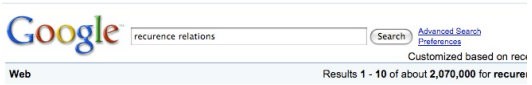
Objectives

- Dynamic Programming
  - Sequence Alignment – linear space
  - Shortest Path

Has This Ever Happened To You?



How does Google know what I really meant?

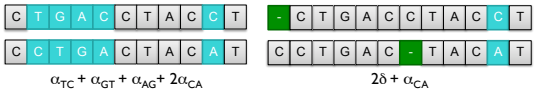
Review: Sequence Alignment

- What was the problem?
- What was our solution?

Edit Distance

- [Levenshtein 1966, Needleman-Wunsch 1970]
  - **Gap** penalty: δ
  - **Mismatch** penalty: α_{pq}
    - If p and q are the same, then mismatch penalty is 0
  - **Cost** = sum of gap and mismatch penalties

Parameters allow us to tweak cost



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

Sequence Alignment: Problem Structure

Gaps for remainder of Y

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

Ran out of 1st string

Ran out of 2nd string

Gaps for remainder of X

Sequence Alignment: Algorithm

Cost parameters

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)
   for i = 0 to m
      M[i, 0] = iδ          Ran out of y
   for j = 0 to n
      M[0, j] = jδ          Ran out of x

   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]
```

Example

X = bait **Y = boot**

α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

j →
i ↓

		b	a	i	t
b					
o					
o					
t					

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
```

What are the space costs?

When computing M[i,j], which entries in M are used?

Sequence Alignment: Analysis

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
```

Space Cost: O(mn)

Observation: to calculate the current value, we only need the row above us and the entry to the left

SEQUENCE ALIGNMENT IN LINEAR SPACE

Sequence Alignment: O(m) Space

- Collapse into an m x 2 array
  - M[i,0] represents previous row; M[i,1] -- current

```
Space-Efficient-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)
   for i = 0 to m        # initialize first row
      M[i, 0] = iδ
   for j = 1 to n
      M[0, 1] = jδ       # first gap

      for i = 1 to m
         M[i, 1] = min(α[xi, yj] + M[i-1, 0],
                       δ + M[i, 0],
                       δ + M[i-1, 1])
      for i = 1 to m     # copy current row into previous
         M[i, 0] = M[i, 1]
   return M[m, 1]
```

Any drawbacks?

Sequence Alignment: O(m) Space

- Collapse into an m x 2 array
  - M[i,0] represents previous row; M[i,1] -- current

```
Space-Efficient-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)
   for i = 0 to m        # initialize first row
      M[i, 0] = iδ
   for j = 1 to n
      M[0, 1] = jδ       # first gap

