

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 two 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 History

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

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
 - "it's impossible to use dynamic in a pejorative sense"
 - "something not even a Congressman could object to"

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

Dynamic Programming Applications

Areas.

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

Some famous dynamic programming algorithms.

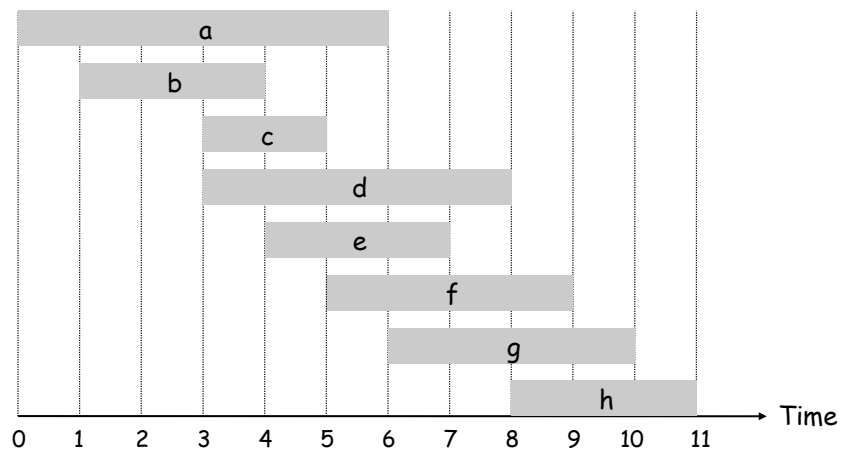
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

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.

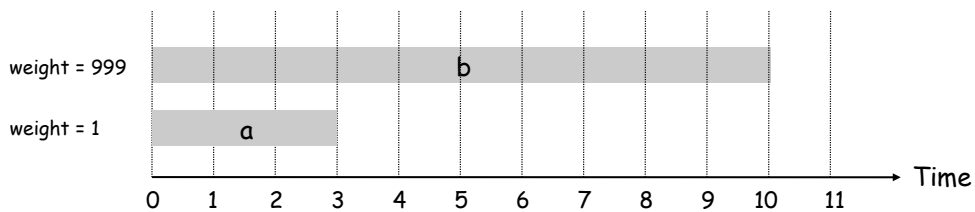


Unweighted Interval Scheduling Review

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.



