COMP 355
Advanced Algorithms

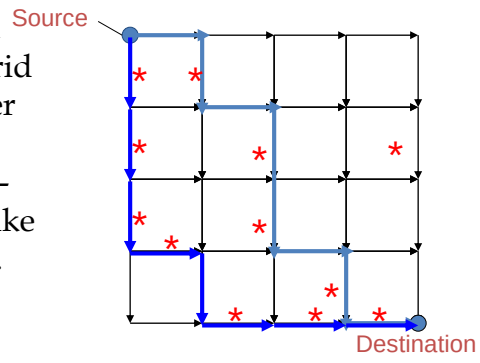
Dynamic Programming:
Manhattan Tourist Problem
Sequence Alignment
Section 6.6-6.7(KT)

Rhodes College

Yet Another DP Example

Manhattan Tourist Problem (MTP)

Imagine seeking a path from source to destination in a Manhattan-like city grid that maximizes the number of attractions (*) passed. With the following caveat– at every step you must make progress towards the goal.



We treat the city map as a graph, with "vertices" at each corner, and weighted edges along each block. The weights are the number of attractions along each block.

Manhattan Tourist Problem: Formulation

Goal: Find the maximum weighted path in a grid.

Input: A weighted grid ***G*** with two distinct vertices, one labeled "*source*" and the other labeled "*destination*"

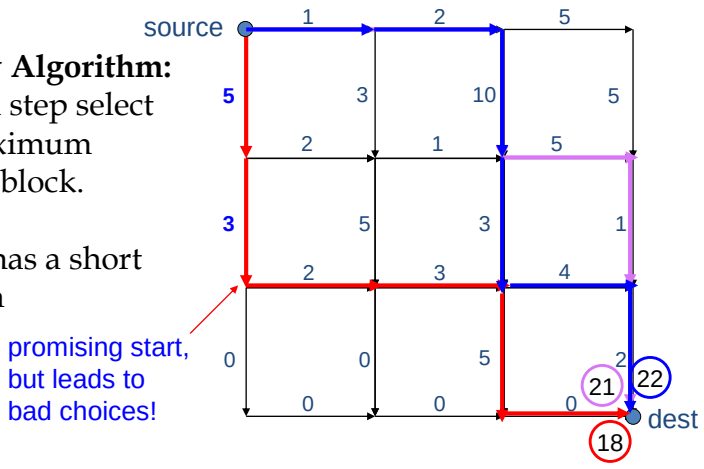
Output: A longest path in ***G*** from "*source*" to "*destination*"

3

MTP: Greedy Algorithm Is Not Optimal

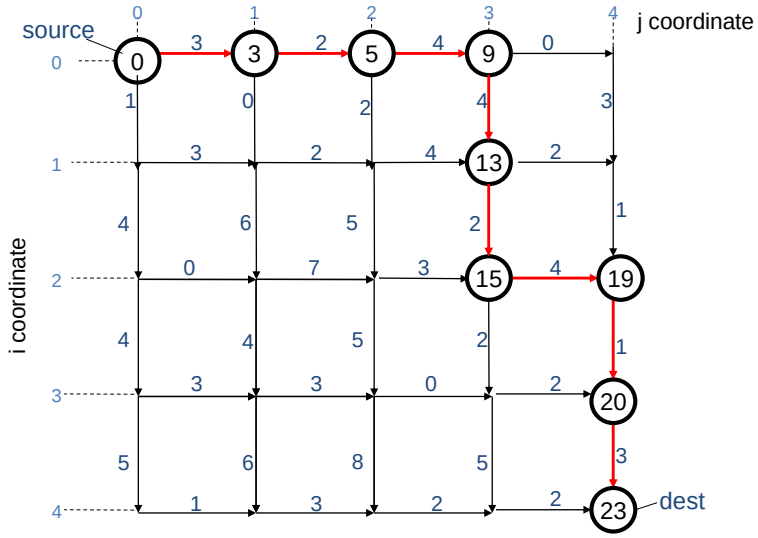
Greedy Algorithm:
At each step select the maximum weight block.

Greed has a short horizon



4

MTP as a Dynamic Program



5

MTP Strategy

- Instead of solving the Manhattan Tourist problem directly, (i.e. the path from (0,0) to (n,m)) we will solve a more general problem: find the longest path from (0,0) to any arbitrary vertex (i,j).
- If the longest path from (0,0) to (n,m) passes through some vertex (i,j), then the path from (0,0) to (i,j) must be the longest. Otherwise, you could increase your path by changing it.