      for i = 1 to m
         M[i, 1] = min(α[xi, yj] + M[i-1, 0],
                       δ + M[i, 0],
                       δ + M[i-1, 1])
      for i = 1 to m     # copy current row into previous
         M[i, 0] = M[i, 1]
   return M[m, 1]
```

Finds optimal value but will not be able to find alignment

Why Do We Care About Space?

- For English words or sentences, probably doesn't matter
- Matters for Biological sequence alignment
  - Consider: 2 strings with 100,000 symbols each
    - Processor can do 10 billion primitive operations
    - BUT dealing with a 10 GB array

Sequence Alignment: Linear Space

- Can we avoid using quadratic space?
  - Optimal value in O(m) space and O(mn) time.
    - Compute OPT(i, •) from OPT(i-1, •)
    - BUT, no 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*

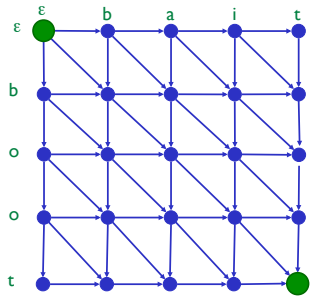
Recall Our Example

α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

X = bait **Y = boot**

i \ j		b	a	i	t
	0	2	4	6	8
b	2	0	2	4	6
o	4	2	1	3	5
o	6	4	3	2	4
t	8	6	5	4	2

Mapping to a Graph Problem



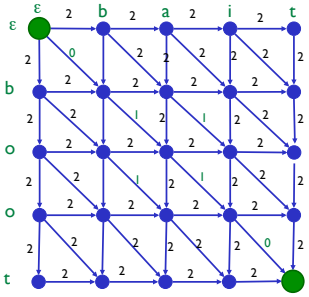
α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

- Horizontal and vertical edges cost δ
- Diagonal edges cost α

Goal: Find shortest path from top-left to bottom-right

Why is this formulation the same as the original?

Mapping to a Graph Problem



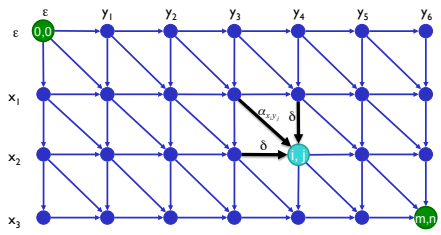
α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

- Horizontal and vertical edges cost δ
- Diagonal edges cost α

Goal: Find shortest path from top-left to bottom-right

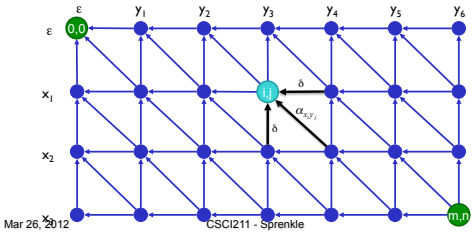
Sequence Alignment: Forward

- Edit distance graph (start)
  - Let f(i, j) be shortest path from (0,0) to (i, j)
  - Observation: f(i, j) = OPT(i, j)



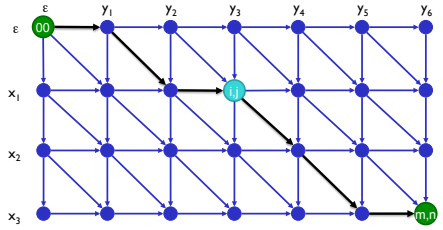
Sequence Alignment: Backward

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



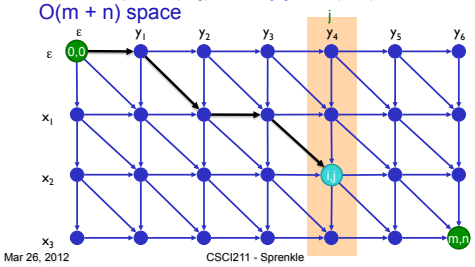
Sequence Alignment: Linear Space

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



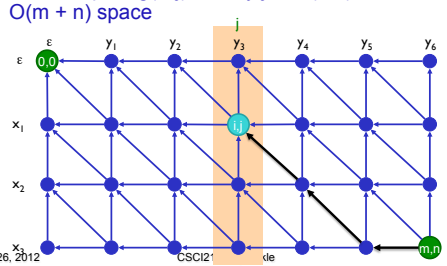
Sequence Alignment: Forward

- Edit distance graph
  - Let f(i, j) be shortest path from (0,0) to (i, j) (start)
  - Can compute f(*, j) for any j in O(mn) time and O(m + n) space



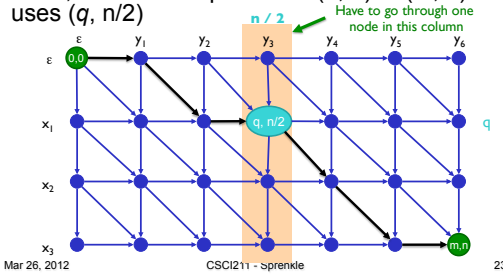
Sequence Alignment: Backward

- Edit distance graph
  - Let g(i, j) be shortest path from (m, n) to (i, j) (end)
  - Can compute g(*, j) for any j in O(mn) time and O(m + n) space



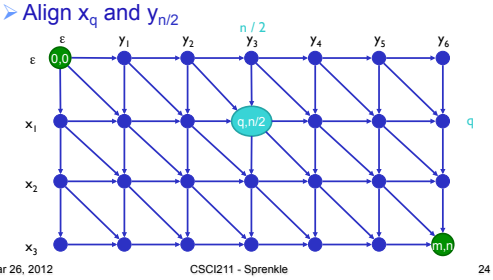
Sequence Alignment: Linear Space

- 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)



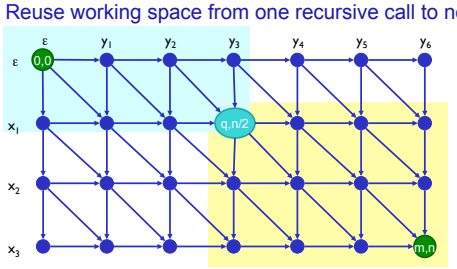
Sequence Alignment: Linear Space

- Divide: find index q that minimizes f(q, n/2) + g(q, n/2) using DP
  - Align x_q and $y_{n/2}$



Sequence Alignment: Linear Space

- Conquer: recursively compute optimal alignment in each piece
  - Reuse working space from one recursive call to next

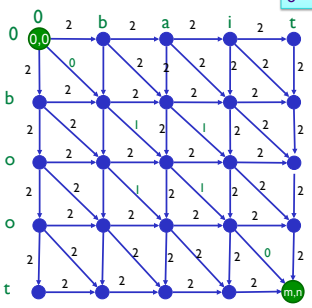


Divide and Conquer Sequence Alignment

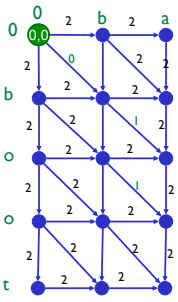
```
Create graph, label edges with weights

