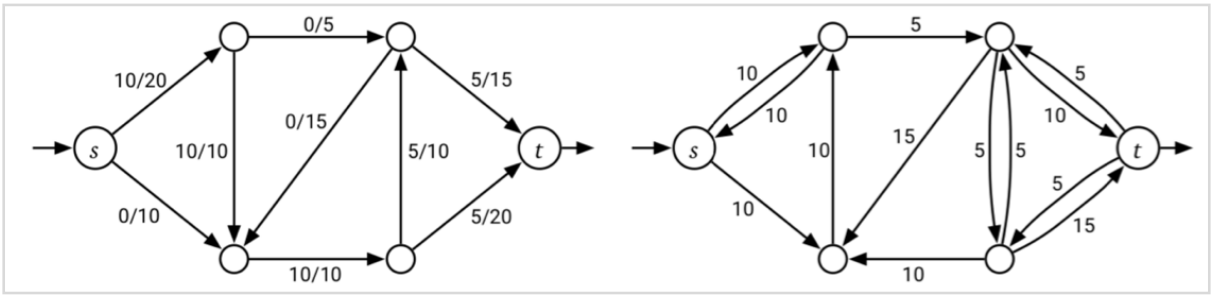
CS 6363.003.21S Lecture 19–April 13, 2021

Main topics are #maximum_flow and #minimum_cut.

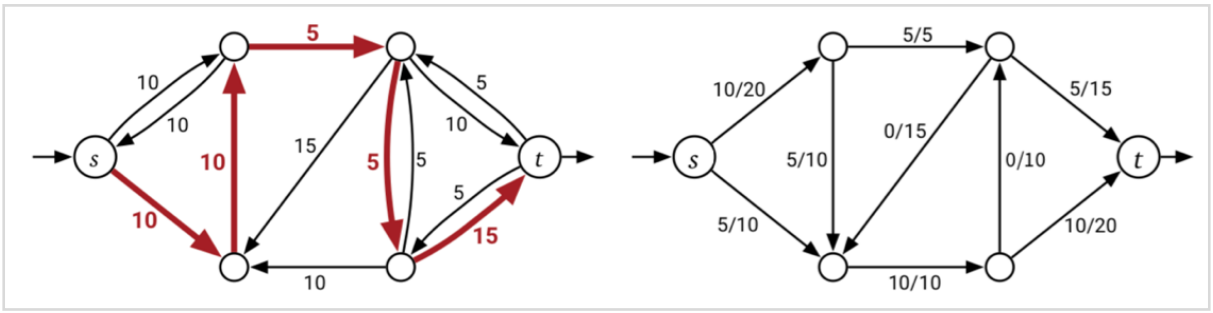
Ford-Fulkerson's Augmenting Path Algorithm

- Last week, we defined the maximum flow and minimum cut problems, and then we saw a proof that for any input to the two problems, the value of the maximum flow equals the capacity of the minimum cut.
- For the proof, we started with an arbitrary feasible (s,t)-flow f : E ➔ R_{≥0} and built the residual graph G_f which contained all edges with positive residual capacity according to c_f : V x V ➔ R_{≥0}:

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}$$

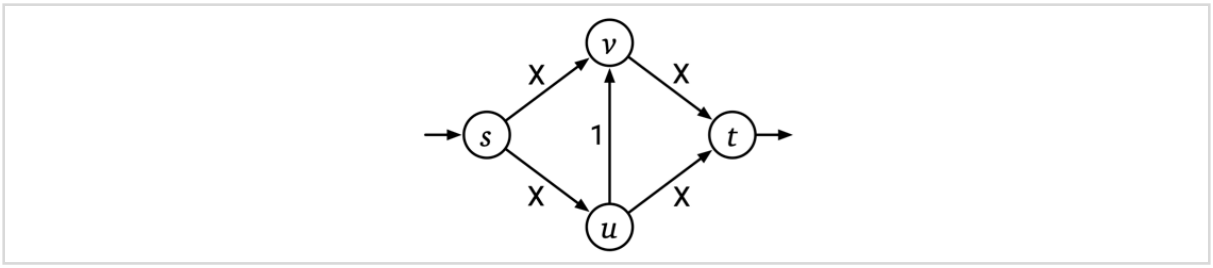


- If G_f contained an augmenting path P from s to t, we pushed a maximal amount of flow along P, creating a new flow of higher value.



- Otherwise, we set S to be the vertices reachable from s in G_f and T := V \ S. Pair (S, T) was a cut whose capacity equaled the value of f. So a max flow and a min cut.
- You can turn this proof into an algorithm for computing a maximum flow usually referred to as the Ford-Fulkerson Augmenting Path Algorithm.
- In short: start with all flow values equal to 0 and repeatedly push flow along augmenting paths until you can't find one anymore.
- But will this process actually result in a maximum flow?
- First, let's assume all the capacities are integers. This has a few repercussions.
  - The initial flow is all integers since 0 is an integer.

