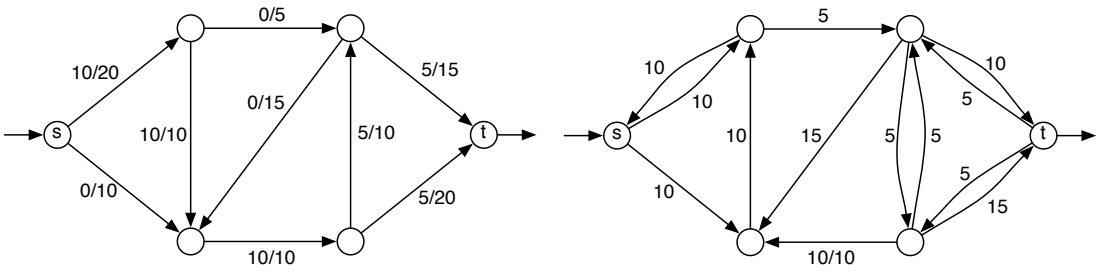
15 Max-Flow Algorithms and Applications (November 15)

15.1 Recap

Fix a directed graph $G = (V, E)$ that does not contain both an edge $u \to v$ and its reversal $v \to u$, and fix a capacity function $c : E \to \mathbb{R}_+$. For any flow function $f : E \to \mathbb{R}_{\geq 0}$, the *residual capacity* is defined as

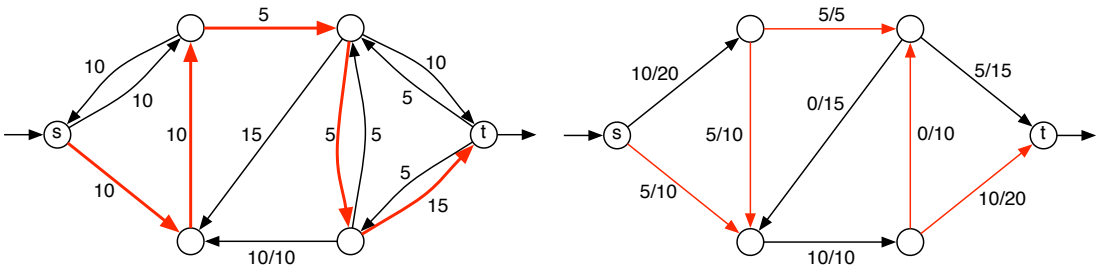
$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}.$$

The *residual graph* $G_f = (V, E_f)$, where E_f is the set of edges whose non-zero residual capacity is positive.



A flow f in a weighted graph G and its residual graph G_f.

In the last lecture, we proved the Max-flow Min-cut Theorem: *In any weighted directed graph network, the value of the maximum (s,t)-flow is equal to the cost of the minimum (s,t)-cut.* The proof of the theorem is constructive. If the residual graph contains a path from s to t, then we can increase the flow by the minimum capacity of the edges on this path, so we must not have the maximum flow. Otherwise, we can define a cut (S, T) whose cost is the same as the flow f, such that every edge from S to T is saturated and every edge from T to S is empty, which implies that f is a maximum flow and (S, T) is a minimum cut.



An augmenting path in G_f and the resulting (maximum) flow f'.

15.2 Ford-Fulkerson

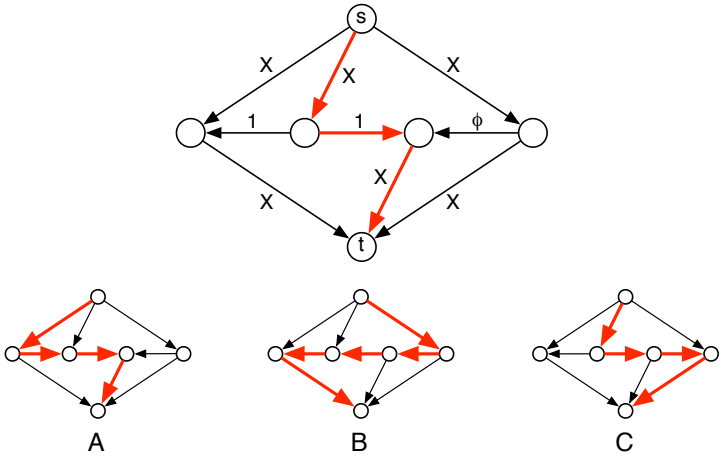
It's not hard to realize that this proof translates almost immediately to an algorithm, first developed by Ford and Fulkerson in the 1950s: Starting with the zero flow, repeatedly augment the flow along **any** path $s \rightsquigarrow t$ in the residual graph, until there is no such path.

