CS 7301.003.20F Lecture 10—September 21, 2020

Main topics are #multiple-source_shortest_paths.

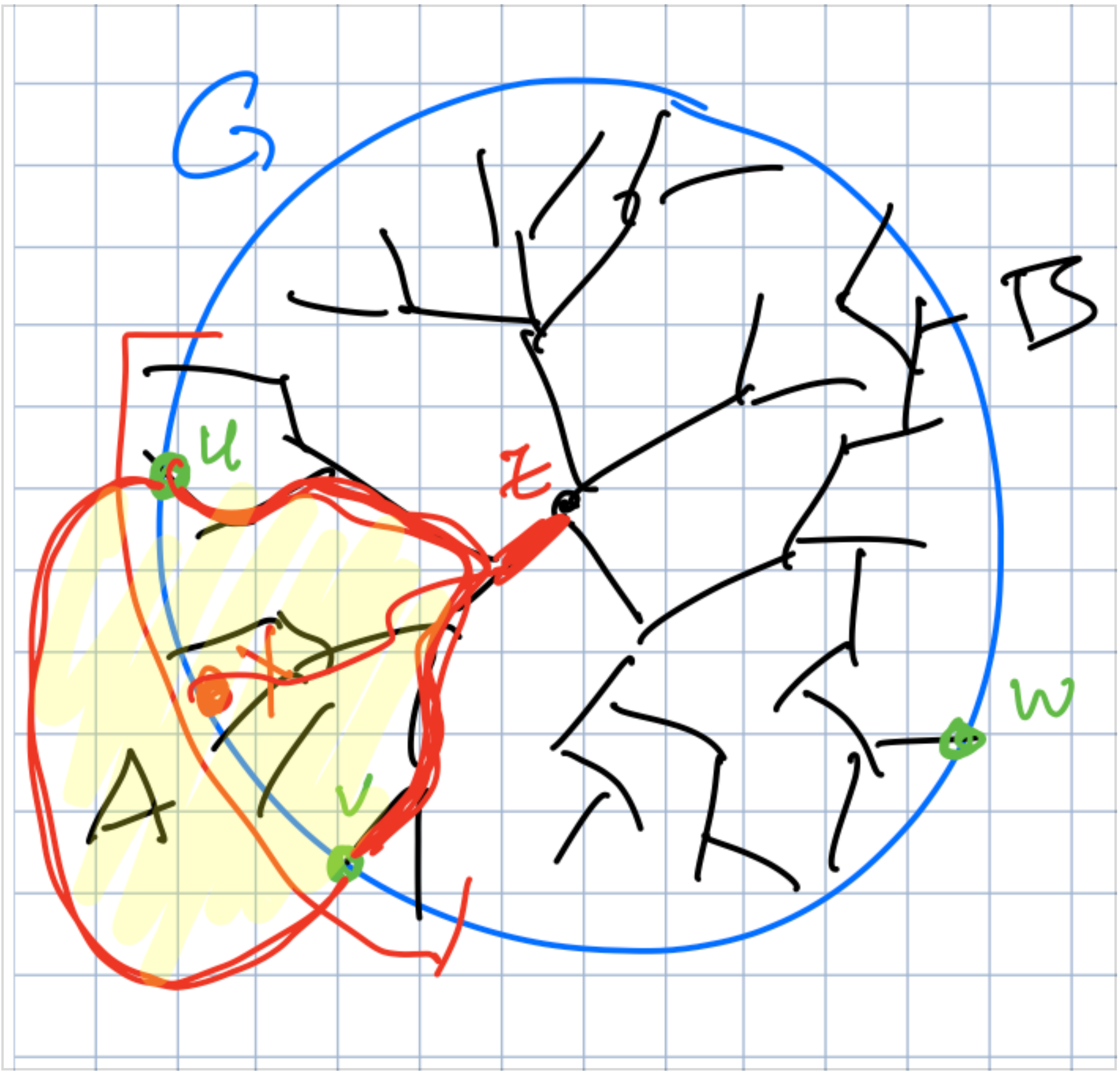
Multiple-Source Shortest Paths

- At the end of the last lecture, I mentioned an algorithm that uses dense distance graphs (DDGs). We're given an r-division of the edges where every piece has at most r edges and O(sqrt{r}) boundary vertices lying on O(1) faces. We need to compute shortest paths within each piece between its boundary vertices, and we need to do so quickly.
- Today, we're going to look at (a modified) algorithm by Klein ['05] that computes so-called *multiple-source shortest paths*.
- Given a directed(!) plane graph G = (V, E) with outer face o and a non-negative weight function w : D ➔ R^+, we want to compute all shortest paths rooted at every boundary vertex of o.
- If o has k vertices, then we could run Dijkstra once per vertex for O(nk log n) time total.
- But Klein claims O(n log n) time!
- Of course, finding all shortest paths this fast is impossible if k = omega(log n). We need to be more restrictive on what we accomplish.
  - If we ask for p source-destination pairs in advance, then we can find all their distances in O(n log n + p log n) time.
  - Or we can build a data structure in O(n log n) time using O(n log n) space that answers distance queries in O(log n) time per query.
  - Or we can find the shortest paths in O(1) time per edge if we know the paths in advance or O(log Delta) time per edge using the data structure if the graph has max degree Delta.

Trees and Disks

- Our algorithm is going to compute a shortest path tree T from some vertex on o in O(n log n) time. Then, we're going to move the source vertex-by-vertex around o.
