Objectives

- Dynamic Programming
  - Sequence Alignment
  - Dijkstra's Algorithm

What was the Key to Solving each of these Problems?

- Weighted interval scheduling
- Segmented least squares
- Knapsack

What was the Key to Solving each of these Problems?

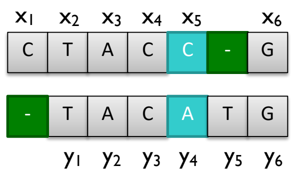
- Weighted interval scheduling
  - Binary decision: job was in or wasn't
  - Know conflicts→ reduce problem
- Segmented least squares
  - Knew last point was definitely in one segment
    - Could reduce
  - Multiway decision→ many possibilities for segment starting point
- Knapsack
  - If select an item, reduce available size by item's size
    - Find opt solution for smaller weight, remaining items

Review

- What is the sequence alignment problem?
  - What is our goal?
  - What problem does sequence alignment help us to solve?

Sequence Alignment Example

- X = CTACCG
- Y = TACATG
- Solution: M = x_2-y_1 , x_3-y_2, x_4-y_3, x_5-y_4 , x_6-y_6

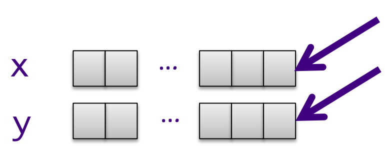


$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i:\, x_i \text{ unmatched}} \delta + \sum_{j:\, y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Recall: mismatch penalty is 0 if x_i and y_j are the same

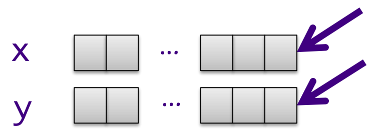
Sequence Alignment Case Analysis

- Consider last character of the strings X and Y: x_M and y_N
  - M and N are not necessarily equal
    - i.e., strings are not necessarily the same length
- What are the possibilities for x_M and y_N in terms of the alignment?



Sequence Alignment Case Analysis

- Consider last character of strings X and Y: x_M and y_N
  - Case 1: x_M and y_N are aligned
  - Case 2: x_M is not matched
  - Case 3: y_N is not matched

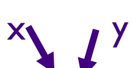


Formulate the optimal solution's value

Sequence Alignment Case Analysis

- Consider last character of strings X and Y: x_M and y_N
  - Case 1: x_M and y_N are aligned
  - Case 2: x_M is not matched
  - Case 3: y_N is not matched

What are the costs for these cases?



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

Sequence Alignment Cost Analysis

- Consider last character of strings X and Y: x_M and y_N
  - Case 1: x_M and y_N are aligned
    - Pay mismatch for x_M-y_N + min cost of aligning rest of strings
    - OPT(M, N) = α_{XmYn} + OPT(M-1, N-1)
  - Case 2: x_M is not matched
    - Pay gap for x_M + min cost of aligning rest of strings
    - OPT(M, N) = δ + OPT(M-1, N)
  - Case 3: y_N is not matched
    - Pay gap for y_N + min cost of aligning rest of strings
    - OPT(M, N) = δ + OPT(M, N-1)

Sequence Alignment Cost Analysis

- Base costs? → i or j is 0
  - What happens when we run out of letters in one string before the other?

```
X = CTACCG
Y = TACTG
```

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
   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]
```

Example

X = boot **Y = bait**

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

j →

i ↓

		b	a	i	t
b					
o					
o					
t					

Example

X = boot **Y = bait**

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

j →

i ↓

		b	a	i	t
	0	2	4	6	8
b	2				
o	4				
o	6				
t	8				

Example

X = boot **Y = bait**

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

j →

i ↓

		b	a	i	t
	0	2	4	6	8
b	2	0	2	4	6
o	4				
o	6				
t	8				

Example

X = boot **Y = bait**

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

j →

i ↓

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

Example

X = boot **Y = bait**

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

j →

i ↓

		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				

Example

What is the value for the problem?
What is the solution?

X = boot **Y = bait**

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

j →

i ↓

		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

Example

X = boot **Y = bait**

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

j →

i ↓

		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

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]
```

O(mn)

Costs?

