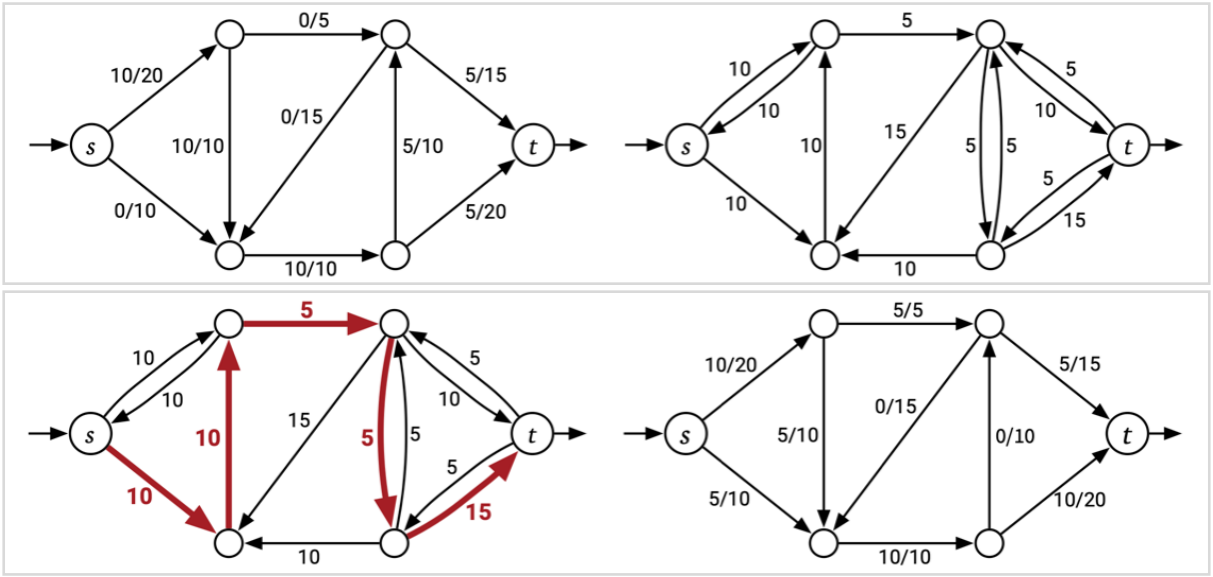
CS 4349.003.19F Lecture 21—November 6, 2019

Main topics for #lecture include #maximum_flow and #minimum_cut.

Ford-Fulkerson

- Last time, we discussed the maxflow mincut theorem that the value of a maximum flow is equal to the capacity of the minimum cut.
- In short, you can take any flow f.
- The residual capacity is the amount of additional flow you could send along an edge, if that edge was in the original graph, or the amount of flow you could undo from an edge if the reversal was in the original graph.
- You build the residual graph that contains all edges of positive residual capacity.
- If there is some augmenting path in the residual graph from s to t, we can push flow along that path, increasing the value of the flow.



- Otherwise, it means we have a maximum flow and the vertices reachable from s in the residual graph form one side of a minimum (s, t)-cut.
- You can turn this proof into a strategy for computing a maximum flow: start with all flow values equal to 0 and repeatedly push flow along augmenting paths until the flow is maximum.
- But will this process actually reach the 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 the flow we push is always a positive integer.
  - Meaning the 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 the residual graph in O(E) time, so these | f^* | augmentations take O(E | f^* |) time total.
- But there's 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 fact, we can write down capacities of size 2^X using only X bits, so the running time may actually be exponential in the input size.
- The algorithm 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 flows and capacities are still well-defined 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 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 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!

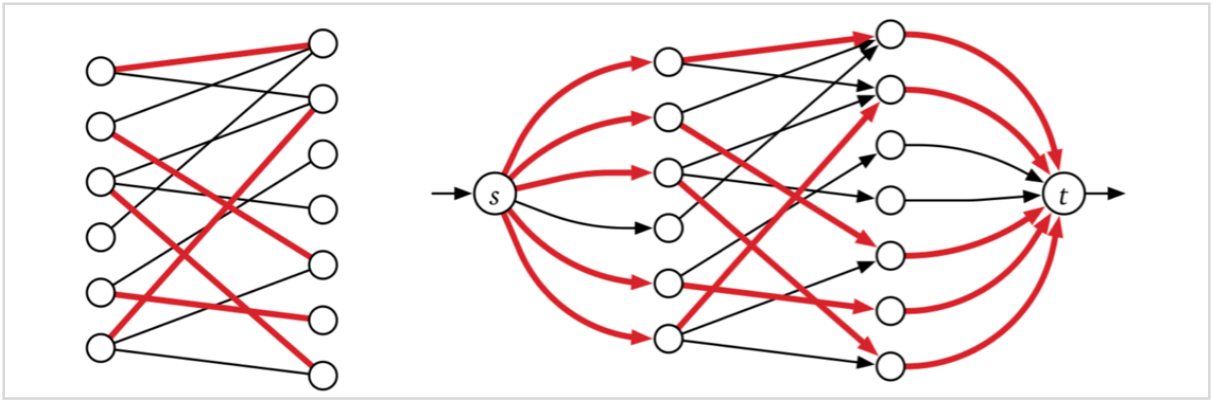
Edmonds-Karp Shortest Augmenting Paths and More!