- Every time we move to the next vertex, some edges leave T and are replaced by other edges. The big surprise that makes the result possible is that only O(n) edges leave and enter the tree during one pass around o.
- The small number of changes is largely do to the following observation.
- Pick *any* spanning tree T of G and an edge e in T.
- T \ e has 2 component trees. Label them A and B.
- Lemma: The set of outer face vertices in A is either empty, includes all vertices of o, or it induces a consecutive path of vertices in o.

- Suppose u, v, and w are boundary vertices such that u, v in A and w not in A.
- Let z be the least common ancestor of u and v after rooting T on an endpoint of e.
- The path from z to u along T, from u to v along a path just outside o, and from v to z along T bounds a disk.
- For any x between u and v on o, the path from x to e in T begins inside the disk and doesn't leave until hitting z. So x in A.



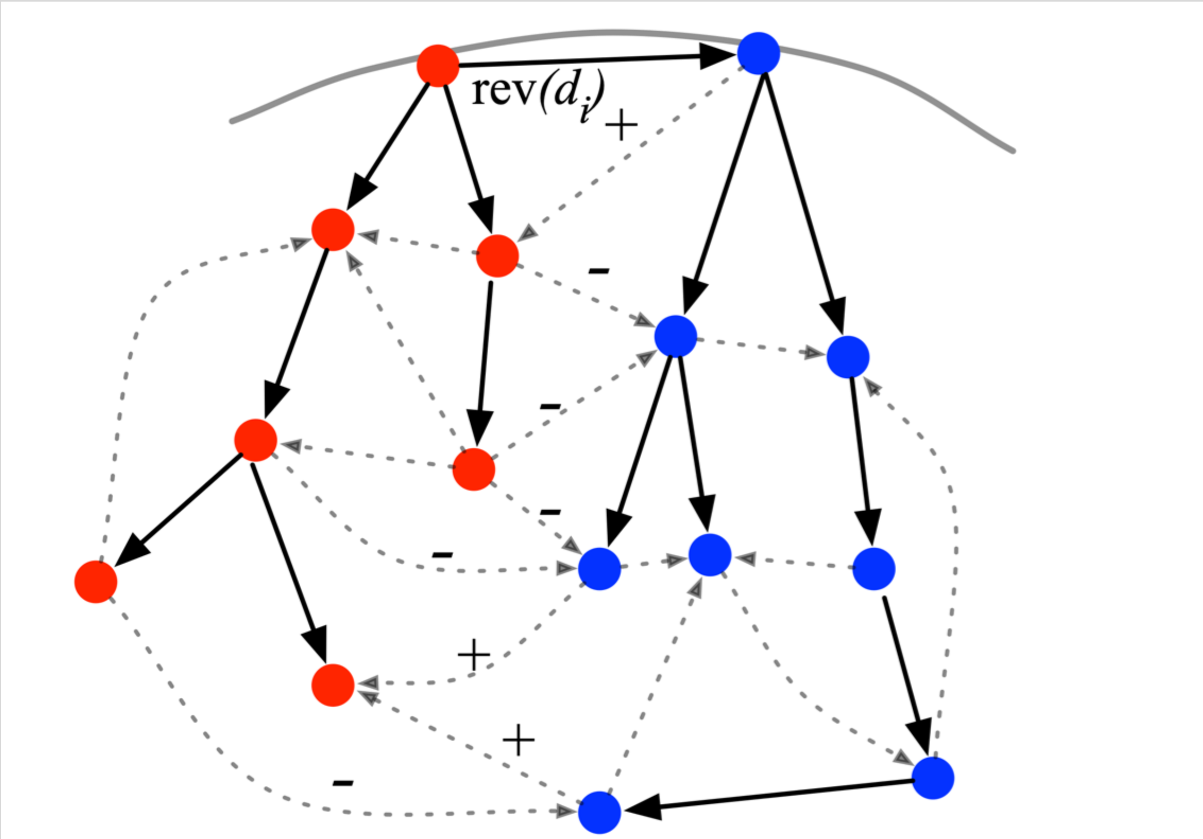
- Let s_0, s_1, … be the vertices on o and T_0, T_1, … be their shortest path trees.
- Pick any dart u ➔ v. Which trees T_i contain u ➔ v?
- Let T_v be a tree of shortest paths into v.
- Assuming each T_i follows the same path s_i, v-path as in T_v, the set of T_i containing u ➔ v are those for which s_i lies in the u side of T_v \ uv.
- But we just saw those s_i are contiguous, so u ➔ v will enter the shortest path tree at most once and leave it at most once.
- There are O(n) darts, so the shortest path tree undergoes O(n) changes total.
- We'll assume there is exactly one shortest path between any pair of vertices so that this analysis holds. There are ways to enforce this.