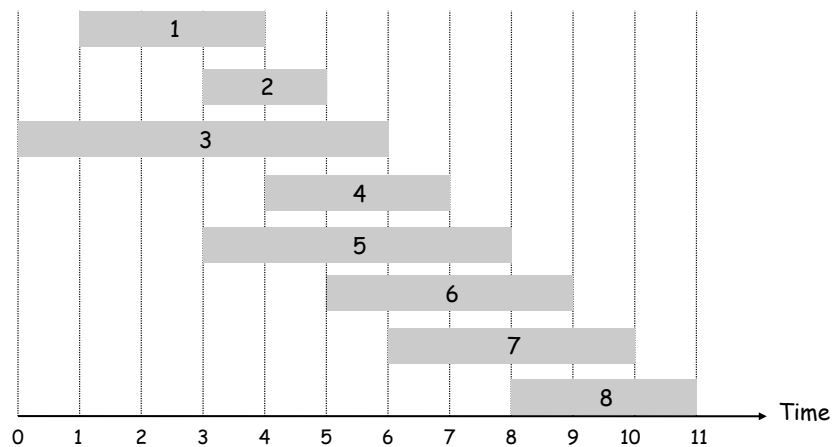
7

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



8

Dynamic Programming: Binary Choice

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

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

↙
↘
optimal substructure

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

9

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

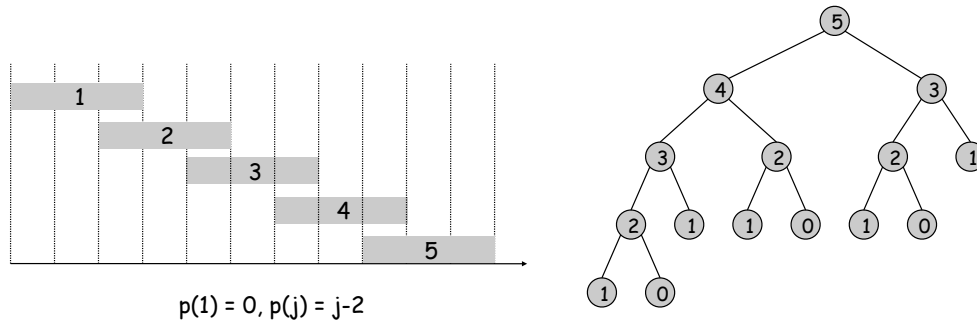
```
Compute-Opt(j) {
  if (j = 0)
    return 0
  else
    return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

10

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.



11

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] = empty ← global array
M[0] = 0
```

```
M-Compute-Opt(j) {
  if (M[j] is empty)
    M[j] = max( $v_j + M\text{-Compute-Opt}(p(j))$ , M-Compute-Opt(j-1))
  return M[j]
}
```

12

Weighted Interval Scheduling: Running Time

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

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$, $O(n)$ after sorting by start time :
scan both (s_1, s_2, \dots, s_n) & (f_1, f_2, \dots, f_n) .
- **M-Compute-Opt**(j): each invocation takes $O(1)$ time and either
 - (i) returns an existing value $M[j]$
 - (ii) fills in one new entry $M[j]$ and makes two recursive calls
- Progress measure $\Phi = \#$ nonempty entries of $M[]$.
 - initially $\Phi = 0$, throughout $\Phi \leq n$.
 - (ii) increases Φ by 1 \Rightarrow at most $2n$ recursive calls.
- Overall running time of **M-Compute-Opt**(n) is $O(n)$. ▪

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

13

Automated Memoization

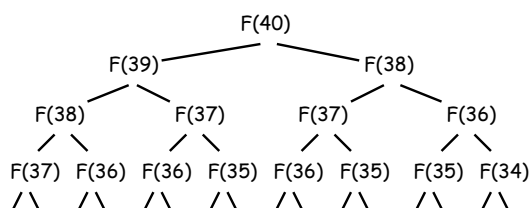
Automated memoization. Many functional programming languages (e.g., Lisp) have built-in support for memoization.

```
(defun F (n)
  (if (<= n 1)
      n
      (+ (F (- n 1)) (F (- n 2))))))
```

Lisp (efficient)

```
static int F(int n) {
  if (n <= 1) return n;
  else return F(n-1) + F(n-2);
}
```

Java (exponential)



14

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 $\leq n \Rightarrow O(n)$.

15

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  $n$ 
     $M[j] = \max(v_j + M[p(j)], M[j-1])$ 
}
```

16

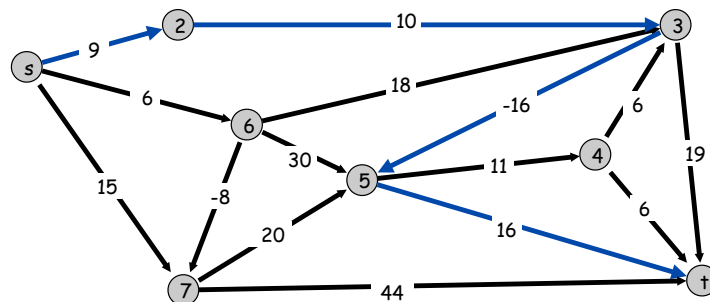
6.8 Shortest Paths

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 .

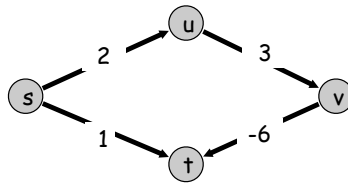
allow negative weights

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 .

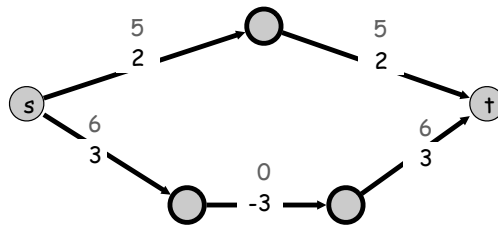


Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs.



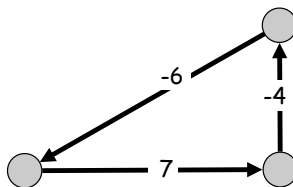
Re-weighting. Adding a constant to every edge weight can fail.



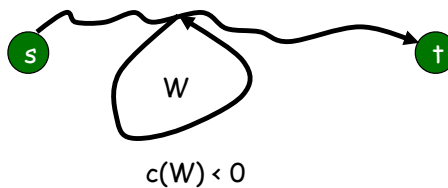
19

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.



20

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 uses exactly 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

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

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

21

Shortest Paths: Implementation