6

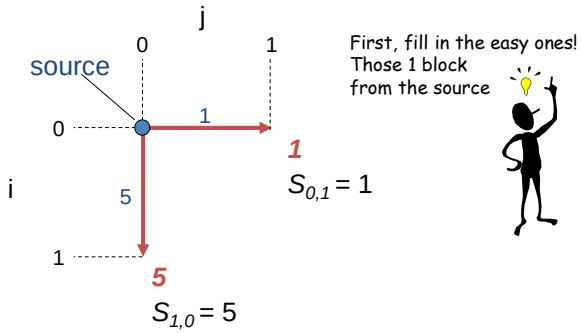
MTP: Simple Recursive Program

What's wrong with this approach?

It computes the same paths multiple times

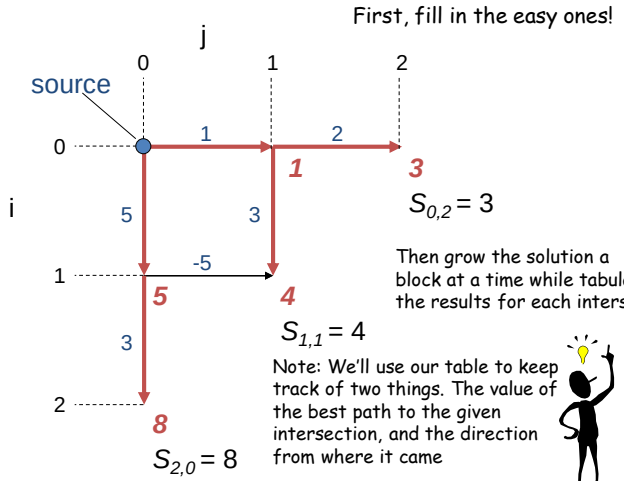
MT(*n,m*)
  if n = 0 and m = 0
    return 0
  if n = 0
    return *MT(0,m−1)* + len(edge) from *(0,m−1) to (0,m)*
  if m = 0
    return *MT(n−1, 0)* + len(edge) from *(n−1,0) to (n,0)*
  x ← *MT(n−1,m)* + len(edge) from *(n− 1,m) to (n,m)*
  y ← *MT(n,m−1)* + len(edge) from *(n,m−1) to (n,m)*
  return *max{x,y}*

MTP: Ordering Evaluations

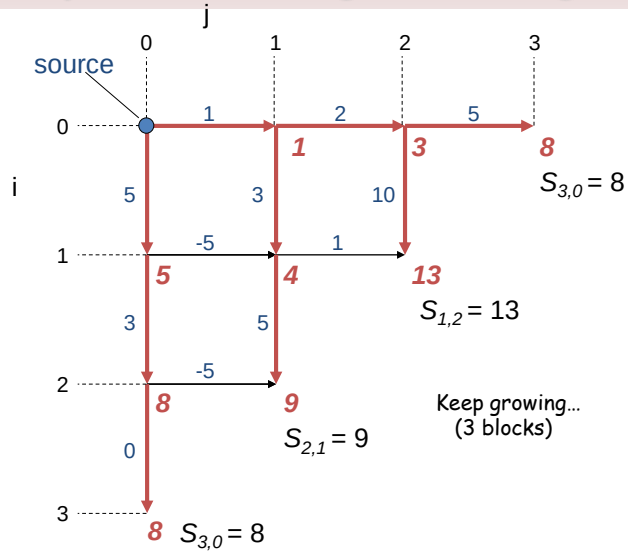


- Calculate optimal path score for each vertex in the graph
- Each vertex's score is the maximum of the prior vertices score plus the weight of the connecting edge in between

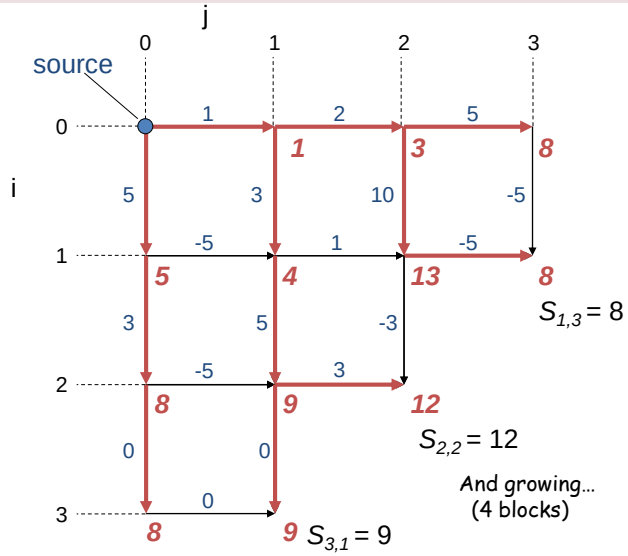
MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)