P contains node on shortest corner-to-corner path

Divide-and-Conquer-Alignment(X, Y)

Divide-and-Conquer-Alignment (X, Y):
    m = length of X
    n = length of Y
    if m <= 2 or n <= 2
        compute optimal alignment using Alignment(X, Y)
        return
    Space-Efficient-Alignment(X, Y[1:n/2])
    Backward-Space-Efficient-Alignment(X, Y[n/2+1:n])
    q = index that minimizes f(q, n/2) + g(q, n/2)
    add(q, n/2) to P
    Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
    Divide-and-Conquer-Alignment(X[q:m],Y[(n/2):n])
    return P
```


Example

α = 1, for vowel mismatch
α = 2, for other mismatches
δ = 2

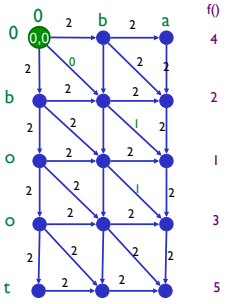


Space-efficient alignment: Left



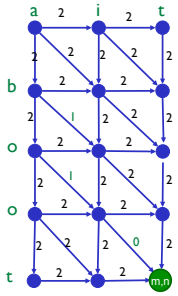
compute f (*, j), shortest path from (0,0) to (i, j)

Space-efficient alignment: Left

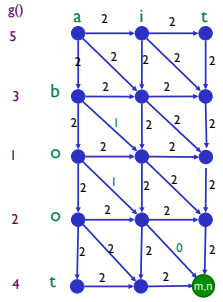


Backward Space Efficient

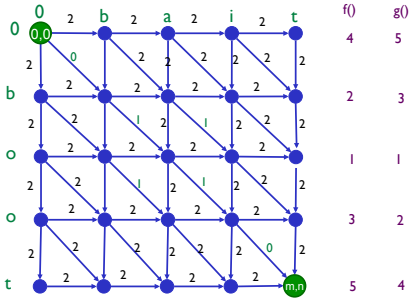
Compute g(*, j), shortest path from (m,n) to (i, j)



Backward Space Efficient

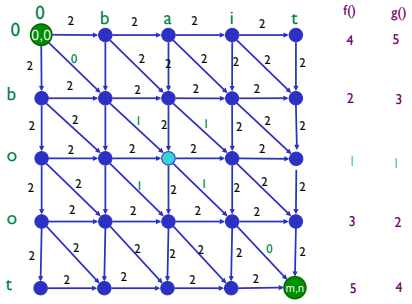


Example



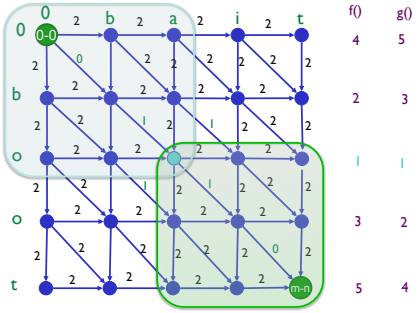
 Pick minimum sum

Example



 Pick minimum sum

Example



Divide and Conquer Sequence Alignment: Analysis

```
P contains node on shortest corner-to-corner path
Divide-and-Conquer-Alignment (X, Y)
    m = length of X
    n = length of Y
    if m <= 2 or n <= 2
        compute optimal alignment using Alignment(X, Y)
        return
    Space-Efficient-Alignment(X, Y[1:n/2])
    Backward-Space-Efficient-Alignment(X, Y[n/2+1:n])
    q = index that minimizes f(q, n/2) + g(q, n/2)
    add(q, n/2) to P
    Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
    Divide-and-Conquer-Alignment(X[q:m],Y[(n/2):n])
    return P
```

What is the recurrence relation?

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).

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

- Remark. Analysis is not tight because sub-problems are of size (q, n/2) and (m - q, n/2).