```
Shortest-Path( $G, t$ ) {  
  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  $(u, w) \in E$   
       $M[i, u] \leftarrow \min \{ M[i, u], M[i-1, w] + c_{uw} \}$   
}
```

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

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

22

Shortest Paths: Practical Improvements

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.

Theorem. 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.

Overall impact.

- Memory: $O(m + n)$.
- Running time: $O(mn)$ worst case, but substantially faster in practice.

23

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path( $G, s, t$ ) {
  foreach node  $v \in V$  {
     $M[v] \leftarrow \infty$ 
    successor[ $v$ ]  $\leftarrow \emptyset$ 
  }

   $M[t] = 0$ 
  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 ( $M[v] > M[w] + c_{vw}$ ) {
             $M[v] \leftarrow M[w] + c_{vw}$ 
            successor[ $v$ ]  $\leftarrow w$ 
          }
        }
      }
    }
    If no  $M[w]$  value changed in iteration  $i$ , stop.
  }
}
```

24

6.10 Negative Cycles in a Graph

Detecting Negative Cycles

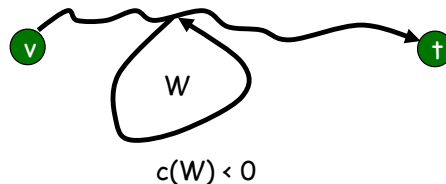
Lemma. If $OPT(n,v) = OPT(n-1,v)$ for all v , then no negative cycles.

Pf. Bellman-Ford algorithm.

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

Pf. (by contradiction)

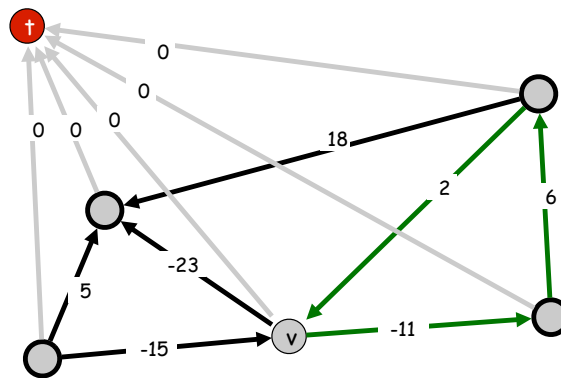
- Since $OPT(n,v) < OPT(n-1,v)$, we know P has exactly n edges.
- By pigeonhole principle, P must contain a directed cycle W .
- Deleting W yields a v - t path with $< n$ edges $\Rightarrow W$ has negative cost.



