CS 7301.003.20F Lecture 11—September 23, 2020

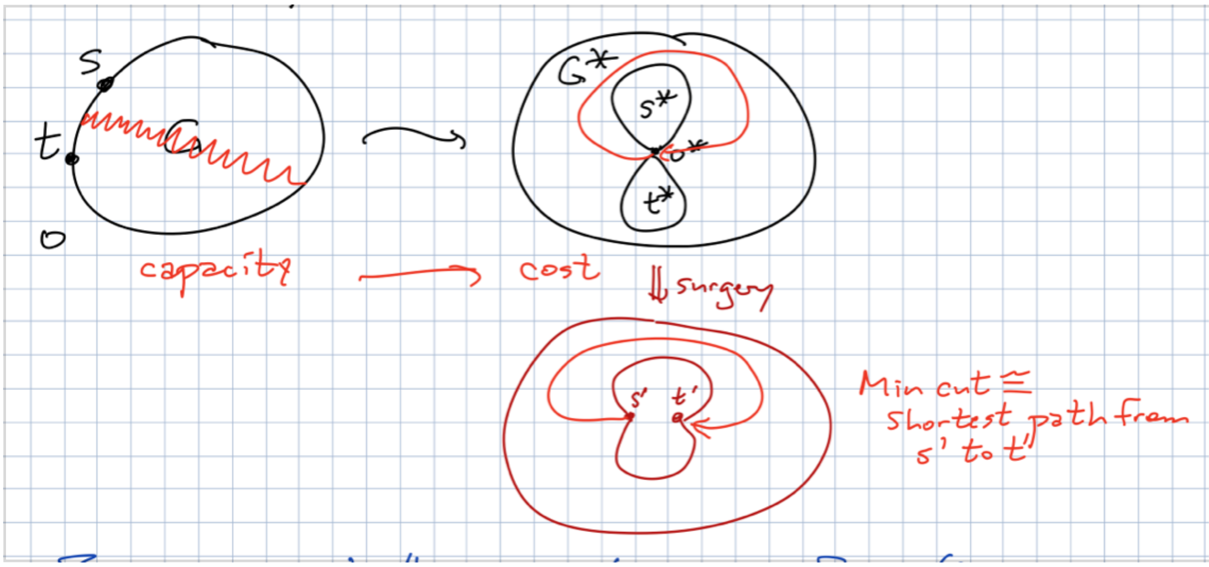
Main topics are #maximum_flow in #planar_graphs.

Maximum Flow

- I checked the calendar, and it looks like there's time for one more topic before we move on to surface embedded graphs: we're going to consider the maximum flow problem in planar graphs.
- First, some definitions. These may seem a bit different from what you're used to, but it makes the math work out so beautifully in the end. I'm going to show you an "antisymmetric flow" formulation.
- Take an abstract graph G = (V, D).
- A *flow* is a function phi : D ➡ R from darts to the reals. FLOWS CAN BE NEGATIVE. In fact, they will be for about half the darts, because we need phi(d) = -phi(rev(d)) for any flow.
- Let s and t be two vertices. Function phi is an *s,t-flow* if sum_u phi(u ➡ v) = 0 for all v ≠ s, t. These equations are often called the *conservation constraints*.
- Let c : D ➡ R^+ be the *capacities* of the darts. The (s,t)-flow is *feasible* if phi(d) ≤ c(d) for all darts d. These inequalities are often called the *capacity constraints*.
- The *value* of an s,t-flow is sum_u phi(u ➡ t). Our goal is to quickly find an s,t-flow of maximum value given a graph with non-negative capacities.
- Given an s,t-cut (S, T), its *capacity* is sum_{u ➡ v : u in S, v in T} c(u ➡ v). Ford and Fulkerson proved the value of any s,t-flow is at most the capacity of any s,t-cut, and this is actually an equality for the maximum value flow and minimum capacity cut.
- Orlin's ['13] algorithm can find both in O(n^2 / log n) time in an arbitrary planar graph, but (surprise!) we're going to do better today and Monday.

s,-planar Graphs

- Let's start with an easier case, where s and t lie on the outer face o. The following algorithm is by Hassin ['81].
- The minimum s,t-cut is a (directed) bond with two darts on the outer face, so its dual is a directed cycle through o^*. In this figure, it goes clockwise.