```
Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
Divide-and-Conquer-Alignment(X[q:m],Y[(n/2):n])
```

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)
- Recurrence Relation:

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

- Solve using substitution:

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

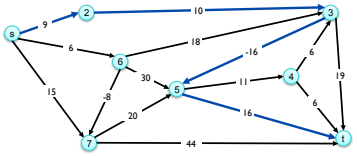
IMPROVING SHORTEST PATH

Shortest Paths

- **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

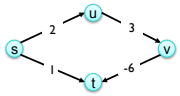
- Allows modeling other phenomena



Shortest Paths: Failed Attempts

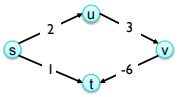
- Review: What was Dijkstra's algorithm?
  - Dijkstra can fail if negative edge costs

Shortest path from s →t?



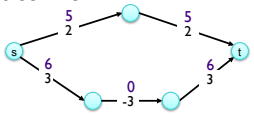
Shortest Paths: Failed Attempts

- Dijkstra. Can fail if negative edge costs



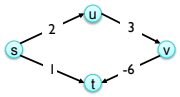
- Re-weighting. Adding a constant to every edge weight can fail

Why?



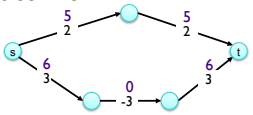
Shortest Paths: Failed Attempts

- Dijkstra. Can fail if negative edge costs



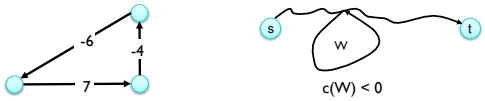
- Re-weighting. Adding a constant to every edge weight can fail

Why?



	Orig:	New:
	4	10
	3	12

Shortest Paths: Negative Cost Cycles



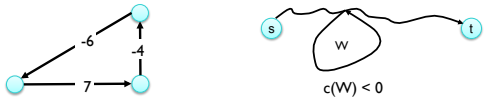
- If some path from *s* to *t* contains a negative cost cycle, there does **not** exist a shortest *s-t* path

Why?

- Otherwise, there exists one that is *simple* (i.e., does not repeat nodes)

What does this mean about number of edges in path?

Shortest Paths: Negative Cost Cycles

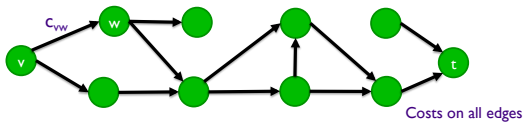


- If some path from *s* to *t* contains a negative cost cycle, there does **not** exist a shortest *s-t* path
- Otherwise, there exists one that is *simple* (i.e., does not repeat nodes)
  - Path has *at most n-1* edges, where *n* is # of nodes in graph

Towards a Recurrence

- **OPT(*i*,*v*)**: minimum cost of a ***v-t*** path *P* using ***at most i*** edges
  - This formulation eases later discussion
- Original problem is OPT(n-1, s)

Break down into subproblems based on *i* and *v*



Shortest Paths: Dynamic Programming

- OPT(i, v) = minimum cost of a ***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
    - Cost: cost of chosen edge

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

Shortest Paths: Implementation

Starting node

```
Shortest-Path(G, s)
   n = number of nodes in G
   foreach node v ∈ V
      M[0, v] = ∞
   M[0, s] = 0

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

Cost of chosen edge

- Shortest path length is M[n-1, s]

Starting node

Shortest Paths: Implementation

Starting node

```
Shortest-Path(G, s)
   n = number of nodes in G
   foreach node v ∈ V
      M[0, v] = ∞
   M[0, s] = 0  # distance to yourself is 0

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

Cost of chosen edge

- Shortest path length is M[n-1, s]

Starting node

Looking Ahead

- Wiki for Tuesday: 6 – 6.4
- PS 8 due Friday