CS 6363.005.19S Lecture 18—March 28, 2019

Main topics are #all-pairs_shortest_paths and #max_flow-min_cut.

Prelude

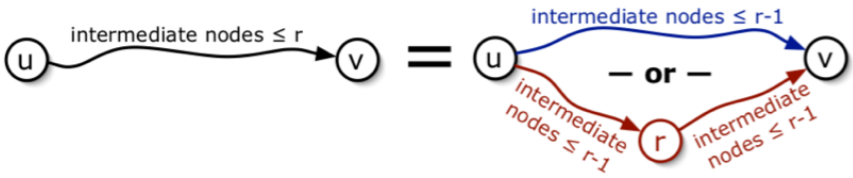
- Homework 4 is due Tuesday April 9.

Floyd-Warshall

- Last time, we started discussing all-pairs shortest paths: Given a directed graph G = (V, E, w) with possibly negative weights on edges, compute dist(u, v) for all pairs u and v.
- We tried using this dynamic programming formulation, but it leads to infinite loops:

$$dist(u,v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{x \to v} \big(dist(u,x) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

- There's a different dynamic programming formulation discovered (in some form) by many people independently as usual.
- We still use a third parameter, but now we won't track how many edges appear in a path, but instead track *which vertices* are allowed to appear in the path.
- Number the vertices arbitrarily from 1 to V.
- pi(u, v, r) := the shortest path from u to v where every intermediate vertex (that is, every vertex except u and v) is numbered at most r
- If r = 0, we can't have any intermediate vertices. So either pi(u, v, 0) = u ➔ v or it's not defined.
- Now, either pi(u, v, r) uses intermediate vertex r or it doesn't.



- If it does, it contains a subpath from u to r and a subpath from r to v so pi(u, v, r) = pi(u, r, r - 1) . pi(r, v, r - 1).
- If it doesn't, then pi(u, v, r) = pi(u, v, r - 1).
- So now, let dist(u, v, r) be the length of pi(u, v, r).

$$dist(u,v,r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min \begin{cases} dist(u,v,r-1) \\ dist(u,r,r-1) + dist(r,v,r-1) \end{cases} & \text{otherwise} \end{cases}$$

- The shortest path from u to v may use any vertex, so dist(u, v) = dist(u, v, |V|).

- We need to compute V x V x V = Theta(V^3) values, but it takes only constant time for each. So the algorithm will take Theta(V^3) time.

```
KLEENEAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r − 1] < dist[u, r, r − 1] + dist[r, v, r − 1]
                  dist[u, v, r] ← dist[u, v, r − 1]
              else
                  dist[u, v, r] ← dist[u, r, r − 1] + dist[r, v, r − 1]
```

- Jeff calls this algorithm Kleene (clay knee), because Kleene discovered this recursive pattern while studying finite automata.
- We can clean this algorithm up a bit. We only need to maintain the shortest paths from each u to v we've found so far, not which specific vertices they were allowed to go through.
- We also don't need to keep track of the specific vertex numbers as long a we loop through all the vertices.