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***
  - Section 6.7

Recall Our Example

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

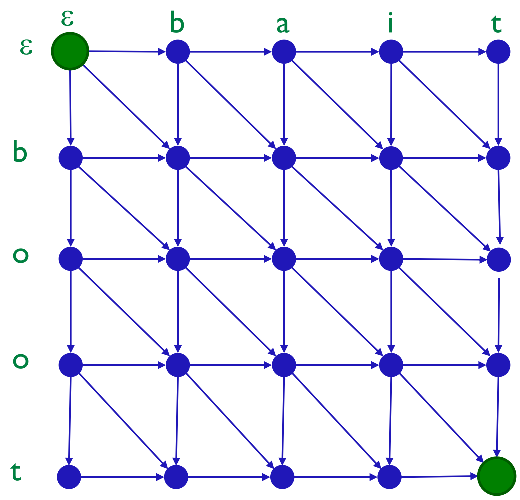
X = bait **Y = boot**

j →

i ↓

		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

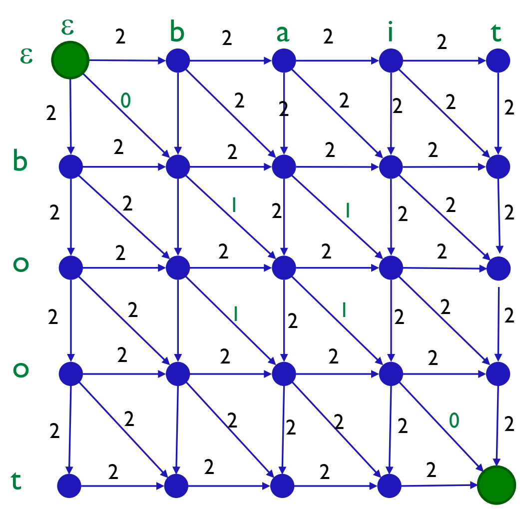
- Horizontal and vertical edges cost δ
- Diagonal edges cost α

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

Why is this formulation the same as the original?

Mar 27, 2019 CSCI211 - Sprenkle 29

Mapping to a Graph Problem



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

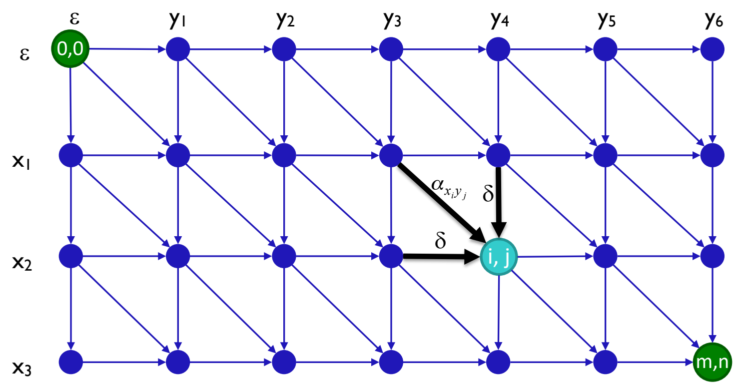
- Horizontal and vertical edges cost δ
- Diagonal edges cost α