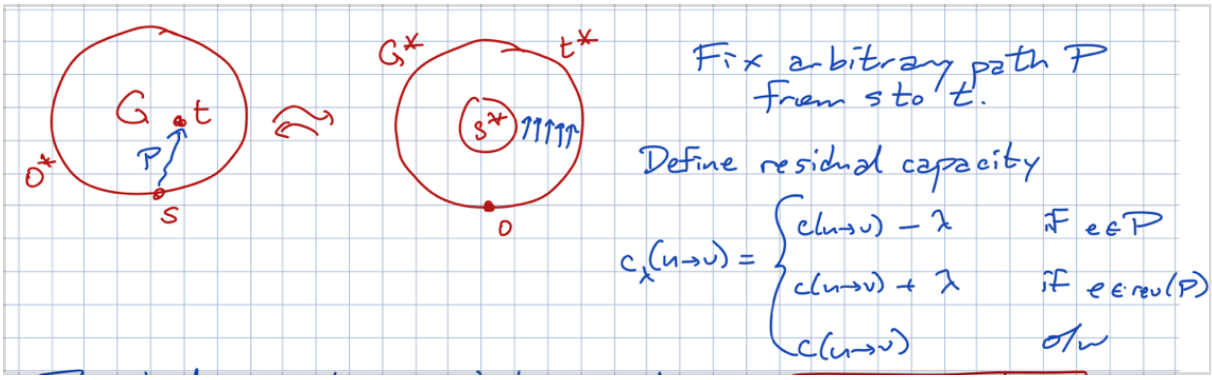
- s^* and t^* are both incident to o^*. Imagine slicing through o^ * to create two dual vertices s' and t'. That shortest cycle through o^ * is now a shortest path from s' to t'. So run Dijkstra's algorithm once to find it.
- We now know the minimum cut and its capacity, but can we find the maximum flow function itself?
- For each dual vertex f^*, let dist(f^ *) denote the distance from s'.
- For each dart u → v, define phi(u → v)
    - := dist(left(u → v)) - dist(right(u → v))
    - = dist(y) - dist(x)   [where (u → v)^* = x → y]
    - = c(x → y) - slack(x → y)   [by definition of slack]
    - ≤ c(x → y)
- We satisfied capacity constraints.
- If you go around any face of G^* other than s^* and t^* summing flows, the + dist(y)s and -dist(u)s cancel, leading to 0 net flow. In G, this means we satisfied conservation constraints.
- By running Henzinger et al. shortest paths in the dual graph, we can compute phi in only O(n) time.

Maximum Flows and Dual Shortest Paths

- So what if s and t are on different faces? We'll assume t lies on the outer face o^*.
- Venkatesan ['87] had an interesting observation. Fix a value lambda > 0. Can we find a feasible s,t-flow of value lambda if one exists?
- Let P be an arbitrary path from s to t. Let pi(d) =
    - 1 if d in P
    - -1 if d in rev(P)
    - 0 otherwise
- Imagine sending lambda units of flow from s to t along P, ignoring the capacities. Now

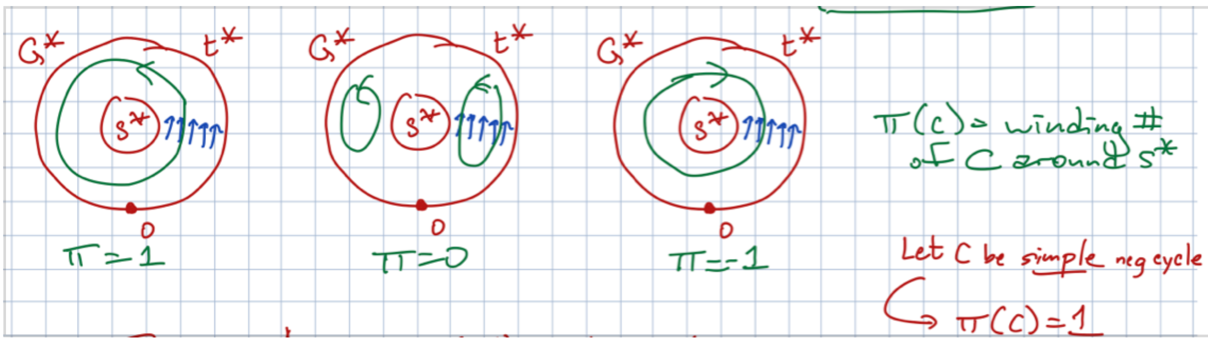
define the residual capacity of a dart as c_lambda(u ➜ v) = c(u ➜ v) - lambda * pi(u ➜ v).

