Network Flow and Ford-Fulkerson

Suhas Arehalli

COMP221 - Spring 2024

1 Network Flow Problems

- A different kind of graph problem!
- Rather than have weights represent the *cost* of an edge, weights represent *capacities* across edges.
- The metaphor is that the graphs represent a pipeline, and we want to measure the possible *flow* through the graph. Capacities are limits on the amount that can flow through any particular edge.
- Let's formalize: Consider a graph $G = (V, E)$ with capacities $c : E \to \mathbb{R}_{\geq 0}$.
- How much can flow from s to t?
  - Along an edge e, the maximum flow is equivalent to the edge's capacity $c(e)$.
  - Along a path P, our flow can get *bottle-necked*! We're limited by the smallest capacity along that path, $\min_{e \in P} c(e)$
  - Across a full graph, things are more complicated (think of all of the different ways that flow can travel! We'll need to develop some theory in order to make sense of this problem!

2 Network Flow Theory

2.1 Max-Flow = Min-Cut

- The first idea we'll need to understand how to handle flows is the *Max-Flow = Min-Cut* theorem.
- Unfortunately, it's proof is a bit beyond what we can do in this class, but we'll study the result regardless!
- But first, definitions!
- A *cut* of a graph $G = (V, E)$ is a partition of vertices V into two sets V_1, V_2 (i.e., for any $v \in V$, $v \in V_1$ or $v \in V_2$, but not both!). An *s-t cut* is a cut where $s \in V_1$ and $t \in V_2$.

- An edge (v, w) *crosses the cut* if $v \in V_1$ and $w \in V_2$. The *cost* of a cut is the sum of the costs of all edges that cross the cut. That is,

$$c(V_1, V_2) = \sum_{\substack{(v,w) \in E \\ v \in V_1 \\ w \in V_2}} c(e)$$

- What this theorem tells us is that the cost of the *cheapest* $s - t$ cut is equal to the maximum flow through a graph.
- Why? Let's think about the relationship between a cut and flow:
  - If $s \in V_1$ and $t \in V_2$, for any flow to get from s to t, it *must* cross the cut.
  - Since this is true for *any* cut, it must be true for the *cheapest* cut.
  - This means that the cheapest cut acts as a bottleneck! Since all flow must cross through the cheapest cut, the maximum possible flow is the capacity/cost of that cut!
- This result will help us prove the correctness of the algorithm we'll eventually use. But before that, the next bit of theory will motivate the algorithmic approach.

2.2 Residual Graphs & Augmenting Paths

- Whenever we talk about flows through a graph, we're talking about using some of the capacity of each edge.
- We're going to adopt an incremental approach: We'll start with a bad flow, and slowly make it better.
- To know whether we can improve our flow, it's useful to know how much of the capacity along each edge we're using.
  - We formalize this in terms of a *flow function* $f : E \to \mathbb{R}$
  - This flow function must obey a number of rules, in order to mirror our intuitions about how flow works:
    1. **Capacity limits:** For all $e \in E$, $f(e) \leq c(e)$. That is, we can't have more flow than there is capacity.
    2. **Direction of flow:** For $(v, w) \in E$, $f((v, w)) = -f((w, v))$. That is, the direction of flow is indicated by the sign of the flow function. Note that this means that we're working with directed graphs here!
    3. **Conservation of flow:** For $v \in V \backslash \{s, t\}$, $\sum_{(w,v) \in E} f((w, v)) + \sum_{(v,w) \in E} f((v, w)) = 0$. That is, the flow into a vertex must equal the flow out of that vertex, unless it's the source or target vertex. Note that because flow in and out are represented by different signs, we can represent this by requiring that their sum is 0!
- With an f in mind, we can then construct what we call a *residual graph* G_r, which keeps track of all of the leftover capacity in our graph.

- We define $G_r = (V, E_r)$ with a residual cost function $c_r : E \to R$ such that

$$c_r(e) = c(e) - f(e)$$
$$E_r = \{e \in E \mid c_r(e) > 0\}$$

- There are a few critical observations here!
- Note that we are implicitly treating G as directed here, since f cares about directed edges.
- This means that, for some edge (v, w) with $c((v, w)) = c((w, v)) = k$, $c_r((v, w)) = k - f((v, w))$, but

$$\begin{aligned} c_r((w, v)) &= k - f((w, v)) \\ &= k + f((v, w)) \end{aligned}$$

  That is, every time we increase flow in one direction, we increase *capacity* in the other!
- One way of thinking about this is by think of capacities as potential *changes in the flow function*. If we currently have a flow of x in one direction on an edge with capacity k, it's possible to undo x flow in that direction and add k flow in the other direction. That's a net change of $x + k$ flow!
- The last thing to notice is that remove edge that have 0 capacity left (in network flow terminology, the edge is *saturated*). This will result in a fairly nice property: *Any path from s to t can carry > 0 flow*!