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

Mar 27, 2019 CSCI211 - Sprenkle 30

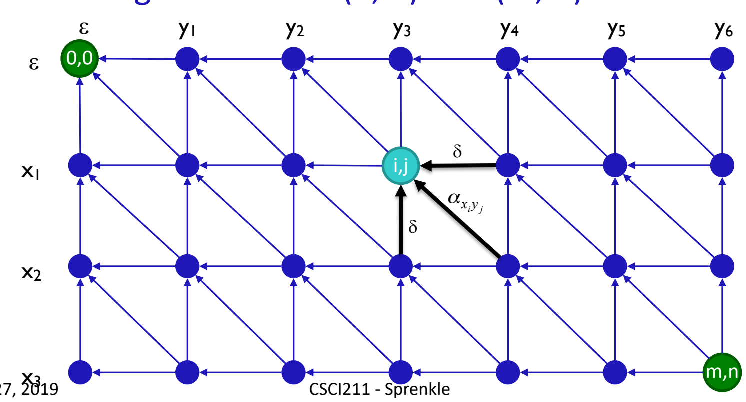
Sequence Alignment: Forward

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



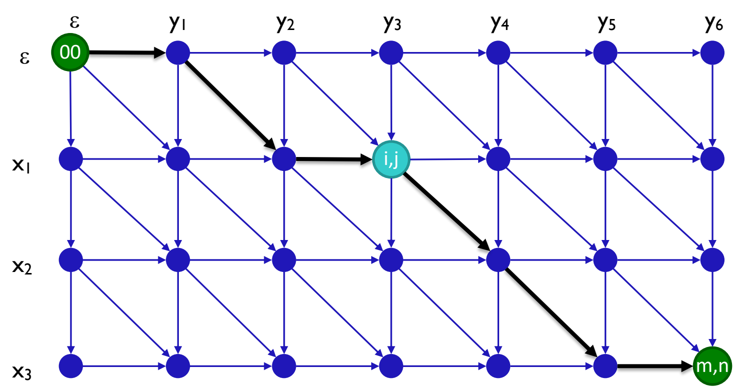
Sequence Alignment: Backward

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



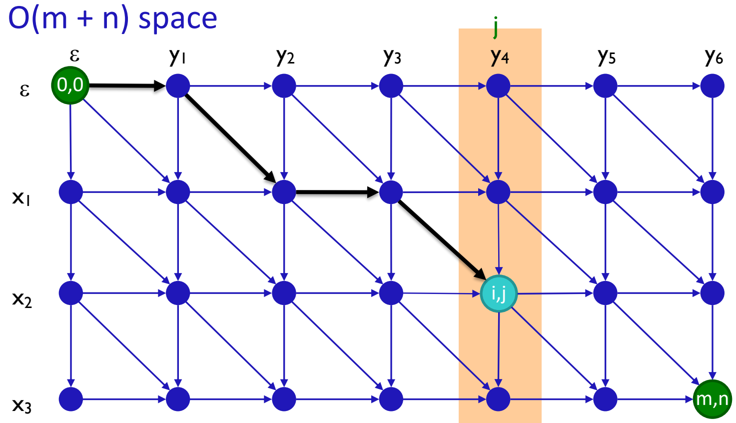
Sequence Alignment: Linear Space

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



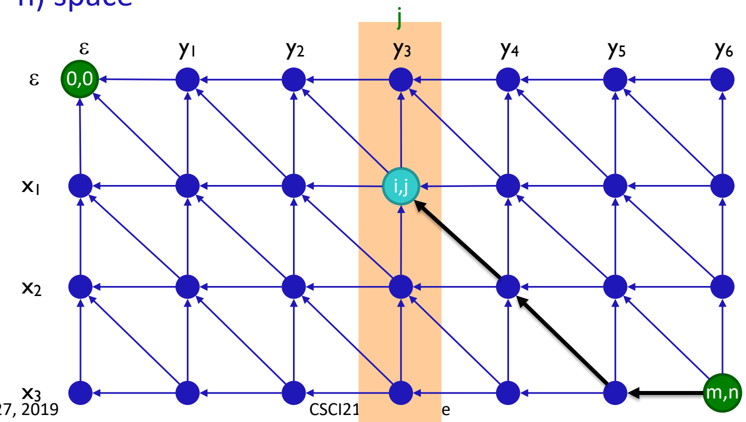