If every edge capacity is an integer, then every augmentation step increases the value of the flow by a positive integer. Thus, the algorithm halts after $|f^*|$ iterations, where f^* is the actual

maximum flow. Each iteration requires $O(E)$ time, to create the residual graph G_f and perform a whatever-first-search to find an augmenting path. Thus, in the words case, the Ford-Fulkerson algorithm runs in $\boxed{O(E|f^*|)}$ time.

If we multiply all the capacities by the same (positive) constant, the maximum flow increases everywhere by the same constant factor. It follows that if all the edge capacities are *rational*, then the Ford-Fulkerson algorithm eventually halts. However, if we allow irrational capacities, the algorithm can loop forever, always finding smaller and smaller augmenting paths. Worse yet, this infinite sequence of augmentations may not even converge to the maximum flow! One of the simplest example of this effect was discovered by Uri Zwick.

Consider the graph shown below, with six vertices and nine edges. Six of the edges have some large integer capacity X, two have capacity 1, and one has capacity $\phi = (\sqrt{5}-1)/2 \approx 0.618034$, chosen so that $1 - \phi = \phi^2$. To prove that the Ford-Fulkerson algorithm can get stuck, we watch the residual capacities of the three horizontal edges as the algorithm progresses. (The residual capacities of the other six edges will always be at least $X - 3$.)



Uri Zwick's non-terminating flow example, and three augmenting paths.

The Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown in the large figure above. The three horizontal edges,, in order from left to right, now have residual capacities 1, 0, ϕ. Suppose the horizontal residual capacities are ϕ^{k-1}, 0, and ϕ^k for some non-negative integer k.

1. Augment along B, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, \phi^k, 0$.
2. Augment along C, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^k$.
3. Augment along B, adding ϕ^{k+1} to the flow; the residual capacities are now $0, \phi^{k+1}, \phi^{k+2}$.
4. Augment along A, adding ϕ^{k+1} to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^{k+2}$.

Thus, after $4n + 1$ augmentation steps, the residual capacities are $\phi^{2n-2}, 0, \phi^{2n-1}$. As the number of augmentation steps grows to infinity, the value of the flow converges to

$$1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5} < 7,$$

even though the maximum flow value is clearly $2X + 1$.

Picky students might wonder at this point why we care about irrational capacities; after all, computers can't represent anything but (small) integers or (dyadic) rationals exactly. Good question! One reason is that the integer restriction is literally *artificial*; it's an *artifact* of actual computational hardware[1], not an inherent feature of the abstract mathematical problem. Another reason, which is probably more convincing to most practical computer scientists, is that the behavior of the algorithm with irrational inputs tells us something about its worst-case behavior *in practice* given floating-point capacities—terrible! Even with very reasonable capacities, a careless implementation of Ford-Fulkerson could enter an infinite loop simply because of round-off error!

15.3 Edmonds-Karp: Fat Pipes

The Ford-Fulkerson algorithm does not specify which alternating path to use if there is more than one. In 1972, Jack Edmonds and Richard Karp analyzed two natural heuristics for choosing the path. The first is essentially a greedy algorithm:

Choose the augmenting path with largest bottleneck value.

It's a fairly easy to show that the maximum-bottleneck (s,t)-path in a directed graph can be computed in $O(E \log V)$ time using a variant of Jarník's minimum-spanning-tree algorithm, or of Dijkstra's shortest path algorithm. Simply grow a directed spanning tree T, rooted at s. Repeatedly find the highest-capacity edge leaving T and add it to T, until T contains a path from s to t. Alternately, once could emulate Kruskal's algorithm—insert edges one at a time in decreasing capacity order until there is a path from s to t—although this is less efficient.

We can now analyze the algorithm in terms of the value of the maximum flow f^*. Let f be any flow in G, and let f' be the maximum flow *in the current residual graph* G_f. (At the beginning of the algorithm, $G_f = G$ and $f' = f^*$.) Let e be the bottleneck edge in the next augmenting path. Let S be the set of vertices reachable from s through edges with capacity greater than $c(e)$ and let $T = V \setminus S$. By construction, T is non-empty, and every edge from S to T has capacity at most $c(e)$. Thus, the cost of the cut (S,T) is at most $c(e) \cdot E$. On the other hand, $\|S,T\| \geq |f|$, which implies that $c(e) \geq |f|/E$.