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 $OPT(n, v) = OPT(n-1, v)$ for all nodes v .
 - if yes, then no negative cycles
 - if no, then extract cycle from shortest path from v to t

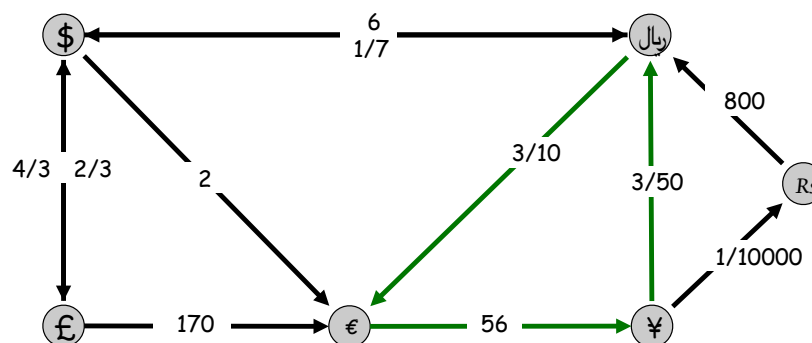


27

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!



28

Detecting Negative Cycles: Summary

Bellman-Ford. $O(mn)$ time, $O(m + n)$ space.

- Run Bellman-Ford for n iterations (instead of $n-1$).
- Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.
- See p. 288 for improved version and early termination rule.

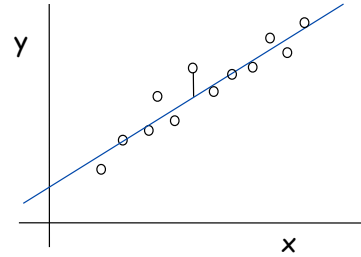
6.3 Segmented Least Squares

Segmented Least Squares

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 $y = ax + b$ that minimizes the sum of the squared error:

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



Solution. Calculus \Rightarrow min error is achieved when

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

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

31

Segmented Least Squares

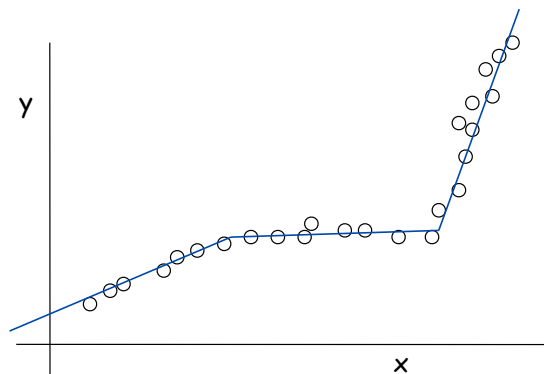
Segmented least squares.

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

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

↑
number of lines

↑
goodness of fit

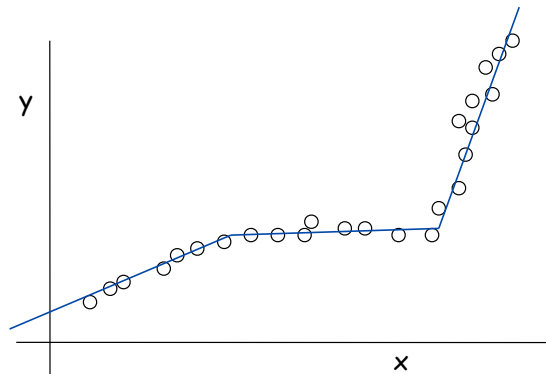


32

Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ 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 $c > 0$.



33

Dynamic Programming: Multiway Choice

Notation.

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

To compute $OPT(j)$:

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

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

34

Segmented Least Squares: Algorithm

```
INPUT:  $n, p_1, \dots, p_N, c$ 

Segmented-Least-Squares () {
   $M[0] = 0$ 
  for  $j = 1$  to  $n$ 
    for  $i = 1$  to  $j$ 
      compute the least square error  $e_{ij}$  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  $M[n]$ 
}
```

