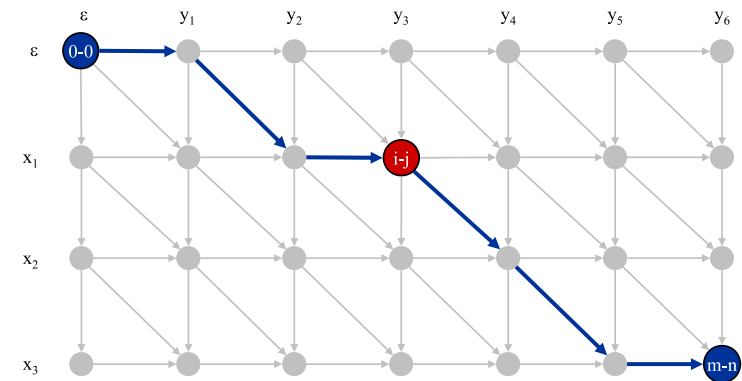
Edit Distance Graph

- 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 $O(m)$ space.



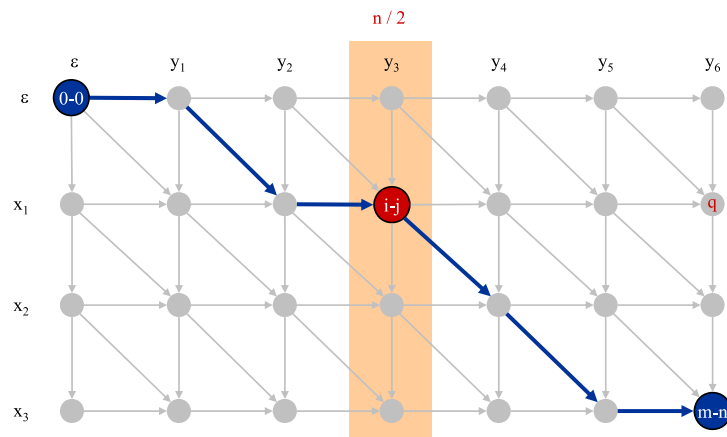
Edit Distance Graph

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



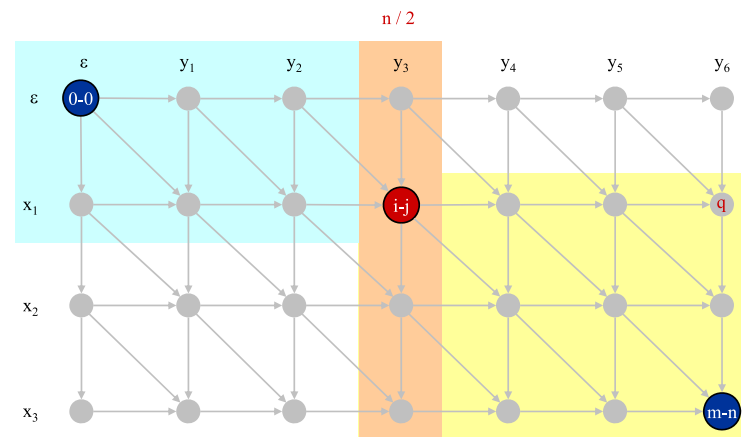
Edit Distance Graph

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 $(q, n/2)$.



Edit Distance Graph

Divide: find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP. Align x_q and $y_{n/2}$.
Conquer: recursively compute optimal alignment in each piece.



Running Time Analysis Warmup

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

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

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

Running Time Analysis

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

Proof: (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:

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

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

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

Running Time Analysis (Continued)

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

Proof:

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2cmn$.

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