    - If we assume inductively that f is all integers, then all the residual capacities are integers.
    - Meaning any amount of flow we push is always a positive integer.
    - Meaning any new flow is all integers and its value is at least 1 greater than the old flow.
- So if we let f^* denote the maximum flow, we do at most | f^* | augmentations and f^* is all integers.
- We can build and search the residual graph in O(E) time, so these | f^* | augmentations take O(E | f^* |) time total.
- But there are two issues with this analysis.
- At the beginning of the semester, I talked about different classes of running times. Our usual goal was to find algorithms with running time polynomial in the input size. For example, we could compute edit distance in O(n^2) time or all pairs shortest paths in O(V^3) time.
- O(E | f^* |) is what we call a *pseudo-polynomial time* algorithm. It runs in time polynomial in |E| and | f^* |, but | f^* | may not be polynomial in the input size.
- In particular, consider the example below: we might try pushing along the 1 edge or its reverse every augmentation, leading to a running time of Theta(X). But X can be written using O(log X) bits; the running time is exponential in the input size!



- Ford-Fulkerson is often efficient in practice, though, or in situations where you can guarantee | f^* | is small.
- The other issue with this analysis is that we're assuming the capacities are integers. But we defined flows and capacities using real numbers.
- You can set up examples with real number capacities where every augmentation gets smaller and smaller and smaller. You always get higher value flows, but you never get to a maximum flow. There's not even a guarantee that you'll approach the maximum flow value in the limit.
- Of course, computers don't actually store real numbers, but you should still be nervous. If your floating point additions or comparisons start doing rounding, you may actually enter an infinite loop where you never make real progress on increasing the flow value!
- But here's the trick. We get to choose which augmenting paths to use. If we pick carefully, maybe the algorithm will run faster.
- Before we discuss our options, though, it might help to learn a bit more about how flows

are structured.

Combining and Decomposing Flows

- (s,t)-flows have some nice (algebraic) structure which we can use in our algorithms or applications. These are easiest to explain if we ignore capacities and allow flow values on edges to be negative.
- A first thing we might observe is that flows combine nicely. Let f and g be two (s,t)-flows, and let alpha and beta be two real numbers. Define a function h : E ➔ R as

$$h(u \to v) := \alpha \cdot f(u \to v) + \beta \cdot g(u \to v)$$

  or more simply, we might say h := alpha f + beta g.
- It's not hard to prove that h is also an (s,t)-flow. Further, its value |h| = alpha |f| + beta |g|. More generally, any linear combination of (s,t)-flows is an (s,t)-flow!
- But if we flip this idea around, we can write any given (s,t)-flow f as a weighted sum of particularly nice simple flows.
- Let P be any directed path from s to t (which may go along some edges of G backwards). The corresponding *path flow* P : E ➔ R is defined as (yes, I reused notation):

$$P(u \to v) = \begin{cases} 1 & \text{if } u \to v \in P, \\ -1 & \text{if } v \to u \in P, \\ 0 & \text{otherwise.} \end{cases}$$

- Similarly, given a directed cycle C, we can define a corresponding *cycle flow* C : E ➔ R as

$$C(u \to v) = \begin{cases} 1 & \text{if } u \to v \in C, \\ -1 & \text{if } v \to u \in C, \\ 0 & \text{otherwise.} \end{cases}$$

- As stated, any linear combination of path and cycle flows is an (s,t)-flow. But it turns out the converse is true, and in a nice way.
- Flow Decomposition Theorem: Every non-negative (s,t)-flow f is the positive linear combination of at most |E| directed (s,t)-path and cycle flows. Moreover, an edge u ➔ v appears in at least one of these paths or cycles if and only if f(u ➔ v) > 0.
- I'll spare you the details, but Erickson uses the following approach in his proof.
  - First, we prove the lemma only for a *circulation* meaning flow is conserved at *every* vertex including s and t.
  - To do so, we greedily walk along edges with positive flow until we repeat a vertex, finding a cycle C. Let F be the minimum flow value on any edge of the cycle. One term in our linear combination is F C.
  - Then, we subtract F C from the circulation, resulting in one fewer edge with positive flow. The Recursion Fairy handles the rest.

    - But what if f is not a circulation? Add an edge t ➔ s to the graph and set f(t ➔ s) := |f|. Now we do have a circulation, and we can use the above argument. Every cycle we use that contains t ➔ s is really a path in the original graph.
- This proof is practically an algorithm! You can find the decomposition in O(V) time per member, so O(VE) time total.
- We may also observe two things from this proof strategy that I've personally used many times in my career:
    - Circulations can be decomposed into a weighted sum of cycles; you don't need paths.
    - An *acyclic* (s,t)-flow (no cycles of positive flow edges) can (only) be decomposed into a weighted sum of paths.
- Further, you can take any (s,t)-flow, and repeatedly remove flow cycles as described above to find an acyclic flow of the same value. Sometimes acyclic flows are easier to use or play with depending on why you needed to compute the flow.
- So now that we know a bit more about the structure of flows, let's see about speeding up Ford-Fulkerson by more carefully selecting our augmenting paths.
- Both of the following algorithms were discovered by Edmonds and Karp (and others) in the 1970s.

