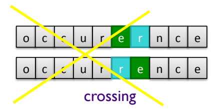
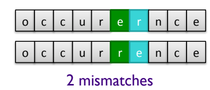
Objectives

- Dynamic Programming: sequence alignment
- Network Flow
  - Max flow
  - Min cut

Sequence Alignment

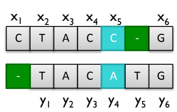
- Goal: Given two strings $X = x_1\, x_2\, .\,.\,.\, x_m$ and $Y = y_1\, y_2\, .\,.\,.\, y_n$ find alignment of minimum cost
- An *alignment* M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and **no** crossings
- The pair x_i-y_j and $x_{i'}$-$y_{j'}$ *cross* if i < i', but j > j'.





Sequence Alignment Example

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



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

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

Sequence Alignment: 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 = 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					

Example

X = bait **Y = boot**

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

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

Apr 1, 2016 CSCI211 - Sprenkle 7

Example

X = bait **Y = boot**

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

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

Apr 1, 2016 CSCI211 - Sprenkle 8

Example

X = bait **Y = boot**

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

		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				

Apr 1, 2016 CSCI211 - Sprenkle 9

Example

X = bait **Y = boot**

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

		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				

Apr 1, 2016 CSCI211 - Sprenkle 10

Example

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

X = bait **Y = boot**

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

		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

Apr 1, 2016 CSCI211 - Sprenkle 11

Example

X = bait **Y = boot**

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

		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

Apr 1, 2016 CSCI211 - Sprenkle 12

Sequence Alignment: Analysis

```
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]
```

O(mn)

Costs?

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]
```

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[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]
```

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

Dynamic Programming Wrapup

- What we didn't cover
  - 6.5: RNA Secondary Structure: Dynamic Programming Over Intervals
  - 6.7: Sequence Alignment in Linear Space
    - Dynamic programming + Divide and Conquer → oh my!
  - 6.8: Shortest Paths
  - 6.9: Shortest Paths and Distance Vector Protocols
    - In practice in internet routing

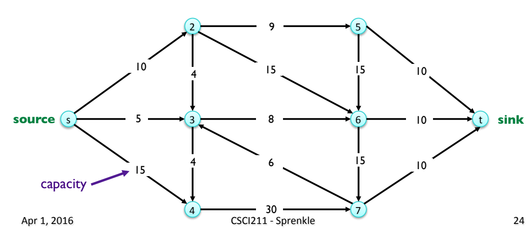
NETWORK FLOW

Motivating Flow Network Problems

- Modeling *transportation* networks
  - Edges: carry traffic
  - Nodes: pass traffic between edges
- Can represent many different types of problems
  - Instead of looking at all possibilities, formulate as a flow problem

Flow Network

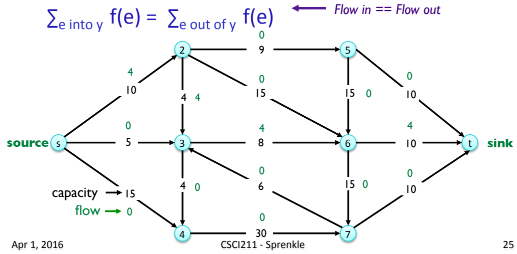
- G = (V, E) = directed graph, no parallel edges
- Two distinguished nodes: *s* = source, *t* = sink
- c(e) = capacity of edge e, > 0



Flows: Definitions

Flow can't exceed capacity

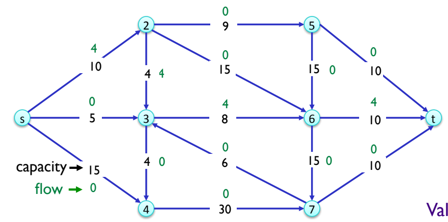
- An **s-t flow** is a function that satisfies
  - ***Capacity* condition**: For each e ∈ E: 0 ≤ f(e) ≤ c(e)
  - ***Conservation* condition**: For each v ∈ V – {s, t}:
    $\sum_{e \text{ into } y} f(e) = \sum_{e \text{ out of } y} f(e)$ ← *Flow in == Flow out*