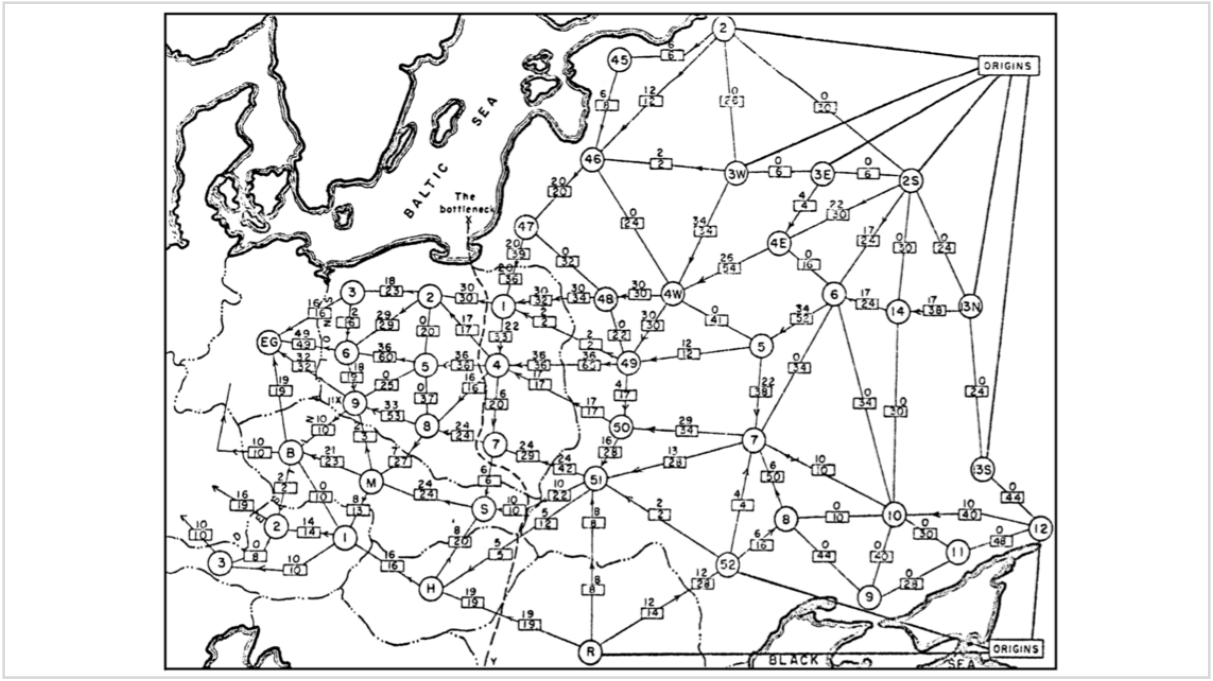
```
FLOYDWARSHALL(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

- This is the cleaned up version of that dynamic programming algorithm. It's usually referred to as Floyd-Warshall. A formal proof of correctness involves a similar induction proof to the one we used for Bellman-Ford, so I'll spare you the details.

Shipment Rates and Bottlenecks

- Now, let's get started on one last subject in graph algorithms, and the one I think is the most interesting.
- "In the mid-1950s, Air Force researcher Theodore E. Harris and retired army general Frank S. Ross published a classified report studying the rail network that linked the Soviet Union to its satellite countries in Eastern Europe. The network was modeled as a graph with 44 vertices, representing geographic regions, and 105 edges, representing links between those regions in the rail network.



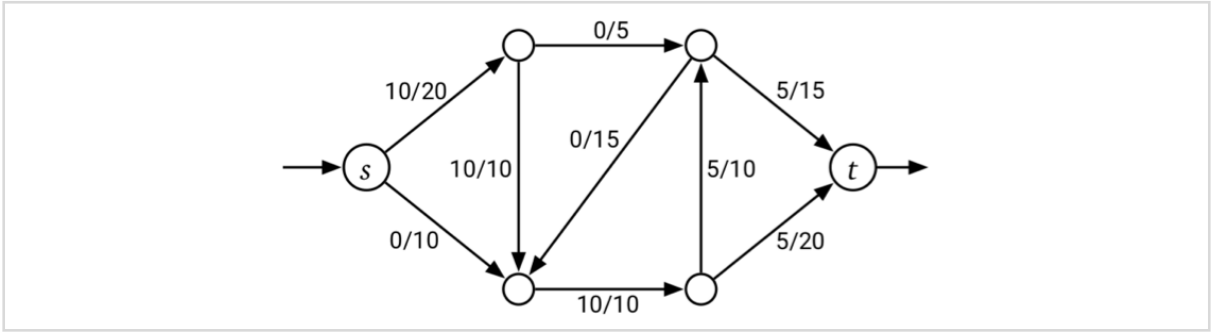
Each edge was given a weight, representing the rate at which material could be shipped from one region to the next. Essentially by trial and error, they determined both the maximum amount of stuff that could be moved from Russia into Europe, as well as the cheapest way to disrupt the network by removing links (or in less abstract terms, blowing up train tracks), which they called 'the bottleneck'. Their results, including the drawing of the network [above], were only declassified in 1999." — Erickson

- We're going to talk about how *not* to do these two things by trial and error.
- Specifically, we're going to discuss two problems known as the *maximum flow* problem, and the *minimum cut* problem.
- For both problems, we're given a directed graph G = (V, E) with special vertices s, the *source*, and t, the *target* or *sink*.
- The maximum flow measures how much material can be transported from s to t.
- The minimum cut measures how much damage we need to do to separate s from t.

Maximum Flow

- An *(s, t)-flow* is a way of assigning values to the edges that models how material flows through a network. You could also imagine the network as a series of tubes or pipes. We're measuring how fast water moves through them.
- Formally, its a function f : E ➔ R≥0 that satisfies the *conservation constraint* at every vertex v expect maybe s and t:
  - sum_u f(u ➔ v) = sum_w f(v ➔ w)
  - In other words, flow into v must equal flow out.
  - Here I'm using the convention that f(u ➔ v) = 0 if there is no edge u ➔ v.
- |f| is the *value* of the flow f. It is the net flow *out of* vertex s.
  - |f| := sum_w f(s ➔ w) - sum_u f(u ➔ s)

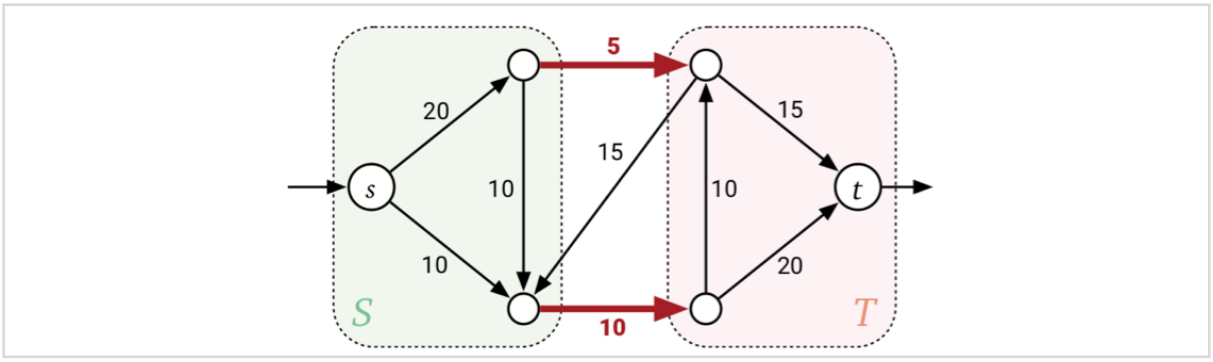
- It turns out the value of f is also equal to the net flow *into* t.
  - Define partial f(v) := sum_w f(v ➔ w) - sum_u f(u ➔ v), the net flow out of vertex v.
  - partial f(v) = 0 for all vertices v not equal to s or t, so
    - sum_v partial f(v) = partial f(s) + partial f(t)
  - But every edge leaves one vertex and enters another, meaning the sum of the net flows out of vertices must equal 0.
  - So sum_v partial f(v) = 0, implying partial(t) = -partial(s) = |f|
- OK, so the name of the problem implies we want to maximize the flow from s to t. So we need some limit on how much flow we'll send through an edge.
- We'll use a *capacity* function c : E ➔ R≥0 where c(e) is a non-negative capacity for an edge. Think of it as the width of the pipe.
