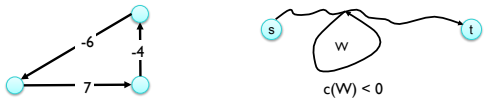
Objectives

- Dynamic Programming: shortest paths
- Network Flow
  - Max flow
  - Min cut

Shortest Paths: Where we left off…

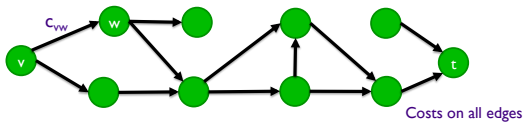


- Dijkstra's algorithm does not handle negative edge costs
- If some path from *s* to *t* contains a negative cost cycle, there does **not** exist a shortest *s-t* path
- Otherwise, there exists one that is *simple* (i.e., does not repeat nodes)
  - Path has *at most n-1* edges
    - where *n* is # of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



Costs on all edges

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

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] + c_vw )
```

Cost of chosen edge

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

Starting node

Shortest Paths: Runtime Analysis

```
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] + c_vw )
```

Starting node

Cost of chosen edge

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

Starting node

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!
  - 6.9: Shortest Paths and Distance Vector Protocols
    - In practice in internet routing

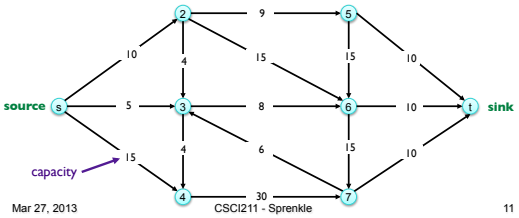
NETWORK FLOW

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

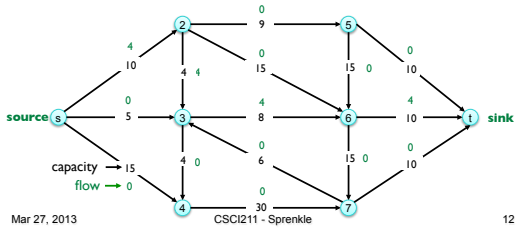
- G = (V, E) = directed graph, no parallel edges
- Two distinguished nodes: *s* = source, *t* = sink
- c(e) = capacity of edge e, > 0



Flows: Definitions

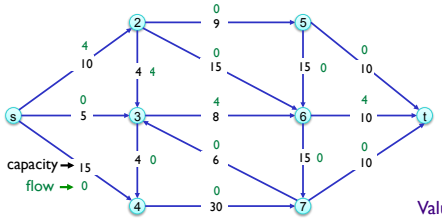
Flow can't exceed capacity

- 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*



Flows: Definitions

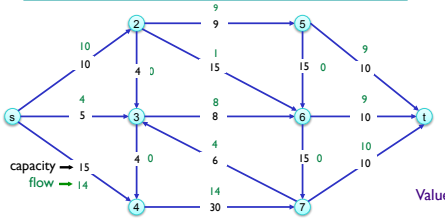
- The **value** of a flow f is v(f) = $\sum_{e \text{ out of } s} f(e)$



Maximum Flow Problem

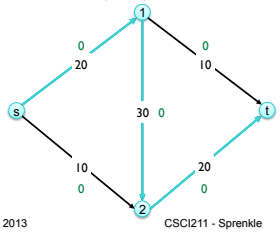
- Make network most efficient
  - Use most of available capacity

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



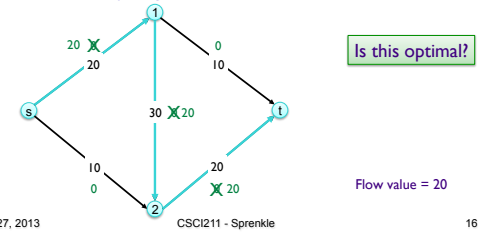
Towards a Max Flow Algorithm

- 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



Towards a Max Flow Algorithm

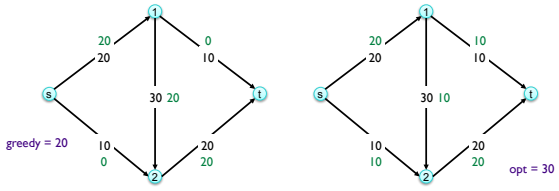
- Greedy algorithm
  - Start all edges e ∈ E at f(e) = 0
  - Find an *s-t* path P with the most capacity: f(e) < c(e)
  - Augment flow along path P
  - Repeat until you get stuck



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

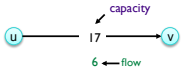


Towards a solution…

RESIDUAL GRAPHS

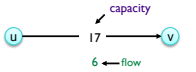
Towards a Residual Graph

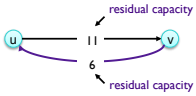
- Original edge: e = (u, v) ∈ E
  - Flow f(e), capacity c(e)



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

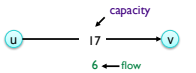


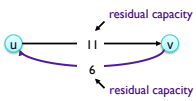


Residual Graph: G_f

- 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
  - $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$

Forward edges *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*

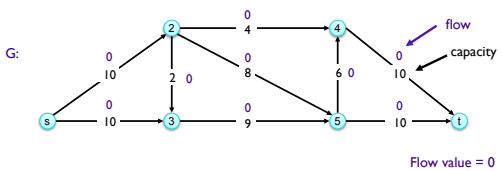
Augmenting Path Algorithm

c=capacity

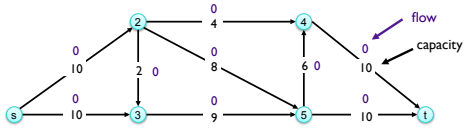
```
Ford-Fulkerson(G, s, t, c)
   foreach e ∈ E  f(e) = 0  # initially no flow
   Gf = residual graph