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

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

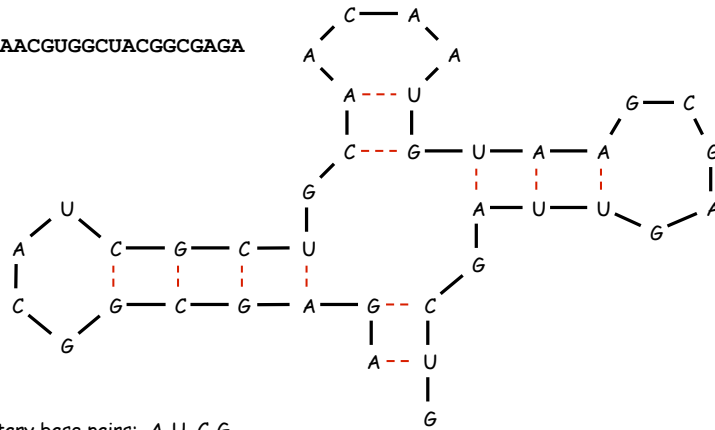
6.5 RNA Secondary Structure

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 behaviour of molecule.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA



complementary base pairs: A-U, C-G

37

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.

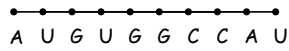
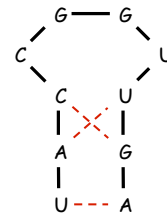
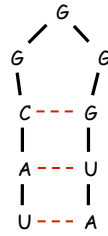
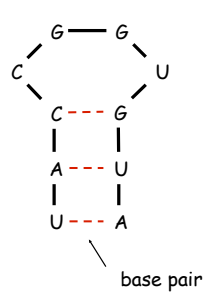
↑
approximate by number of base pairs

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

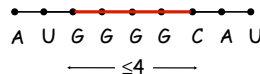
38

RNA Secondary Structure: Examples

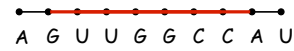
Examples.



ok



sharp turn



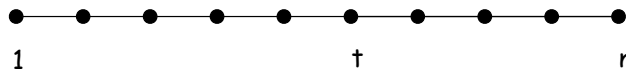
crossing

39

RNA Secondary Structure: Subproblems

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

match b_t and b_n



Difficulty. Results in two sub-problems.

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

40

Dynamic Programming Over Intervals

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

- **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
 - $OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

\uparrow
 $i \leq t < j-4$ and b_t and b_j
 are Watson-Crick complements

Remark. Same core idea in Cocke-Younger-Kasami algorithm to parse context-free grammars.

41

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  $M[1, n]$ 
}
    
```

\nwarrow
 using recurrence

	4	0	0	0	/
	3	0	0	/	/
	2	0	/	/	/
i	1	/	/	/	/
		6	7	8	9
					j

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

42

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. — Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

CKY parsing algorithm for context-free grammar has similar structure

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

Extra Slides

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.

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 ratio v_i / w_i .

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

Dynamic Programming: False Start

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

- Case 1: OPT does not select item i.
 - OPT selects best of { 1, 2, ..., 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!

47

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, ..., i-1 } using weight limit w
- Case 2: OPT selects item i.
 - new weight limit = $w - w_i$
 - OPT selects best of { 1, 2, ..., i-1 } using this new weight limit

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

48

Knapsack Problem: Bottom-Up

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

```

Input: n, 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]
    
```

49

Knapsack Algorithm

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

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

W = 11

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

50

Knapsack Problem: Running Time

Running time. $\Theta(nW)$.

- 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]

6.6 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

5 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

53

Edit Distance

Applications.

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

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

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

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

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

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

54

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.

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

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

Ex: CTACCG vs. TACATG.

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

	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, x1x2...xm, y1y2...yn, δ, α) {  
  for i = 0 to m  
    M[0, i] = iδ  
  for j = 0 to n  
    M[j, 0] = 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]  
}
```

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

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

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

6.7 Sequence Alignment in Linear Space

Sequence Alignment: Linear Space

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

Easy. Optimal **value** in $O(m + n)$ 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.