- Flow f is *feasible* with respect to c if f(e) ≤ c(e) for every edge e.
- f *saturates* edge e if f(e) = c(e) and *avoids* e if f(e) = 0.
- Here's an example of a feasible (s, t)-flow of value 10.



- The *maximum flow problem* is to compute a maximum value (s, t)-flow that is feasible with respect to c.
- We'll eventually get to algorithms for this problem, but first let's talk about minimum cuts.

Minimum Cut

- An *(s, t)-cut* is a partition of the vertices into disjoint subsets S and T, meaning S U T = V and S intersect T = empty, where s in S and t in T.
- Again, we'll work with a capacity function c : E ➔ R≥0. The *capacity* of a cut (S, T) is the sum of capacities for edges that start in S and end in T.
  - ||S, T|| := sum_{v in S} sum_{w in T} c(v ➔ w)
  - Again, I'm being lazy here and just assuming c(v ➔ w) = 0 if v ➔ w is not in the graph.
- This definition is asymmetric. Edges that start in T and end in S don't matter at all when defining the capacity of the cut.
- Here's an example of an (s, t)-cut of capacity 15. Yes, 15. That backwards edge does not count.



- The *minimum cut problem* is to compute an (s, t)-cut with minimum capacity.
- One way to think about the problem is that the minimum (s, t)-cut is the cheapest way to disrupt all flow from s to t. And we can make that relationship formal.
- Lemma: The value of *any* feasible (s, t)-flow f is at most the capacity of *any* (s, t)-cut (S, T).

$$
\begin{aligned}
|f| &= \partial f(s) && \text{[by definition]} \\
&= \sum_{v \in S} \partial f(v) && \text{[conservation constraint]} \\
&= \sum_{v \in S} \sum_{w} f(v \to w) - \sum_{v \in S} \sum_{u} f(u \to v) && \text{[math, definition of } \partial \text{]} \\
&= \sum_{v \in S} \sum_{w \notin S} f(v \to w) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) && \text{[removing edges from } S \text{ to } S \text{]} \\
&= \sum_{v \in S} \sum_{w \in T} f(v \to w) - \sum_{v \in S} \sum_{u \in T} f(u \to v) && \text{[definition of cut]} \\
&\leq \sum_{v \in S} \sum_{w \in T} f(v \to w) && \text{[because } f(u \to v) \geq 0 \text{]} \\
&\leq \sum_{v \in S} \sum_{w \in T} c(v \to w) && \text{[because } f(v \to w) \leq c(v \to w) \text{]} \\
&= \|S, T\| && \text{[by definition]}
\end{aligned}
$$

- Now, look at the two inequality lines. The first is an equality if and only if there is no flow going from T to S. The second is an equality if and only if the flow saturates every edge from S to T.
- In other words: |f| = ||S, T|| if and only if f saturates every edge from S to T and avoids every edge from T to S.
- Therefore, if we have a flow f and cut (S, T) that satisfies this equality condition, f must be a maximum flow and (S, T) must be a minimum cut.

The Maxflow Mincut Theorem

- The surprising thing, and the thing most algorithms for this problem rely upon, is that the

value of the maximum flow is always *equal* to the capacity of the minimum cut.

- This was shown by Ford and Fulkerson in 1954 and independently by Elias, Feinstein, and Shannon in 1956.
- The Maxflow Mincut Theorem (Ford-Fulkerson): In any flow network with source s and target t, the value of a maximum (s, t)-flow is equal to the capacity of a minimum (s, t)-cut.
- We'll prove this on Tuesday and discuss efficient algorithms for finding maximum flows and minimum cuts.