Apr 1, 2016 CSCI211 - Sprenkle 25

Flows: Definitions

- The **value** of a flow f is $v(f) = \sum_{e \text{ out of } s} f(e)$



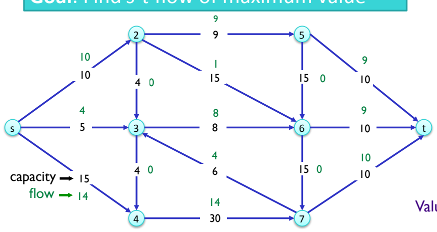
Value = 4

Apr 1, 2016 CSCI211 - Sprenkle 26

Maximum Flow Problem

- Make network most efficient
  - Use most of available capacity

Goal: Find *s-t* flow of maximum value

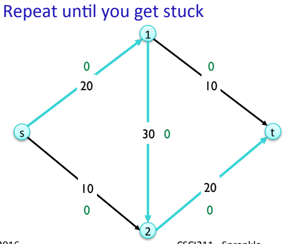


Value = 28

Apr 1, 2016 CSCI211 - Sprenkle 27

Towards a Max Flow Algorithm

- Greedy algorithm
  - Start all edges e ∈ E at f(e) = 0
  - Find an *s-t* path P with the most capacity: f(e) < c(e)
  - Augment flow along path P
  - Repeat until you get stuck

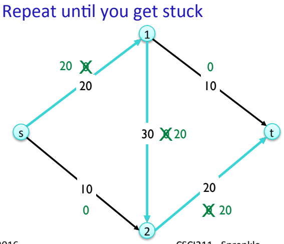


Flow value = 0

Apr 1, 2016 CSCI211 - Sprenkle 28

Towards a Max Flow Algorithm

- Greedy algorithm
  - Start all edges e ∈ E at f(e) = 0
  - Find an *s-t* path P with the most capacity: f(e) < c(e)
  - Augment flow along path P
  - Repeat until you get stuck



Is this optimal?

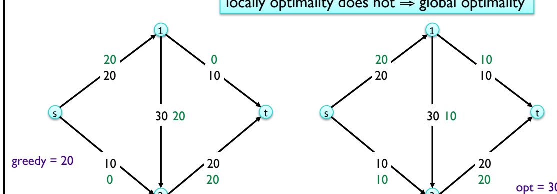
Flow value = 20

Apr 1, 2016 CSCI211 - Sprenkle 29

Towards a Max Flow Algorithm

- Greedy algorithm
  - Start all edges e ∈ E at f(e) = 0
  - Find an *s-t* path P with the most capacity: f(e) < c(e)
  - Augment flow along path P
  - Repeat until you get stuck

locally optimality does not ⇒ global optimality



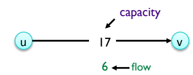
Apr 1, 2016 CSCI211 - Sprenkle 30

Towards a solution…

RESIDUAL GRAPHS

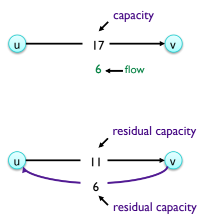
Towards a Residual Graph

- Original edge: e = (u, v) ∈ E
  - Flow f(e), capacity c(e)



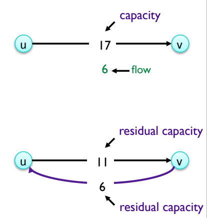
Towards a Residual Graph

- Original edge: e = (u, v) ∈ E
  - Flow f(e), capacity c(e)
- Residual edge
  - e = (u, v) w/ capacity c(e) - f(e)
  - e^R = (v, u) with capacity f(e)
    - To undo flow