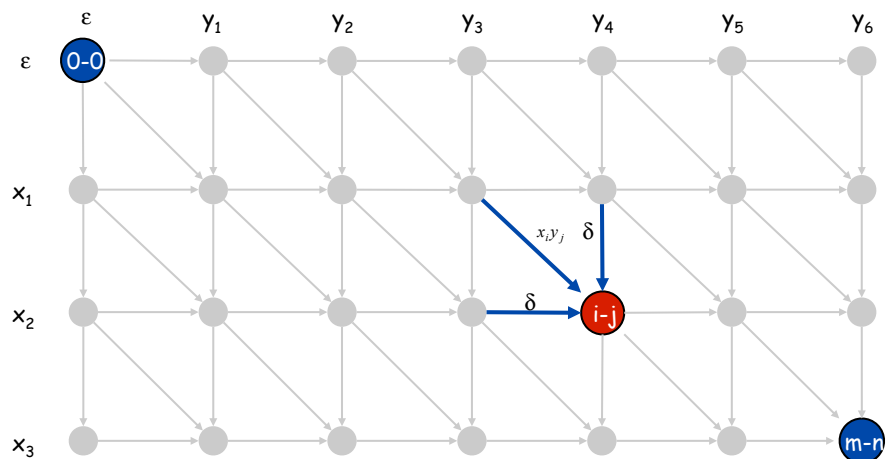
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

59

Sequence Alignment: Linear Space

Edit distance graph.

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

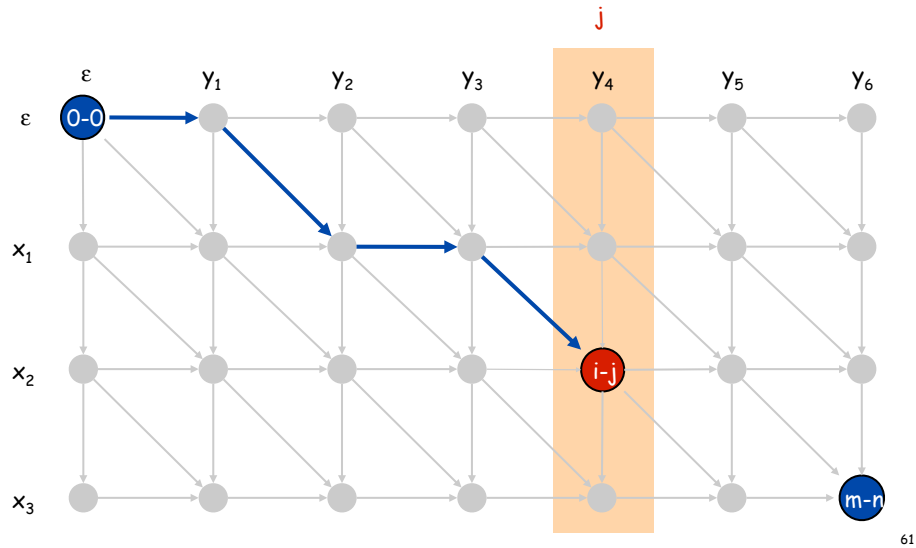


60

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.

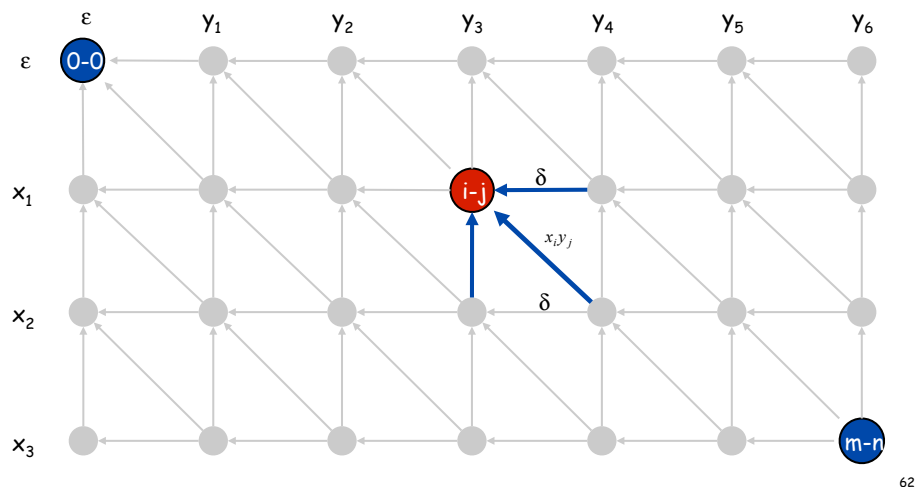


61

Sequence Alignment: Linear 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)