- Because of that last property, we call any path from s to t in G_r an *augmenting path*, since it means that there is unused capacity along that path we can use to further augment the flow from s to t!
- And this leads to a critical pair of results:
  - If there exists an augmenting path from s to t in G_r, the flow function f is not *maximal* (i.e., there exists a flow function such that the flow out of s (and into t) is greater!).
  - As a corollary, if s is disconnected from t in G_r, f is a maximal flow from s to t.

3 The Ford-Fulkerson method

- Building on the theory of Augmenting Paths and Residual Graph, Ford-Fulkerson presents a method for finding the maximum flow, outlined in Alg. 1.
  - That is, we start with no flow on any edge. Then we repeatedly find an augmenting path and augment our flow functions along that path with the maximum flow through that augmenting path, and then update G_r.
  - We repeat until there is no augmenting path (i.e., t is not reachable from s in G_r)
- Note that this is a *method*, not an algorithm! Like Quicksort, in practice Ford-Fulkerson is a famility of algorithms that use this approach. For this to be an actual algorithm we can analyze more carefully, we need to make some choices, the most critical being how we find an augmenting path in G_r!

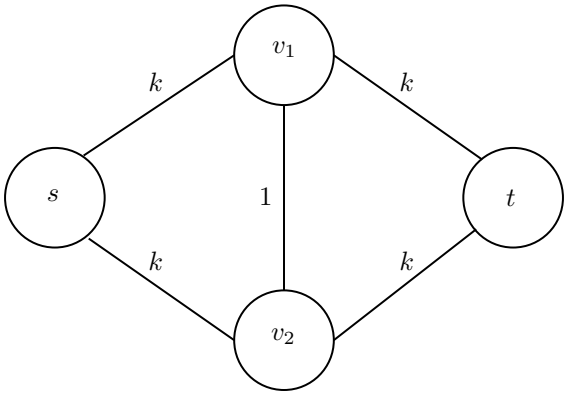

Figure 1: A weighted graph G

- For reasons beyond the scope of this course, the most common algorithm under the Ford-Fulkerson umbrella, called the Edmonds-Karp Algorithm, uses BFS to find augmenting paths!

Algorithm 1 The Ford-Fulkerson Method

```
function FordFulkerson(G = (V, E), c, s, t)
    Let f(e) ← 0, ∀e ∈ E
    G_r ← G
    while s is connected to t in G_r do
        Find an augmenting path P
        f_aug ← min_{e∈P} c(e)
        for (v, w) ∈ P do
            f((v, w)) ← f((v, w)) + f_aug
            f((w, v)) ← f((w, v)) − f_aug
        end for
        E_r ← {e ∈ E | c(e) − f(e) > 0}
        G_r ← (V, E_r)
    end while
end function
```

3.1 Runtime Complexity

- Alternatively, this section's goal is to convince you that picking *good* augmenting paths matters!
- Consider the graph in Fig. 1
- Observe that if we pick *good* augmenting paths, we can find the maximum flow in two iterations of Ford-Fulkerson: Find the paths $P_1 = (s, v_1), (v_1, t)$ and $P_2 = (s, v_2), (v_2, t)$ to get the maximum flow $2k$!

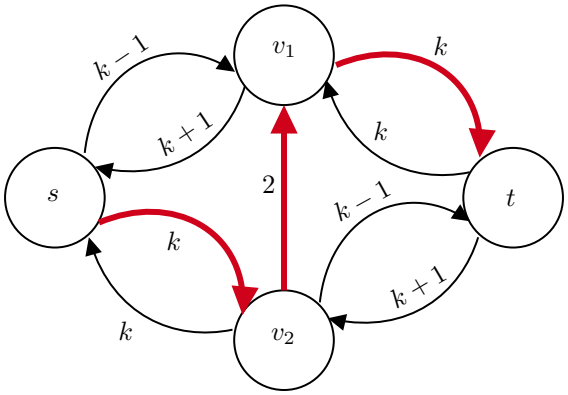

Figure 2: A residual graph G_r after one step of Ford-Fulkerson on the graph in Fig. 1 with augmenting path $P = (s, v_1), (v_1, v_2), (v_2, t)$. Consider the next (bad) augmenting path in red.