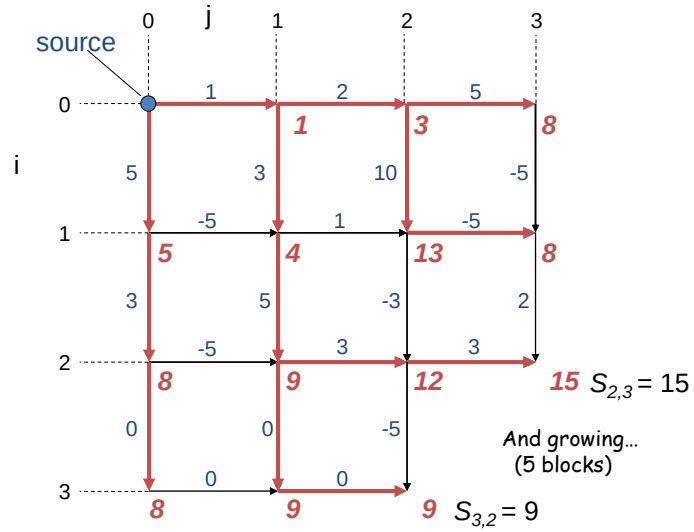


MTP: Dynamic Programming (cont'd)



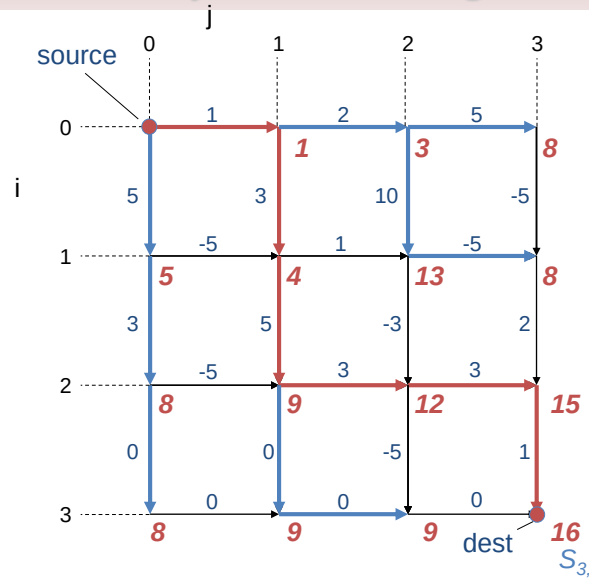
11

MTP: Dynamic Programming (cont'd)



12

MTP: Dynamic Programming (cont'd)



Once the "destination" node (intersection) is reached, we're done.

Our table will have the answer of the maximum number of attractions stored in the entry associated with the destination.

We use the "links" back in the table to recover the path. (Backtracking)

$S_{3,3} = 16$

13

MTP: Recurrence

Computing the score for a point *(i,j)* by the recurrence relation:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of the edge between } (i-1, j) \text{ and } (i, j) \\ s_{i,j-1} + \text{weight of the edge between } (i, j-1) \text{ and } (i, j) \end{cases}$$

Path to the intersection from the left

Path to the intersection from above

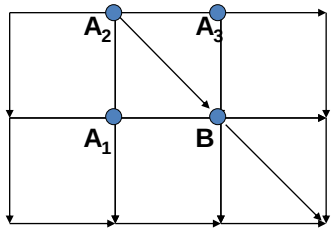
The running time is ***n x m*** for a ***n*** by ***m*** grid

(You visit all intersections once, and performed 2 tests)

(***n*** = # of rows, ***m*** = # of columns)

14

Manhattan Is Not A Perfect Grid



What about diagonals?

Broadway, Greenwich, etc.

- Easy to fix. Just adds more recursion cases.
- The score at point B is given by:

$$s_B = \max \begin{cases} s_{A1} + \text{weight of the edge } (A_1, B) \\ s_{A2} + \text{weight of the edge } (A_2, B) \\ s_{A3} + \text{weight of the edge } (A_3, B) \end{cases}$$

15

How similar are two strings?