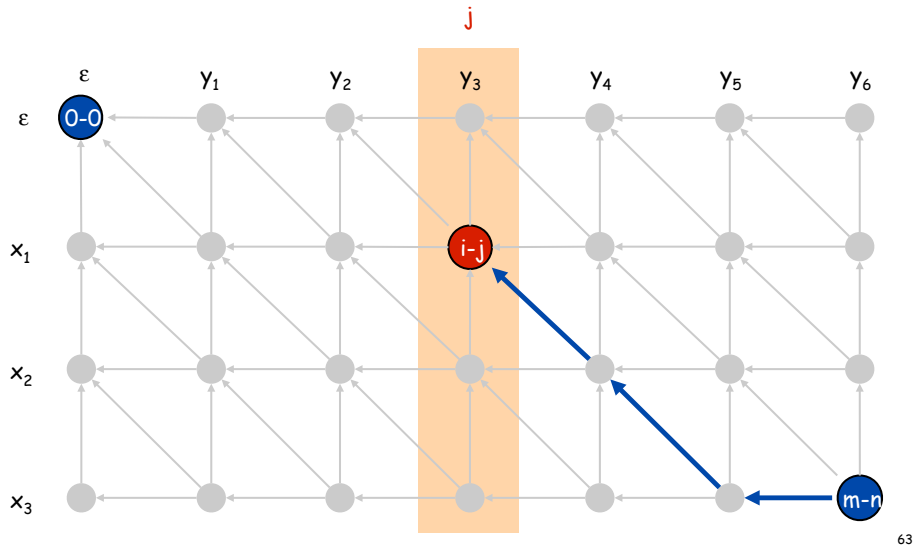


62

Sequence Alignment: Linear Space

Edit distance graph.

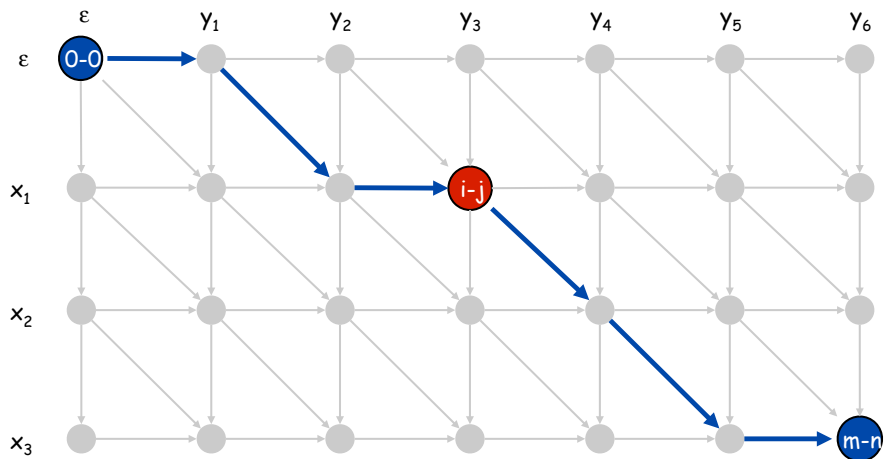
- Let $g(i, j)$ be shortest path from (i, j) to (m, n) .
- Can compute $g(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