Residual Graph: G_f

- Original edge: e = (u, v) ∈ E
  - Flow f(e), capacity c(e)
- Residual edge
  - e = (u, v) w/ capacity c(e) - f(e)
  - e^R = (v, u) with capacity f(e)
    - To undo flow
- Residual graph: $G_f = (V, E_f)$
  - Residual edges with *positive* residual capacity
  - $E_f = \underbrace{\{e : f(e) < c(e)\}}_{\textit{Forward edges}} \cup \underbrace{\{e^R : f(e) > 0\}}_{\textit{Backward edges}}$



Applying Residual Graph

- Used to find the maximum flow
  - Use similar idea to greedy algorithm
- Residual path: simple *s*-*t* path in G_f
  - Also known as *augmenting path*

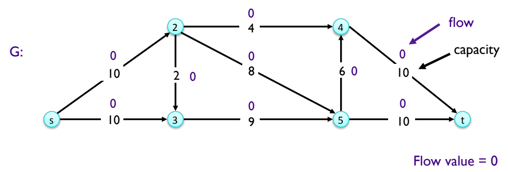
Augmenting Path Algorithm

c=capacity

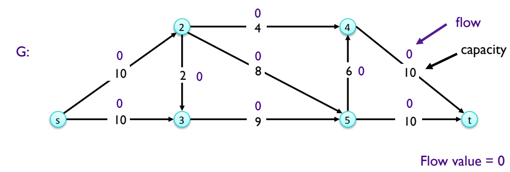
```
Ford-Fulkerson(G, s, t, c)
   foreach e ∈ E  f(e) = 0  # initially no flow
   Gf = residual graph

   while there exists augmenting path P
      f = Augment(f, c, P)      # change the flow
      update Gf              # build a new residual graph

   return f
```

```
Augment(f, c, P)
   b = bottleneck(P) # edge on P with least capacity
   foreach e ∈ P
      if (e ∈ E) f(e) = f(e) + b  # forward edge, ↑ flow
      else       f(eR) = f(e) - b  # forward edge, ↓ flow
   return f
```

Ford-Fulkerson Algorithm

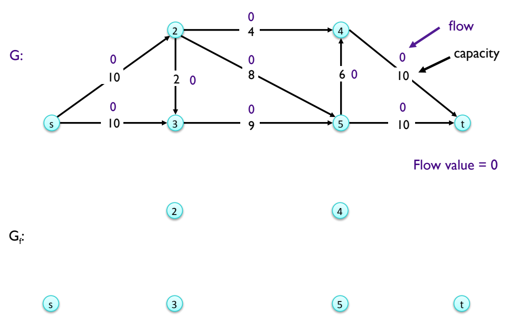


Ford-Fulkerson Algorithm

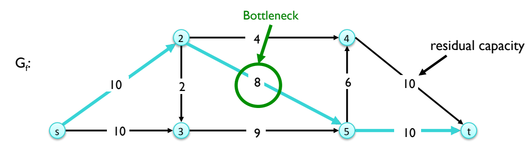


What does the residual graph look like?