- But we can get around these issues if we carefully select our augmenting paths.
- The following strategy was proposed in Ford and Fulkerson's original paper, but Jack Edmonds and Richard Karp provided an analysis in the 1970s:
  - Choose the augmenting path with the smallest number of edges.
- We can find this path in O(E) time by doing a breadth-first search from s in the residual graph.
- Through a careful analysis, one can prove there are only O(VE) iterations, even if | f^* | is very large and even if you have arbitrary real capacities.
- Which means the total running time is O(VE^2).
- We don't have time to get into the details, but the proof uses the following intuition: you remove an edge from the residual graph in every iteration, either by saturating it with flow or setting it's reversal to have 0 flow.

- Every time you do that, the minimum number of edges on any residual path from u ➔ v increases. But paths have at most O(V) edges, so any edge is removed O(V) times.
- There have been many more algorithms discovered since then. The fastest one known today was described by Orlin in 2012 and runs in O(VE) time.
- The algorithm uses a lot of fancy techniques and advanced data structures. In fact, very few people understand this algorithm. I'm afraid I am not one of them.
- But for the purposes of doing homework or the final exam, you should feel free to cite it.
- **Maximum flows and minimum cuts can be computed in O(VE) time.**

Bipartite Matching

- For the rest of today, I want to discuss some applications for computing maximum flows, because they're surprisingly useful tools
- Suppose we have an undirected bipartite graph G = (U cup W, E). A *matching* in G is a subgraph in which every vertex has degree at most one. In other words, we pair up some vertices, but no vertex appears in more than one pair.
- We want to find a matching with the maximum number of edges. Maybe U is a set of medical students and W is a set of slots in residency programs. There are edges between compatible assignments, and we want as many assignments as possible.
- We can use flows after modifying the graph a bit. We create a directed graph G' by
  - orienting edges from U to W
  - adding new vertices s and t
  - adding edges from s to every vertex in U
  - adding edges from every vertex in W to t
  - every edge in G' gets a capacity of 1



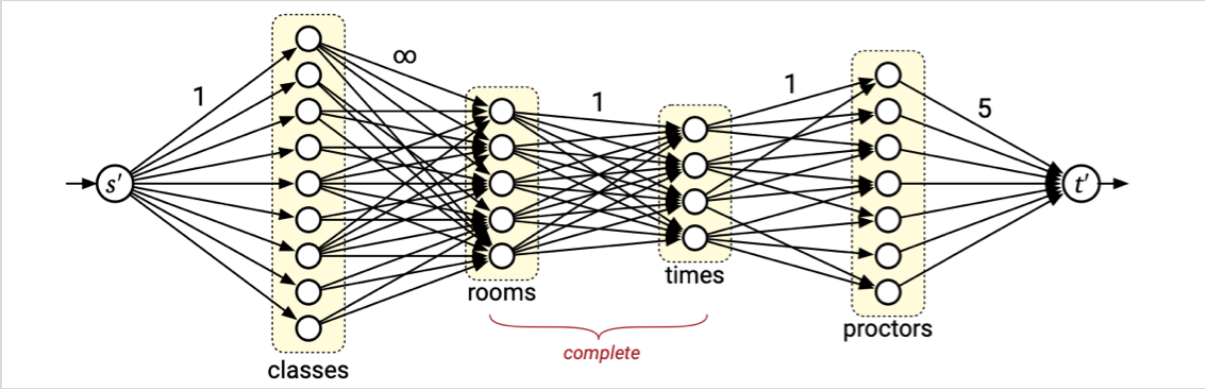
- Any matching M can be turned into a flow f_M in G'. For each edge uw in M, push one unit of flow along s ➔ u ➔ w ➔ t. We have | f_M | = | M |.
- But now say we compute a maximum flow f^*. We now know |f^ *| ≥ |M^ *| where M^ * is the maximum cardinality matching. But can we fine a matching that large?
- The capacities are integers, so every edge is assigned an integral value of flow, either 0 or 1.