Edmonds-Karp 1: Fattest Augmenting Paths

- Edmonds-Karp: Choose the augmenting path with the largest bottleneck value.
- You can find this path using a variant of the Prim-Jarník minimum spanning tree algorithm: Build a spanning tree from s in the residual graph, repeatedly adding edges of largest residual capacity that leave the tree.
- You can pull edges out of a priority queue implemented with, say, a binary heap, so O(log V) time per edge or O(E log V) time total to find each augmenting path.
- So how many augmenting paths are there?
- Let f be the current flow and f' be the maximum flow *in the current residual graph G_f*. In other words, f + f' is the maximum flow in G.
- Let e be the bottleneck edge in the current iteration, so we're about to push c_f(e) units of flow.
- There is a decomposition of f' that includes at most |E| path flows, so c_f(e) ≥ |f'| / |E|.
- So pushing down the maximum-bottleneck path multiplies the *residual* maximum flow value by (1 - 1 / |E|) or less.
- After |E| * ln |f^* | iterations, the residual value of the maximum flow is at most

$$|f^*| \cdot (1 - 1/E)^{E \cdot \ln |f^*|} < |f^*| e^{-\ln |f^*|} = 1.$$

- In other words, we can't do another augmentation after |E| * ln |f^* | iterations if the capacities are integers, because there won't be an integral amount of flow left to push.

- The total running time *assuming integer capacities* is O(E^2 log V log |f^* |).
- This running time *is* polynomial in the problem size, but it still relies on integer capacities.

Edmonds-Karp 2: Shortest Augmenting Paths

- Edmonds-Karp (again): Choose an augmenting path with the smallest number of edges.
- Can be found in O(E) time by running a breadth-first search in the residual graph.
- Now to bound the number of iterations.
- Let f_i be the flow after i iterations, and G_i = G_{f_i}. Flow f_0 is zero everywhere and G_0 = G.
- Let level_i(v) be the unweighted shortest path distance from s to v in G_i.
- Lemma: level_{i}(v) ≥ level_{i-1}(v) for all vertices v and non-negative integers i.
  - We'll do induction on level_i(v).
  - level_i(s) = 0 = level_{i-1}(s). Check.
  - If we cannot reach v from s, then level_i(v) = infty ≥ level_{i-1}(v). Check.
  - Otherwise, let s ➔ … ➔ u ➔ v be a shortest path to v in G_{i}.
  - level_i(v) = level_{i}(u) + 1, so the induction hypothesis shows level_i(u) ≥ level_{i-1}(u).
  - If u ➔ v is in G_{i-1}, then level_{i-1}(u) + 1 ≥ level_{i-1}(v).
  - If u ➔ v is not in G_{i-1}, then we must have pushed along v ➔ u to create residual capacity in u ➔ v. Meaning v ➔ u was on the shortest s to t path. So level_{i-1}(u) + 1 > level_{i-1}(u) -1 = level_{i-1}(v).
  - Either way, level_i(v) = level_i(u) + 1 ≥ level_{i-1}(u) + 1 ≥ level_{i-1}(v)
- Lemma: Any edge u ➔ v disappears from the residual graph at most |V| / 2 times.
  - Suppose u ➔ v is in G_i and G_{j+1} but not in G_{i+1}, …, G_j for some i < j.
  - u ➔ v must be in the ith augmenting path, so level_i(v) = level_i(u) + 1.
  - and v ➔ u must be in the jth augmenting path, so level_j(u) = level_j(v) + 1.
  - So, level_j(u) = level_j(v) + 1 ≥ level_i(v) + 1 = level_i(u) + 2.
  - So the distance from s to u increased by 2 between the disappearance and reappearance of u ➔ v. Every level is less than |V| or infinite (if there is no path to u), so an edge can disappear at most |V| / 2 times.
- There are 2|E| possible residual edges so |E| |V| disappearances total. Each augmentation makes its bottleneck edge disappear, so there are at most |E| |V| iterations.
- The total running time is O(VE^2).
- And this running time is correct even for arbitrary non-negative *real number* edge capacities.
- A variation on this idea was independently proposed by Dinitz in 1970. His algorithm was more complicated, but it runs in only O(V^2 E) time.
- And there have been many more algorithms discovered since these. Some rely on fancy data structures. Some use methods other than augmenting paths.

- Building upon decades of more or less steady progress from several researchers, Orlin in 2012 described an algorithm that runs in only O(VE) time.
- Very few people understand this algorithm, and I am not one of them, so it's well beyond the scope of this class.
- But for the purposes of doing homework or exams, you should feel free to cite it.
- Orlin [2012]: **Maximum flows and minimum cuts can be computed in O(VE) time.**