Sequence Alignment: Forward

- 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



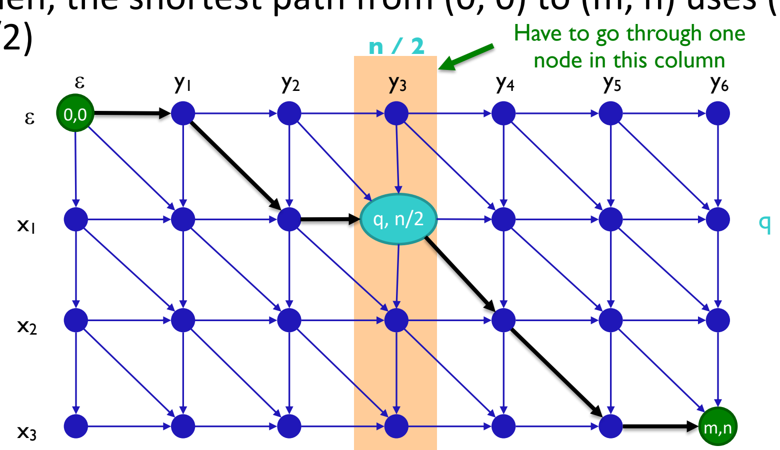
Sequence Alignment: Backward

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



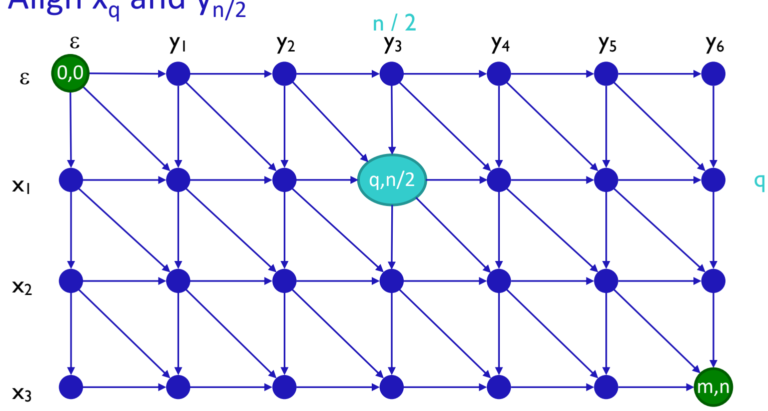
Sequence Alignment: Linear Space

- 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

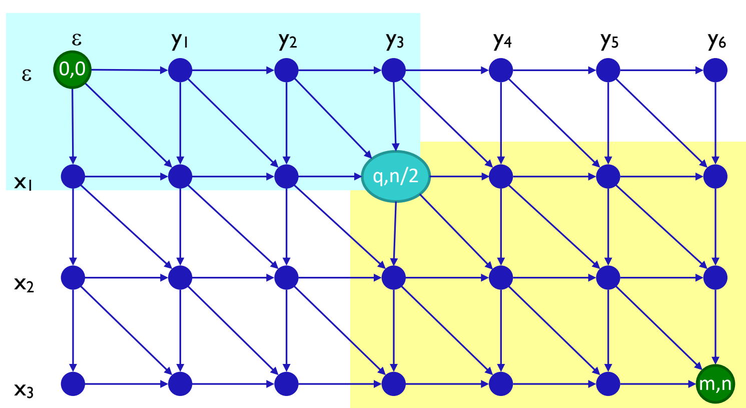
- Divide: find index q that minimizes f(q, n/2) + g(q, n/2) using DP
  - Align x_q and $y_{n/2}$