- But what if we use an algorithm that picks really bad augmenting paths?
  - Consider first an augmenting path $P = (s, v_1), (v_1, v_2), (v_2, t)$. This path can carry a maximum flow of 1 (bottlenecked by (v_1, v_2))!
  - The residual graph will then look like Fig. 2. Then let's run another step of Ford-Fulkerson with the augmenting path in red, which will have maximum flow 2!
  - Now if you draw the residual graph that results, you'll see that the middle edge between v_1 and v_2 runs in one direction with residual capacity 2. We can repeatedly construct a bad augmenting path through that edge that has a maximum flow of 2.
  - This means that a particularly bad (i.e., *worst-case*) implementation of Ford-Fulkerson would take $Theta(k)$ iterations of the while loop in order to find the maximum flow on this graph!
- Based on this kind of argument, one can prove that if the graph has integer weights (i.e., $c : E \to \mathbb{Z}_{\geq 0}$, then Ford-Fulkerson, worst-case, takes $\Theta(f(|V| + |E|))$ time, assume a graph-linear way of finding an augmenting path (i.e., BFS/DFS).
  - Why? Consider: What is the minimum increase in flow from each augmenting path?
- Edmonds-Karp (i.e., using BFS for find augmenting paths) is the preferred Ford-Fulkerson algorithm because it can be shown that we tend to pick good paths: We need, at worst, $|V||E|$ augmenting paths to find the maximum flow). This gives us a time complexity of $\Theta(|V||E|(|V| + |E|))$, which is often just written as $\Theta(|V||E|^2)$ assuming that $|E| \gg |V|$ (i.e., the graph is dense).

3.2 Proof of Correctness

- We're not going to *formally* prove this algorithm correct (Though I ask you to walk through the structure in the activity!)
- But it's worth reiterating the structure here for posterity:

- First, we prove with a loop invariant that the algorithm computes a valid flow at each step (all flow properties are preserved!).
- Then we show that once no augmenting paths exist, that all edges in the minimum cut at saturated (i.e., those edges don't exist in E_r, and thus s is disconnected from t!). Then, as a result of the conservation of flow, the flow from s into t is equal to the capacity across the minimum cut! By max-flow = min-cut, that means we found the maximum flow!
- The last missing bit is proof that the algorithm will terminate. We haven't had to consider this often, so it's worth highlighting here: Are we guaranteed to find the maximum flow in a finite number of iterations?
  - If we have integer capacities, this is straightforward to show! This is how we get our runtime complexity!
  - It's also fairly easy to convert a rational-weighted graph to an integer weighted graph, so those are fine as well!
  - It turns out if we have real weights, it depends on our choice of path-finding algorithm! This is part of the reason Edmonds-Karp is important: They proved that BFS will find augmenting paths in a way that guarantees termination in finite time![1]

4 Application: Bipartite Matching

- While maximum flow might seem like a very specific kind of problem, it turns out that a variety of problems can be converted into maximum-flow problems!
- One major class of problems you can do this with is that of *bipartite matching* problems.
- Suppose you have two class of entities, A and B, such that some $a \in A$ and $b \in B$ can form a pair (a, b). Let E be the set of possible pairs. The bipartite matching problem asks you to find the *maximum* number of pairs (a, b) for some $a \in A, b \in B$ such that every $a \in A, b \in B$ is only in a single pair.
- For example, consider a set J of job listings and a set A of applicants for those jobs. Then $(j, a) \in E$ iff $ainA$ is qualified for job $j \in J$. Each listing will only hire one applicant, and each applicant can only accept one job. The solution to the bipartite matching problem then asks what the optimal pairing of applicants and jobs is, where optimal means the most jobs get filled/people get hired (turns out given the other constraints, these are equivalent!).
- This is called the bipartite matching problem because this problem can be represented as a *bipartite graph*. For entities A and B and possible pairings E, $G = (A \cup B, E)$.

[1] To see why this is a problem, it's worth considering our situation: Because we exclude edges with capacity 0, we are guaranteed non-zero capacity gained every iteration. If our flow from s to t is increasing each iteration, how it it possible we don't eventually hit the max flow? A motivating example might be the idea of infinite convergent sequences. Consider the sum $\sum_{i=1}^{k} \frac{1}{2^i}$. For any choice of $k > 1$ (i.e., no matter how many new, positive terms we add), this sum will be less than 1! One can construct pathological graphs and pathological sequences of augmenting paths over those graphs that mirror these conditions, though it's not quite as simple as the summation (given the statement about rational graphs, we need *irrational* weights to construct a non-convergent graph for Ford-Fulkerson)! You can see a non-convergent example at Wikipedia!

- To frame this as a maximum flow problem:
  - Construct two addition vertices s and t.
  - Add edges from s to every vertex in A and from t to every vertex in B.
  - Add capacities of 1 to every edge.
- Formally, we construct $G' = (V', E')$, where

$$V' = A \cup B \cup \{s, t\}$$
$$E' = E \cup \{(s, a) \mid a \in A\} \cup \{(b, t) \mid b \in B\}$$

  and $c(e) = 1, \forall e \in E'$.
- My claim is that if we find the maximal flow function f, the optimal set of pairings for the bipartite matching problem is precisely the edges in E with a flow of 1! That is,

$$\{e \in E \mid f(e) = 1\}.$$

- Proving this is correct (i.e., a maximum flow solution is equivalent to a bipartite matching solution) is a good exercise!