Thus, augmenting f along the maximum-bottleneck path in G_f multiplies the maximum flow value in G_f by a factor of at most $1 - 1/E$. In other words, the residual flow *decays exponentially* with the number of iterations. After $E \cdot \ln|f^*|$ iterations, the maximum flow value in G_f is at most

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

(That's Euler's constant e, not the edge e. Sorry.) In particular, *if all the capacities are integers*, then after $E \cdot \ln|f^*|$ iterations, the maximum capacity of the residual graph is *zero* and f is a maximum flow.

We conclude that for graphs with integer capacities, the Edmonds-Karp 'fat pipe' algorithm runs in $\boxed{O(E^2 \log E \log|f^*|)}$ time.

15.4 Dinits/Edmonds-Karp: Short Pipes

The second Edmonds-Karp heuristic was actually proposed by Ford and Fulkerson in their original max-flow paper, and first analyzed by the Russian mathematician Dinits (sometimes transliterated Dinic) in 1970. Edmonds and Karp published their independent and slightly weaker analysis in 1972. So naturally, almost everyone refers to this algorithm as 'Edmonds-Karp'.

[1]...or perhaps the laws of physics. Yeah, right. Whatever. Like *reality* actually matters in this class.

Choose the augmenting path with fewest edges.

The correct path can be found in $O(E)$ time by running breadth-first search in the residual graph. More surprisingly, the algorithm halts after a polynomial number of iterations, independent of of the actual edge capacities!

The proof of this upper bound relies on two observations about the evolution of the residual graph. Let f_i be the current flow after i augmentation steps, let G_i be the corresponding residual graph. In particular, f_0 is zero everywhere and $G_0 = G$. For each vertex v, let $level_i(v)$ denote the unweighted shortest path distance from s to v in G_i, or equivalently, the *level* of v in a breadth-first search tree of G_i rooted at s.

Our first observation is that these levels can only increase over time.

Lemma 1. *$level_{i+1}(v) \geq level_i(v)$ for all vertices v and integers i.*

Proof: The claim is trivial for $v = s$, since $level_i(s) = 0$ for all i. Choose an arbitrary vertex $v \neq s$, and let $p \to \cdots \to u \to v$ be a shortest path from s to v in G_{i+1}. (If there is no such path, then $level_{i+1}(v) = \infty$, and we're done.) Because this is a shortest path, we have $level_{i+1}(v) = level_{i+1}(u) + 1$, and the inductive hypothesis implies that $level_{i+1}(u) \geq level_i(u)$.

We now have two cases to consider. If $u \to v$ is an edge in G_i, then $level_i(v) \leq level_i(u) + 1$, because the levels are defined by breadth-first traversal.

On the other hand, if $u \to v$ is not an edge in G_i, then $v \to u$ must be an edge in the ith augmenting path. Thus, $v \to u$ must lie on the shortest path from s to t in G_i, which implies that $level_i(v) = level_i(u) - 1 \leq level_i(u) + 1$.

In both cases, we have $level_{i+1}(v) = level_{i+1}(u) + 1 \geq level_i(u) + 1 \geq level_i(v)$. □

Whenever we augment the flow, the bottleneck edge in the augmenting path disappears from the residual graph, and some other edge in the *reversal* of the augmenting path may (re-)appear. Our second observation is that an edge cannot appear or disappear too many times.

Lemma 2. *During the execution of the Dinits/Edmonds-Karp algorithm, any edge $u \to v$ disappears from the residual graph G_f at most $V/2$ times.*

Proof: Suppose $u \to v$ is in two residual graphs G_i and G_{j+1}, but not in any of the intermediate residual graphs $G_{i+1}, \ldots, G_j$, for some $i < j$. Then $u \to v$ must be in the ith augmenting path, so $level_i(v) = level_i(u)+1$, and $v \to u$ must be on the jth augmenting path, so $level_j(v) = level_j(u)-1$. By the previous lemma, we have

$$level_j(u) = level_j(v) + 1 \geq level_i(v) + 1 = level_i(u) + 2.$$

In other words, the distance from s to u increased by at least 2 between the disappearance and reappearance of $u \to v$. Since every level is either less than V or infinite, the number of disappearances is at move $V/2$. □

Now we can derive an upper bound on the number of iterations. Since each edge can disappear at most $V/2$ times, there are at most $EV/2$ edge disappearances overall. But at least one edge disappears on each iteration, so the algorithm must halt after at most $EV/2$ iterations. Finally, since each iteration requires $O(E)$ time, Dinits' algorithm runs in $\boxed{O(VE^2) \text{ time}}$ overall.