Mar 27, 2019 CSCI211 - Sprenkle 37

Sequence Alignment: Linear Space

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



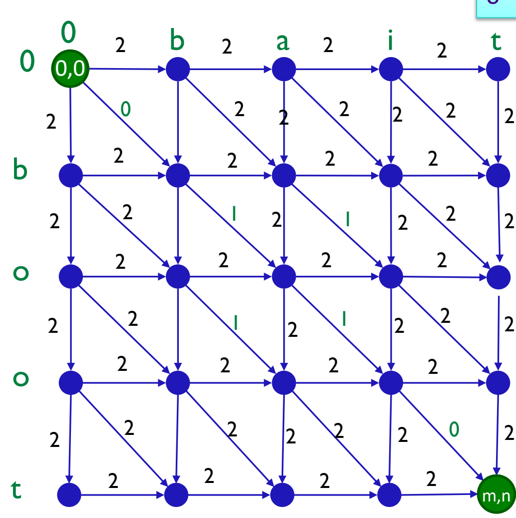
Mar 27, 2019 CSCI211 - Sprenkle 38

Divide and Conquer Sequence Alignment

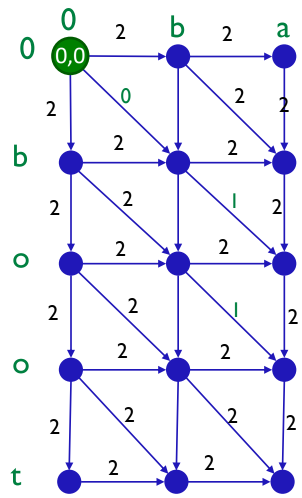
```
Create graph, label edges with weights

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

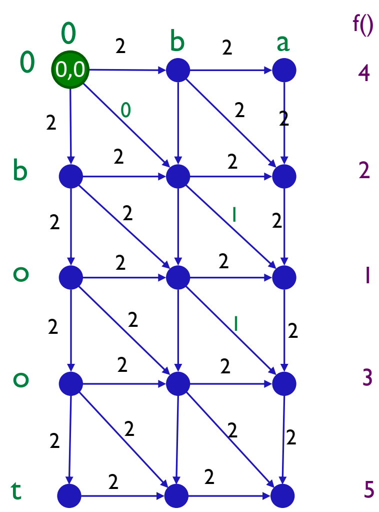


Space-efficient alignment: Left



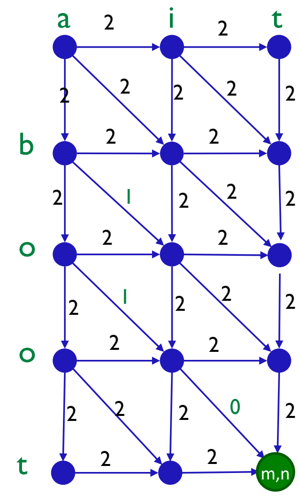
compute f (*, j), shortest path from (0,0) to (i, j)

Space-efficient alignment: Left



Backward Space Efficient

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

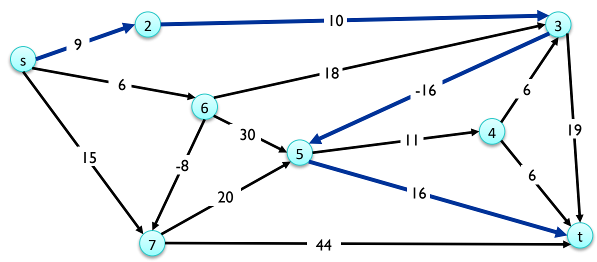


IMPROVING SHORTEST PATH

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

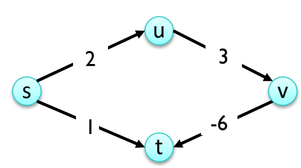
- Allows modeling other phenomena



Shortest Paths: Failed Attempts

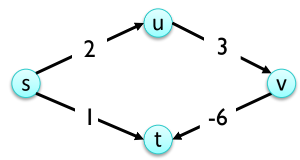
- Review: What was Dijkstra's algorithm?
  - Dijkstra can fail if negative edge costs

Shortest path from s →t?



Shortest Paths: Failed Attempts

- Dijkstra. Can fail if negative edge costs

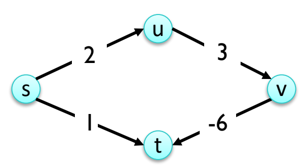


- Re-weighting. Adding a constant to every edge weight can fail

Why?

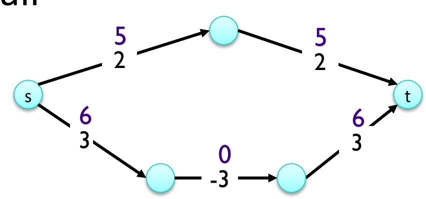
Shortest Paths: Failed Attempts

- Dijkstra. Can fail if negative edge costs



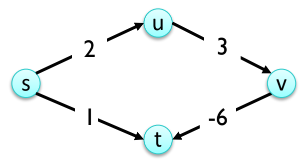
- Re-weighting. Adding a constant to every edge weight can fail

Why?



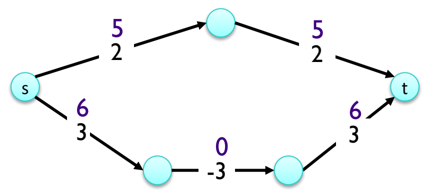
Shortest Paths: Failed Attempts

- Dijkstra. Can fail if negative edge costs



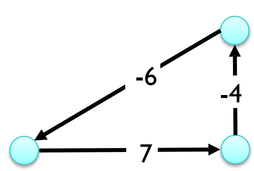
- Re-weighting. Adding a constant to every edge weight can fail

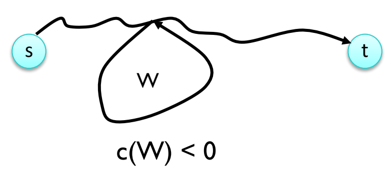