Changing Sources

- Suppose we've computed T := T_{i - 1} and we want to compute T_i. There's a couple ways we could do this that all use about the same sequence of computations.
- I'm going to describe a strategy by Eisenstat and Klein ['13].
- We're going to incrementally transform T into T_i using a sequence of *pivots* where one dart leaves T, and another takes its place. After all the pivots are complete, we'll have T_i.
- First, we do the *special pivot*. We remove the dart going into s_i and replace it with dart s_i → s_{i - 1}. We now have s_i as the root of T.
- Doing special pivot with no other changes means T may not be a shortest path tree. Therefore, we will temporarily assign s_i → s_{i - 1} a new weight of lambda := -dist(s_{i - 1}, s_i).
- Define the *slack* of a dart as slack(u → v) := dist(u) + w(u → v) - dist(v) (where dist uses distance from the current shortest paths source vertex).
- Ford showed slack ≥ 0 for all edges of G, and slack = 0 for all edges appearing on shortest paths. Since we're assuming shortest paths are unique, the slack is 0 if and only if the edge is in the shortest paths tree.
- Claim: Immediately after the special pivot and reweighting of s_i → s_{i - 1}, T is a shortest path tree rooted at s_i.
    - I claim the distance to any vertex x dropped by dist(s_{i - 1}, s_i) when we did the special pivot and reweighted s_{i - 1} → s_i.
    - If the new shortest path to x uses s_{i - 1}, then we have prepended the old path with dart s_{i - 1} → s_i.
    - If the new shortest path doesn't use s_{i - 1}, then the old path from s_{i - 1} to s_i to x just had its prefix from s_{i - 1} to s_i removed.
    - Other than s_{i - 1} → s_i, no weights changed, so all other slacks are the same. Finally, slack(s_{i - 1} → s_i) = 0 by our choice of lambda.
    - So T is a shortest path tree rooted at s_i, but only because s_i → s_{i - 1} has a new weight.
- We need to recover the original weight, so we'll "continuously" increase lambda, the current weight of s_i → s_{i - 1}, until lambda = w(s_i → s_{i - 1}), the original weight of the dart.
- What happens as we increase lambda? T \ (s_{i - 1} s_i) has two components. Those in the s_i component retain their distance from s_i. Color these vertices red. Those in the s_{i - 1} component have their distances from s_i increase at the same rate lambda is increasing. Color those blue. (Here, I'm using Klein and Mozes figures/colors. Erickson uses the opposite colors.)
- Meanwhile, what is happening to the slacks? blue → blue and red → red darts don't

change slack. blue → red darts have their slack increase at the same rate lambda is increasing.

- But red → blue darts (except for s_{i - 1} → s_i) have their slacks decrease at the same rate lambda increases.