- Every vertex in U has one incoming edge, so at most 1 unit of incoming flow. So there is at most one outgoing edge with 1 unit of flow. Similarly, every member of W has at most 1 incoming edge with one unit of flow.
- So the edges from U to W with 1 unit form a matching. And they are incident to the | f^* | edges coming out of s, so the matching has size |f^ *|.
- We could use Orlin's algorithm in O(VE) time to find the maximum flow, but that's overkill.
- Instead, we can just run Ford-Fulkerson in O(VE) time, because there are at most V/2 edges in any matching and the maximum flow has value |V| / 2.

Exam Scheduling

- I gave an example where the matching can be thought of as an assignment. We can extend these ideas to assign multiple kinds of things at once.
- Suppose we've been asked to schedule the final exams for next Spring. There are
  - n different classes
  - r rooms, and
  - t different time slots
  - p proctors to oversee the exams
- At most one class's final exam can be scheduled in each room during each time slot, and classes cannot be split between multiple rooms and time slots.
- Each proctor can
  - oversee one exam at a time,
  - is available for only certain time slots,
  - can oversee at most 5 exams total
- Finally, we have to worry about whether people can fit in the rooms too!
- Here's all the input:
  - Integer array E[1 .. n] where E[i] is the number of students enrolled in class i.
  - Integer array S[1 .. r] where S[j] is the number of seats in rooms j. So class i's final can be held in room j if and only if E[i] ≤ S[j].
  - Boolean array A[1 .. t, 1 .. p] where A[k, ell] = True if and only if proctor ell is available during the kth time slot.
- Wow, that's a lot to take in!
- But there's a pretty slick reduction to maximum flow, just like before.
- We'll construct a graph G with *six* types of vertices:
  - source vertex s'.
  - a vertex c_i for each class i
  - a vertex r_j for each room j
  - a vertex t_k for each time slot
  - a vertex p_ell for each proctor, and

    - a target vertex t'
- There are five types of edges:
    - an edge s ➔ c_i with capacity 1 for each class (one final exam per class),
    - an infinite capacity edge c_i ➔ r_j for each class i and room j such that E[i] ≤ S[j] (class i can fit in room j if there are more students than seats)
    - an edge r_j ➔ t_k with capacity 1 for each room j and time slot k (at most one exam can be held in room j at time k)
    - an edge t_k ➔ p_ell with capacity 1 for time slot k and proctor ell such that A[ell, k] = True (a proctor can oversee one exam at a time, and only when they are available)
    - an edge p_ell ➔ t' with capacity 5 for each proctor ell (each proctor can oversee at most 5 exams)
- So G has n + r + t + p + 2 vertices and O(nr + rt + tp) edges.



- As you might guess, we're going to compute a maximum (s',t')-flow.
- If we're given an assignment of class i, room j, time k, and proctor ell, we can map it to a path s' ➔ c_i ➔ r_j ➔ t_k ➔ p_ell ➔ t'.
- So, we can take a correct assignment for all the classes, and it will map to a flow of value n. There's n paths out of s. One per class. One per room-time slot pair. One per time slot-proctor pair. And at most 5 per proctor. So if we write down how many times each edge is used in these paths, we get a feasible (s',t')-flow of value n.
- So if we compute a maximum (s',t')-flow f*, it will have value at least n if there is any way to assign all the exams.
- If there is, it will have value exactly n, because there's that cut ({s}, V \ {s}).
- And like before, this flow will have integral values. So we'll peel off one path, reduce the flow on each edge of the path by 1, and recurse to get all the assignments.
- Again, Ford-Fulkerson works just fine here, so the whole thing will take O(VE) = O((n + r + t + p)(nr + rt + tp)) time.
- You can solve many kinds of "assignment" style problems in this way. Make one set of vertices per type of object. Add edges between objects that can be assigned to one another. Add some capacities if some pair or some object can be involved in a limited number of assignments.
- The lecture notes provide a bit more of a framework, but this is another one of those

things you kind of get a feel for.

- And with that, we're done discussing graph algorithms.
- And in a way, we're done discussing algorithm design.
- Next week, we'll do midterm review and Midterm 2.
- The week after that, we're going to discuss situations in which we (probably) *can't* design efficient algorithms.