Why?



Orig:	New:
4	10
3	12

Shortest Paths: Negative Cost Cycles





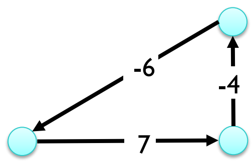
- If some path from *s* to *t* contains a negative cost cycle, there does **not** exist a shortest *s*-*t* path

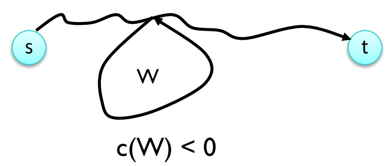
Why?

- 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



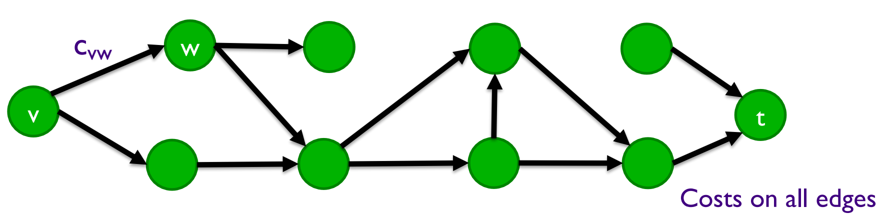


- If some path from *s* to *t* contains a negative cost cycle, there does **not** exist a shortest *s-t* path
- Otherwise, there exists one that is *simple* (i.e., does not repeat nodes)
  - Path has *at most n-1* edges, where *n* is # of nodes in graph

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

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

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] + cvw )
```

Cost of chosen edge

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

Starting node

Shortest Paths: Implementation

Starting node

Costs?

```
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] + cvw )
```

Cost of chosen edge

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

Starting node

Mar 27, 2019 CSCI211 - Sprenkle 55

Shortest Paths: Runtime Analysis

Starting node

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

   for i = 1 to n-1                        O(nm)
      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] + cvw )
```

Cost of chosen edge

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

Starting node

Mar 27, 2019 CSCI211 - Sprenkle 56

Dynamic Programming Wrapup

- What we didn't cover
  - 6.5: RNA Secondary Structure: Dynamic Programming Over Intervals
  - 6.7: Sequence Alignment in Linear Space
  - 6.9: Shortest Paths and Distance Vector Protocols
    - In practice

Looking Ahead

- PS8