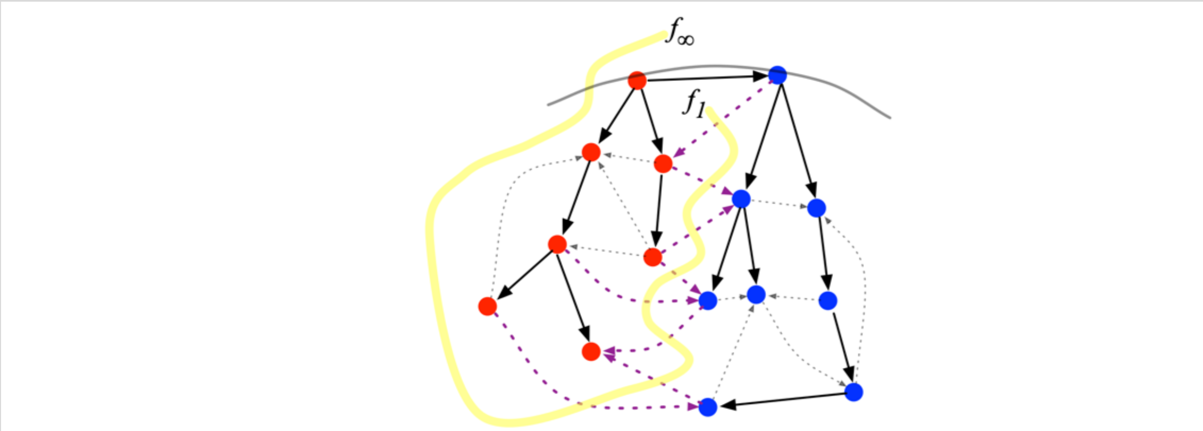
(Here, rev(d_i) = s_i → s_{i - 1}.)

- Eventually, one of the red → blue darts x → y hits slack 0 and is about to go negative. Let z → y be the predecessor dart of y in T. We can now pivot z → y out of T, replacing it with x → y. And T is still a shortest path tree!
- z → y was blue → blue but is now blue → red. So its 0 slack is starts to increase. We may now continue increasing lambda.
- After a sequence of pivots, lambda eventually reaches w(s_i → s_{i - 1}) and we've computed T_i.

Fast Pivots

- We still need a fast way to find each edge x → y that pivots into T and perform the pivot.
- T \ (s_{i - 1} s_i) has red and blue components. The red-blue edges form an edge cut. In particular, they form a bond.
- The dual of this bond is a cycle.
- Let C := (G \ T)^* be the complementary spanning tree. The dual cycle of red-blue edges has exactly one edge outside of C, namely s_{i - 1} s_i. The red → blue darts with decreasing slack form a path in C from right(s_{i - 1} → s_i)^* to left(s_{i - 1} → s_i)^*. The

darts with increasing slack form the reversal of that path.



(Here, the path goes from f_1 to f_{infty}.)

- At this point, I'm going to punt and use a couple data structures as black boxes. Describing how they work would take another week.
- A *dynamic forest* data structure maintains a changing collection of darts between a fixed set of vertices. We need to guarantee that the edges always form a forest.
- We can do the following operations (and more!) in O(log n) time per operation:
  - Link (insert) an edge between given vertices u and v.
  - Cut (delete) the edge between u and v.
  - Read or write a weight on a given vertex or dart.
  - Add a value to the weight of all darts in the unique path between given vertices u and v.
  - Add a value to the weight of all vertices in a subtree rooted at a given vertex u.
  - Find the minimum weight dart in a path.
- So, we store both T and C in separate dynamic forests. We store distances on vertices of T and slacks on darts of C. To find a normal pivot, we ask for the minimum slack Delta on the path in C.
- To do a pivot, we decrease the slacks on the path by Delta and increase the slacks on the reversal by Delta. We increase distances to all blue vertices by Delta. We do a pair of links and cuts and a pair of slack assignments to pivot the actual edges.
- So that's O(log n) time doing a constant number of dynamic forest operations each pivot. Earlier, we saw there are O(n) pivots across the whole algorithm, so we can find all the shortest path trees in O(n log n) time total.
- We can look up distances as we need them in O(log n) time per distance.
- And if we use something called *persistence*, we can remember how our data structure progressed over time so we can look up shortest path distances between vertices of o and other vertices later.