   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



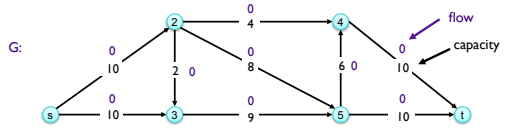
Ford-Fulkerson Algorithm



Flow value = 0

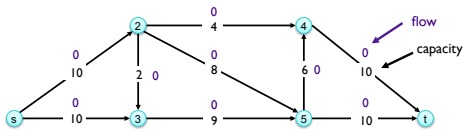
What does the residual graph look like?

Ford-Fulkerson Algorithm

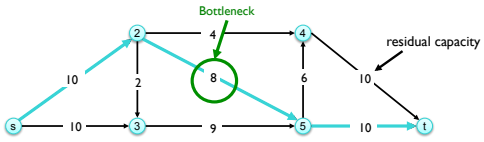


Flow value = 0

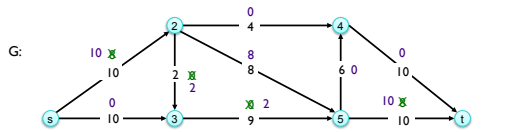
Ford-Fulkerson Algorithm



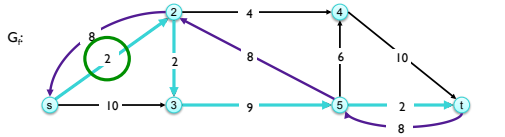
Flow value = 0



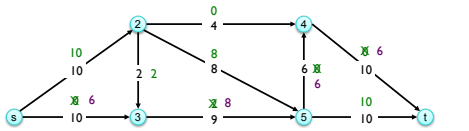
Ford-Fulkerson Algorithm



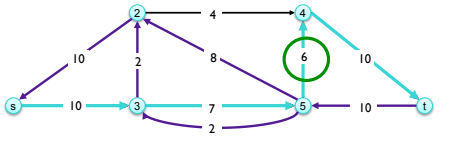
Flow value = 8



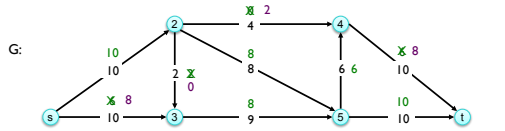
Ford-Fulkerson Algorithm



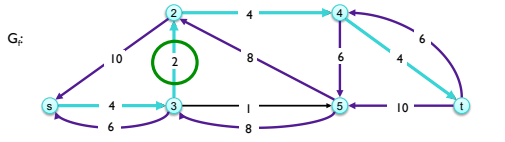
Flow value = 10



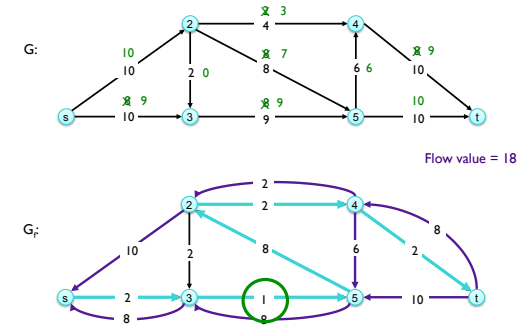
Ford-Fulkerson Algorithm



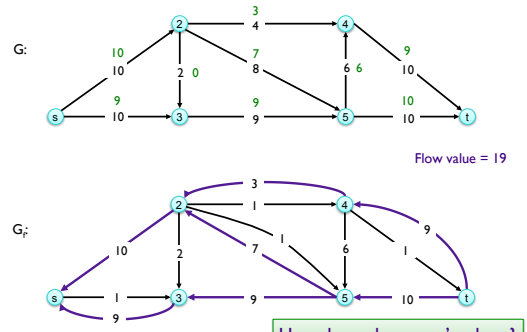
Flow value = 16



Ford-Fulkerson Algorithm

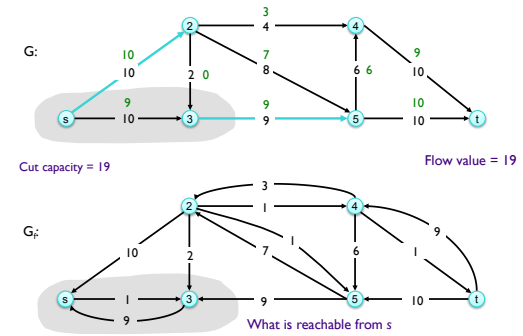


Ford-Fulkerson Algorithm



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 …

This Week

- Problem Set 8 due Friday
- Start reading chapter 7