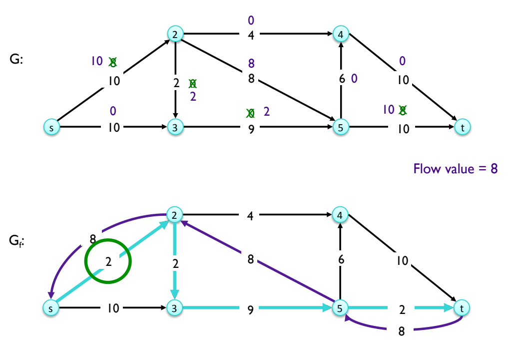
Ford-Fulkerson Algorithm



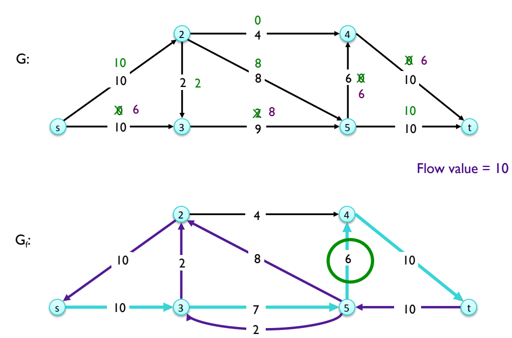
Ford-Fulkerson Algorithm



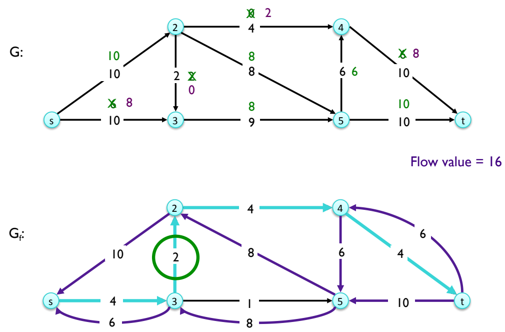
Ford-Fulkerson Algorithm



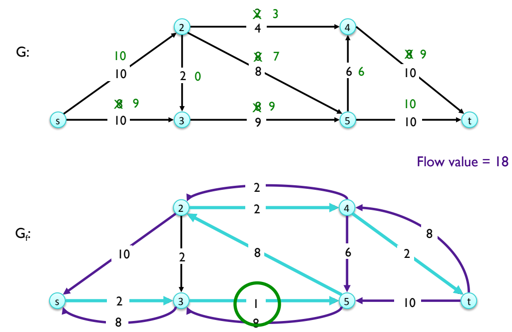
Ford-Fulkerson Algorithm



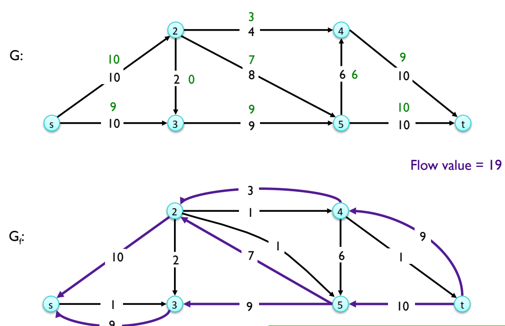
Ford-Fulkerson Algorithm



Ford-Fulkerson Algorithm

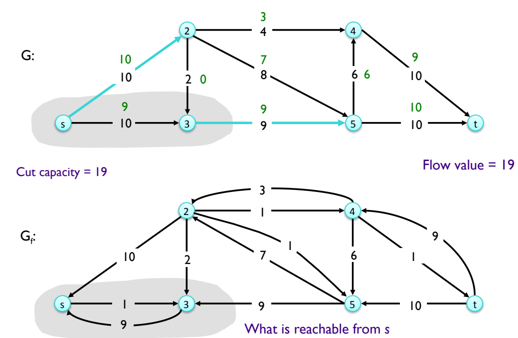


Ford-Fulkerson Algorithm



How do we know we're done?

Ford-Fulkerson Algorithm



Analyzing Augmenting Path Algorithm

```
Ford-Fulkerson(G, s, t, c)
   foreach e ∈ E  f(e) = 0  # initially no flow
   Gf = residual graph

   while there exists augmenting path P
      f = Augment(f, c, P)     # change the flow
      update Gf              # build a new residual graph

   return f
```

```
Augment(f, c, P)
   b = bottleneck(P) # edge on P with least capacity
   foreach e ∈ P
      if (e ∈ E) f(e) = f(e) + b  # forward edge, ↑ flow
      else       f(eR) = f(e) - b  # forward edge, ↓ flow
   return f
```

Why does alg work? What is happening at each iteration?
What is the running time?

Need more analysis …

Looking Ahead

- PS 9 (last one!) due Friday
  - See Course schedule page for starter code.
- Wiki due Monday – Network flows focus.