- Intuitively, the residual capacities are telling you how much more (or less!) flow can go through each dart to have a feasible flow. RESIDUAL CAPACITIES CAN BE NEGATIVE! It just means you need to reduce the flow on the dart to make the flow feasible.
- Now imagine the residual capacities as edge lengths in the dual G^*. We refer to the combination of G^ * and c_lambda as the *dual residual network* G^ *_lambda. Let dist_lambda(f^ *) be the distance from o to f^ * in G^ * with regard to c_lambda.



(The figure on the left should have t and s swapped.)

- These distances are well-defined if and only if there are no negative cycles wrt c_lambda.
- Suppose there is a negative cycle C in G^*.
    - c_lambda(C) = sum_{d in C} (c(d) - lambda * pi(d)) < 0
    - But sum_{d in C} c(d) ≥ 0 and lambda ≥ 0, implying pi(C) ≥ 0.
    - C goes around s^* exactly once, so pi(C) = 1.



(These orientations seem to be backwards.)

- Meaning C goes clockwise around s^*. It's dual to an s,t-cut!
- Also, sum_{d in C^*} < lambda, implying lambda > mincut = maxflow.
- But what if dist_lambda is well-defined?
    - Define slack_lambda(p ➜ q) := dist_lambda(p) + c_lambda(p ➜ q) - dist_lambda(q) ≥ 0
    - Define phi_lambda(p ➜ q)
        - := dist_lambda(q) - dist_lambda(p) + lambda * pi(p ➜ q)
        - = c(p ➜ q) - slack_lambda(p ➜ q)
    - So phi_lambda(p ➜ q) ≤ c(p ➜ q).
    - And similar to before, sum_u phi(u ➜ v)
        - = sum_u lambda * pi(u ➜ v) =

        - lambda if v = t
        - -lambda if v = s
        - 0 otherwise
    - So phi_lambda is a feasible s,t-flow of value lambda.

Parametric Shortest Paths

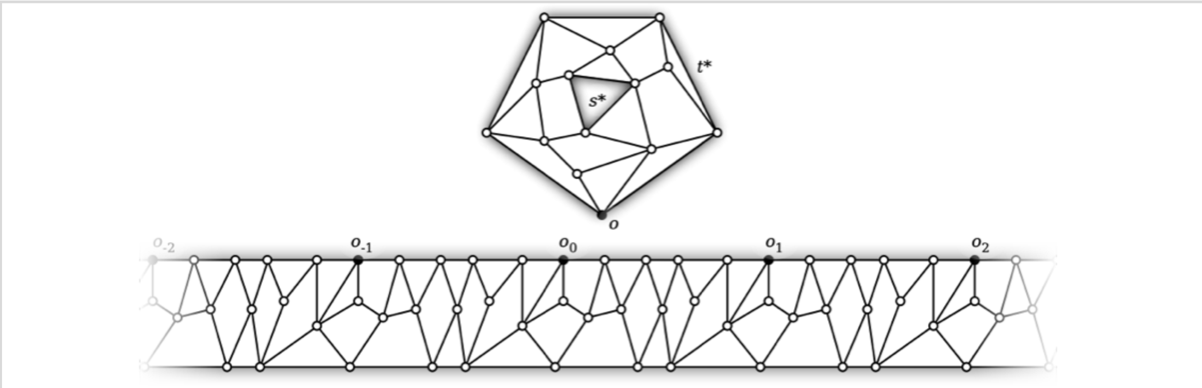
- But how do we find the maximum good value for lambda?
- We'll use a strategy similar to Monday's. We'll start with lambda = 0 and compute the dual shortest path tree. Then we'll continuously increase lambda as we compute so-called *parametric shortest paths*.
- dist_lambda and phi_lambda will vary continuously as we increase lambda, but shortest path tree T_lambda will change at discrete *pivots* just like before.
- And just like before, we'll wait until just before some slack_lambda(p → q) < 0, pivot p → q into T_lambda, and pivot x → q out.
- slack'_lambda(p → q) := derivative of slack for p → q and path_lambda(p) := the shortest path in T_lambda to p
- slack'_lambda(p → q)
    - = dist'_lambda(p) + pi(p → q) - dist'_lambda(q)
    - = -pi(path_lambda(p)) - pi(p → q) - pi(rev(path_lambda(q)))
    - = -pi(cycle(T_lambda, p → q))
    - in {-1, 0, 1}
- Also, we see slack'_lambda(p → q) = -slack'_lambda(rev(p → q))
- We'll call p → q *active* if slack'(p → q) = -1.
- But how do we find the active darts?
- Let L_lambda = (G \ T_lambda)^*, the complementary spanning tree of G.
- Assuming shortest paths are unique, L_lambda is the set of *loose* edges, those where both darts have slack > 0.
- LP_lambda := unique path from s to t in L_lambda
- Lemma: d^* is active if and only if d in LP_lambda
    - Darts of T_lambda have slack = 0, so slack' = 0 for them and their reversals.
    - d^* is active if and only if pi(cycle(T_lambda, d^*)) = 1 if and only if C(d) = (cycle(T_lambda, d^ *))^ * is an s,t-cut
    - If d^* is active, then LP_lambda contains at least one edge of C(d). But d is on the only loose edge of C(d), so d in LP_lambda
    - If d in LP_lambda, then C(d) is an s,t-cut.
- So now the algorithm behaves similar to MSSP.
- As we increase lambda, slack_lambda(d) decreases for all d in LP_lambda and increases for all d in rev(LP_lambda).