Spell correction

- The user typed "graffe"

Which is closest?

- graf
- graft
- grail
- giraffe

Computational Biology

- Align two sequences of nucleotides

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC
```

- Resulting alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

Also for Machine Translation, Information Extraction, Speech Recognition

16

Minimum Edit Distance

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Editing Operations

- Insertion
- Deletion
- Substitution

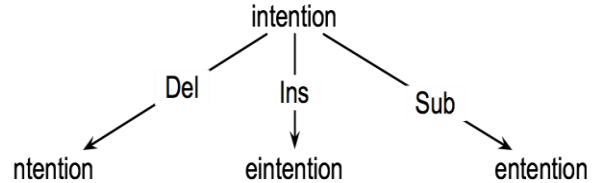
- If each operation has cost of 1
  - Distance between these is 5
- If substitutions cost 2 (Levenshtein)
  - Distance between them is 8

17

How to find the Min Edit Distance?

Searching for a path (sequence of edits) from the start string to the final string:

- **Initial state**: the word we're transforming
- **Operators**: insert, delete, substitute
- **Goal state**: the word we're trying to get to
- **Path cost**: what we want to minimize: the number of edits



18

Defining Min Edit Distance

- For two strings
  - X of length *n*
  - Y of length *m*
- We define D(*i,j*)
  - the edit distance between X[1..*i*] and Y[1..*j*]
    - i.e., the first *i* characters of X and the first *j* characters of Y
  - The edit distance between X and Y is thus D(*n,m*)

19

Defining Min Edit Distance (Levenshtein)

Initialization

```
D(i,0) = i
D(0,j) = j
```

Recurrence Relation:

```
For each  i = 1…M
      For each  j = 1…N
                          ⎧ D(i-1,j) + 1
                          ⎪
D(i,j)= min               ⎨ D(i,j-1) + 1
                          ⎪
                          ⎩ D(i-1,j-1) + ⎧ 2; if X(i) ≠ Y(j)
                                         ⎩ 0; if X(i) = Y(j)
```

Termination:

```
D(N,M) is distance
```

20

Min Edit Distance Algorithm

```
"Calculate Levenshtein edit distance for strings s1 and s2."
    len1 = len(s1) # vertically
    len2 = len(s2) # horizontally
    # Allocate the table
    table = [None]*(len2+1)
    for i in range(len2+1): table[i] = [0]*(len1+1)
    # Initialize the table
    for i in range(1, len2+1): table[i][0] = i
    for i in range(1, len1+1): table[0][i] = i
    # Do dynamic programming
    for i in range(1,len2+1):
        for j in range(1,len1+1):
            if s1[j-1] == s2[i-1]:
                d = 0
            else:
                d = 2
            table[i][j] = min(table[i-1][j-1] + d,
                              table[i-1][j]+1,
                              table[i][j-1]+1)
```

21

The Edit Distance Table

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1									
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

22

The Edit Distance Table

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1									
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

23

Edit Distance

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1	2								
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

24

The Edit Distance Table

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1	2	3	4	5	6	7	6	7	8
N	2	3	4	5	6	7	8	7	8	7
T	3	4	5	6	7	8	7	8	9	8
E	4	3	4	5	6	7	8	9	10	9
N	5	4	5	6	7	8	9	10	11	10
T	6	5	6	7	8	9	8	9	10	11
I	7	6	7	8	9	10	9	8	9	10
O	8	7	8	9	10	11	10	9	8	9
N	9	8	9	10	11	12	11	10	9	8

25

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.

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

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

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

$2\delta + \alpha_{CA}$

26

Sequence Alignment: Problem Structure

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

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

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

27

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?

28

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, •) from OPT(i-1, •).
- No longer a simple way to recover alignment itself.

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

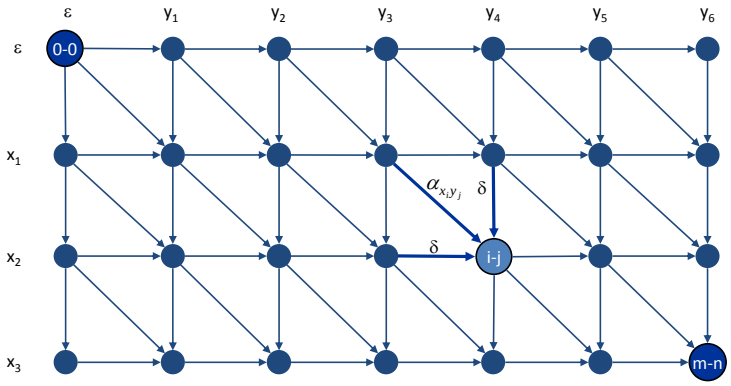
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

29

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).

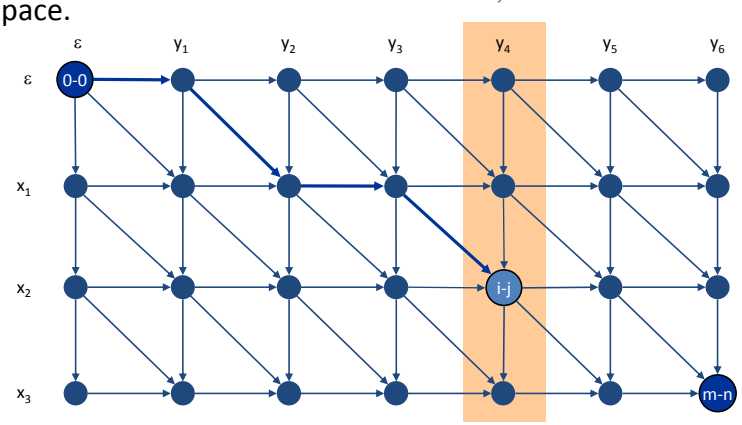


30

Sequence Alignment: Linear Space

Edit distance graph.