63

Sequence Alignment: Linear Space

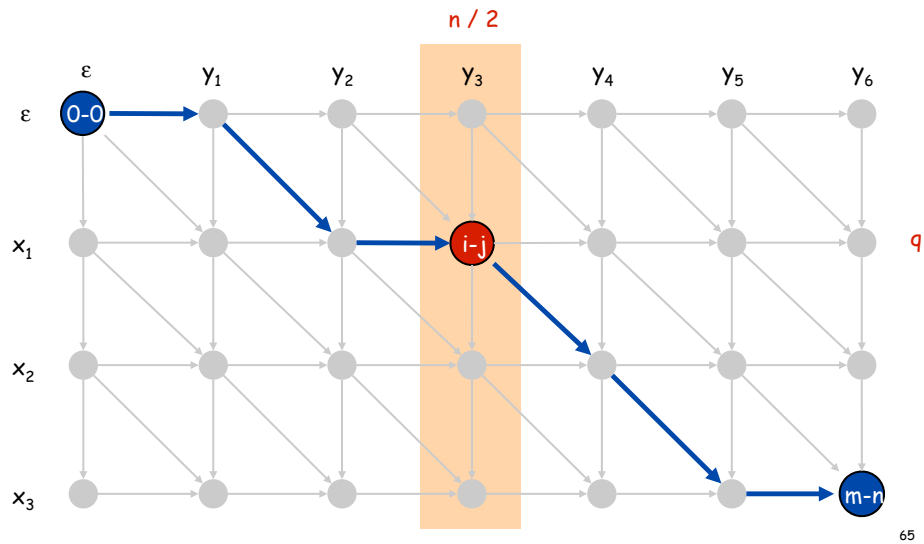
Observation 1. The cost of the shortest path that uses (i, j) is $f(i, j) + g(i, j)$.



64

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 $(q, n/2)$.

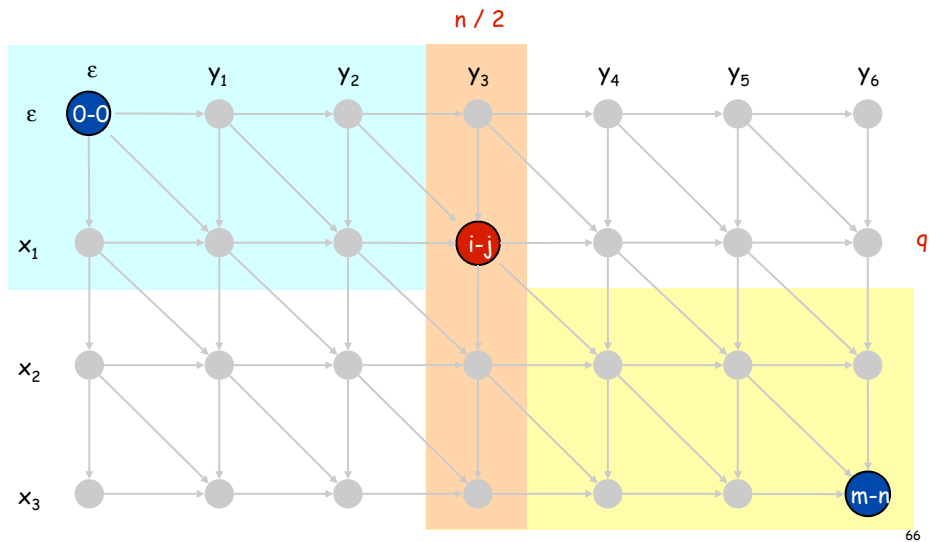


Sequence Alignment: Linear Space

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

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.



Sequence Alignment: Running Time Analysis Warmup

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

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

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.

67

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:

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

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

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

68

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. — naturally nonnegative, but Bellman-Ford used anyway!

Dijkstra's algorithm. Requires global information of network.

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

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

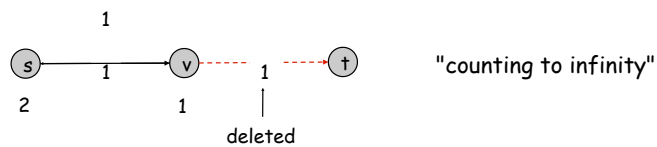
Distance Vector Protocol

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).



71

Path Vector Protocols

Link state routing.

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

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

72