15.5 Maximum Matchings in Bipartite Graphs

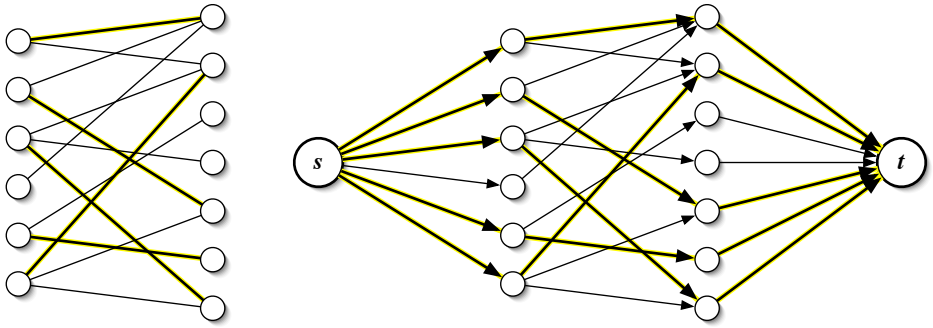
Perhaps one of the simplest applications of maximum flows is in computing a maximum-size *matching* in a bipartite graph. A matching is a subgraph in which every vertex has degree at most one, or equivalently, a collection of edges such that no two share a vertex. The problem is to find the largest matching in a given bipartite graph.

We can solve this problem by reducing it to a maximum flow problem as follows. Let G be the given bipartite graph with vertex set $V = U \cup W$, such that every edge joins a vertex in U to a vertex in W. We create a new *directed* graph G' by (1) orienting each edge from U to W, (2) adding two new vertices s and t, (3) adding edges from s to every vertex in U, and (4) adding edges from each vertex in W to t. Finally, we assign every edge in G' a capacity of 1.

Any matching M in G can be transformed into a flow f_M in G' as follows: For each edge (u, w) in M, push one unit of flow along the path $s \to u \to w \to t$. These paths are disjoint except at s and t, so the resulting flow satisfies the capacity constraints. Moreover, the value of the resulting flow is equal to the number of edges in M.

Conversely, consider any (s, t)-flow f in G' computed using the Ford-Fulkerson augmenting path algorithm. Because the edge capacities are integers, the Ford-Fulkerson algorithm assigns an integer flow to every edge. (This is easy to verify by induction hint hint.) Moreover, since each edge has *unit* capacity, the computed flow either saturates ($f(e) = 1$) or avoids ($f(e) = 0$) every edge in G'. Finally, since at most one unit of flow can enter any vertex in U or leave any vertex in W, the saturated edges from U to W form a matching in G. The size of this matching is exactly $|f|$.

Thus, the size of the maximum matching in G is equal to the value of the maximum flow in G', and provided we compute the maxflow using augmenting paths, we can convert the actual maxflow into a maximum matching. The maximum flow has value at most $\min\{|U|, |W|\} = O(V)$, so the Ford-Fulkerson algorithm runs in $\boxed{O(VE) \text{ time}}$.



A maximum matching in a bipartite graph G, and the corresponding maximum flow in G'

15.6 Edge-Disjoint Paths

Similarly, we can compute the maximum number of edge-disjoint paths between two vertices s and t in an graph using maximum flows. A set of paths in G is *edge-disjoint* if each edge in G appears in at most one of the paths. (Several paths may pass through the same vertex, however.)

If we give each edge capacity 1, then the maxflow from s to t assigns a flow of either 0 or 1 to every edge. Moreover, even if the original graph is undirected, the maxflow algorithm will assign a direction to every saturated edge. Thus, the subgraph S of saturated edges is the union of several edge-disjoint paths; the number of paths is equal to the value of the flow. Extracting the actual paths from S is easy: Just follow any directed path in S from s to t, remove that path from S, and recurse. The overall running time is $O(VE)$, just like for maximum bipartite matchings.

Conversely, we can transform any collection of edge-disjoint paths into a flow by pushing one unit of flow along each path from s to t; the value of the resulting flow is equal to the number of paths in the collection. It follows that the maxflow algorithm actually computes the largest possible set of edge-disjoint paths.