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



31

Space-Efficient Alignment Algorithm

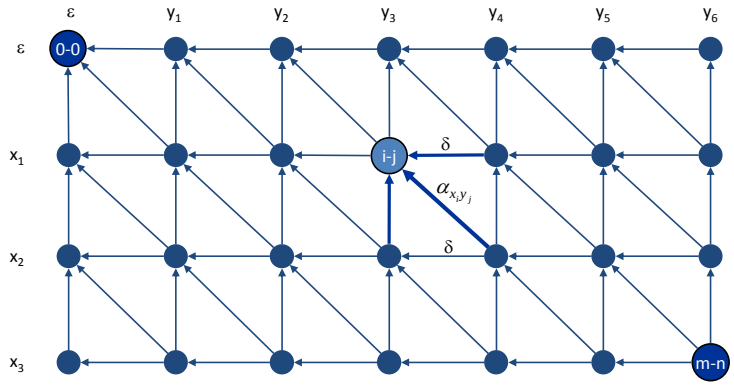
```
Space-Efficient-Alignment(X, Y)
  Array B[0 ... m, 0 ... 1]
  Initialize B[i, 0] = i δ (just as in column 0 of A)
  For j = 1, ..., n
     B[0, 1] = jδ (since this corresponds to entry A[0, j])
     For i = 1, ..., m
          B[i, 1] = min[α_{x_i y_j} + B[i − 1, 0],
                        δ + B[i − 1, 1], δ + B[i, 0]].
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
          Update B[i, 0] = B[i, 1] for each i
  Endfor
```

32

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)

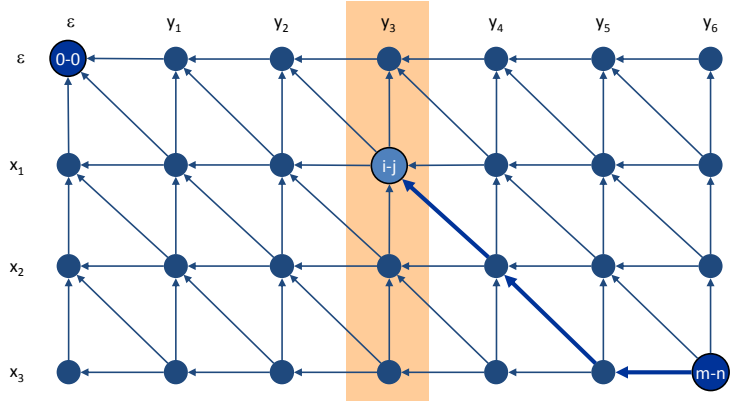


33

Sequence Alignment: Linear Space

Edit distance graph.

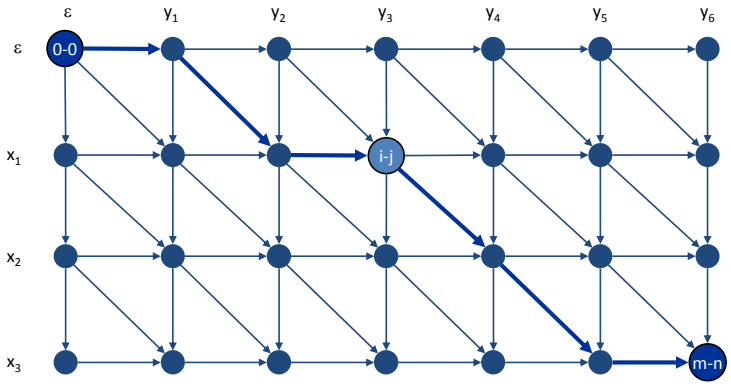
- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute g(•, j) for any j in O(mn) time and O(m + n) space.



34

Sequence Alignment: Linear Space

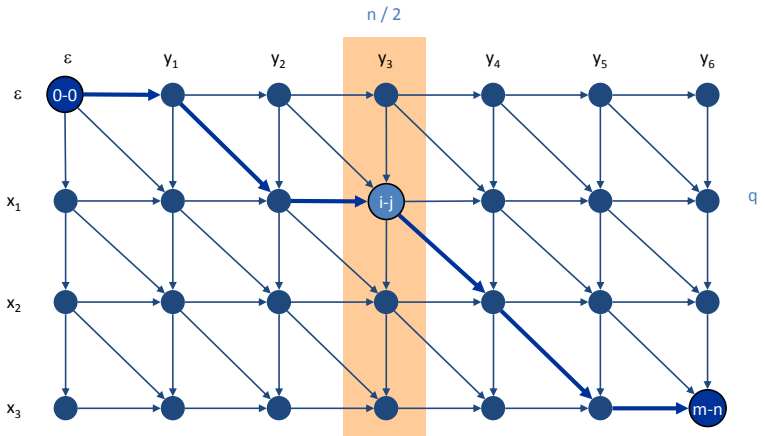
Observation 1. The cost of the shortest path that uses (i, j) is f(i, j) + g(i, j).



35

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).



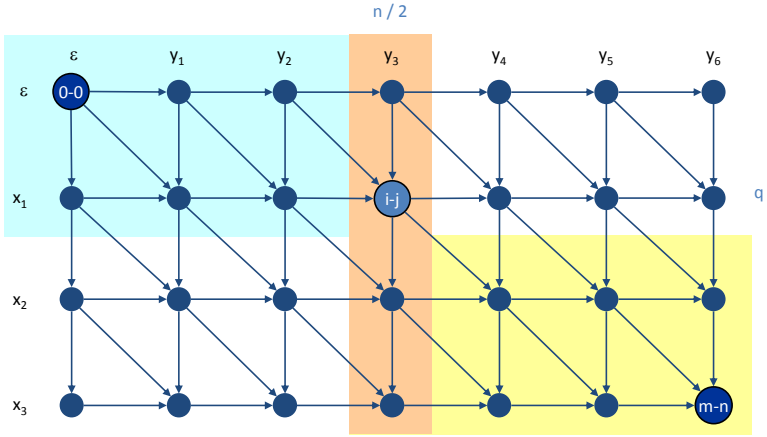
36

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.



37

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).

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

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.

38

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(•, n/2) and g (•, n/2) and find index q.
- T(q, n/2) + T(m - q, n/2) time for two recursive calls.
- Choose constant c so that:

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

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

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

39

Divide-and-Conquer Alignment Algorithm

```
Divide-and-Conquer-Alignment(X, Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X, Y)
  Call Space-Efficient-Alignment(X, Y[1 : n/2]),
     obtaining array B
  Call Backward-Space-Efficient-Alignment(X, Y[n/2 + 1 : n]),
     obtaining array B'
  Let q be the index minimizing B[q, 1] + B'[q, n]
  Add (q, n/2) to global list P
  Divide-and-Conquer-Alignment(X[1 : q], Y[1 : n/2])
  Divide-and-Conquer-Alignment(X[q + 1 : n], Y[n/2 + 1 : n])
  Return P
```

Rhodes College

40

Practice

Find the Levenshtein minimum edit distance of the words **mean** and **name**.

Assume a mismatch penalty of 2, and a gap (indel) penalty of 1.

Rhodes College