        - Equivalently, phi_lambda(d) increases along LP_lambda and decreases along rev(LP_lambda).
        - It's like we're pushing flow from s to t along LP_lambda.
- Pivot d into T_lambda when slack_lambda(d) = 0. It's like we saturated the dart.
- Pivot pred(q) → q out at the same time. It's like we made a new augmenting path!
- Eventually, we do a pivot that creates a directed cycle of slack 0 darts.
- But that means each of those darts is saturated. We found the minimum s,t-cut and the maximum s,t-flow!
- Pseudocode:
    - Create initial dual shortest path tree T_0
    - Maintain primal spanning tree L
    - While L is connected
        - LP ← path from s to t in L
        - (p → q) ← min slack edge in LP^*
        - decrease slacks along LP by slack(p → q). Increase slacks along rev(LP) by the same amount.
        - delete (p → q)^* from L
        - insert (pred(q) → q)^* into L
        - pred(q) ← p
    - phi ← c - slack

Analysis

- Using dynamic forests, each step of the algorithm can be implemented in O(log n) time. Our running time therefore depends on the number of pivots.
- Let path_i denote the shortest walk from o to q s.t. pi(path_i(q)) = i.
- We see dist_lambda(q) = min_i (c_lambda(path_i(q)).
- We can also observe that whenever path_lambda(q) changes, pi(path_lambda(q)) increases by 1, because pi(cycle(T, p → q)) = 1 if p → q is pivoting in.
- So now we want to know, for which i is p → q in path_i(q)?
- Imagine removing faces s^* and t^* from the plane. We now have a sphere with two boundary, also known as an *annulus*.
- We'll define something called the *universal cover* as follows: Imagine cutting along that path P from earlier, turning our annulus into a disk. Now make a doubly infinite sequence …, G^ *_-1, G^ * _0, G^ *_1, G^ *_2, … of copies of G^ * and paste them together along them respective copies of P.
- Formally, its a plane graph Gbar* = (Vbar*, Ebar *) where Vbar * = {p_i | p in V * and i in Z} and Ebar * = {p_i → q_{i + pi(p → q)} | p → q in E *}. We also have dart costs c(p_i → q_j) = c(p → q).

- Each vertex p_i is a *lift* of p to Gbar *. There is a *projection map* omega bar(Gbar * → G *) that drops subscripts so omega bar(p_i) = p and omega bar (p_i → q_j) = p → q.
- The preimage omega bar^-1(Pi) for any path Pi in G^* is a doubly-infinite set of paths in Gbar * called the *lifts* of Pi. If Pi starts at p and ends at q, then for any i, there is a lift of Pi from p_i to q_{i + pi(Pi)}.
- s^* and t^* lift to two unbounded faces sbar * and tbar *, and every other face lists to an infinite sequence of faces.



- So now for the punchline. Each path_i(q) in G^* lifts to a shortest path to q_0 from o_{-i}.
- The i for which p → q is in path_i(q) are the set of i for which the shortest path from o_{-i} goes through p_{-pi(p → q)} → q_0.
- But we saw on Monday that these i are contiguous!
- So p → q pivots into T_lambda at most once, and it leaves at most once.
- Which implies O(n) pivots.
- Which implies an O(n log n) running time!
- Some quick notes:
  - Erickson ['10] described this formulation of the algorithm based on parametric shortest paths.
  - His algorithm is essentially identical to one by Borradaile and Klein ['09]. However, they describe things mostly in the primal graph, sending flow along LP each iteration to always have a "leftmost flow" of each value lambda. Their analysis is much more complicated, because they focus on how often you can saturate each dart with this primal flow.