CS261: A Second Course in Algorithms
Lecture #2: Augmenting Path Algorithms for Maximum Flow*

Tim Roughgarden†

January 7, 2016

1 Recap

Figure 1: (a) original edge capacity and flow and (b) resultant edges in residual network.

Recall where we left off last lecture. We're considering a directed graph $G = (V, E)$ with a source s, sink t, and an integer capacity u_e for each edge $e \in E$. A flow is a nonnegative vector $\{f_e\}_{e \in E}$ that satisfies capacity constraints ($f_e \leq u_e$ for all e) and conservation constraints (flow in = flow out except at s and t).

Recall that given a flow f in a graph G, the corresponding residual network has two edges for each edge e of G, a forward edge with residual capacity $u_e - f_e$ and a reverse edge with residual capacity f_e that allows us to "undo" previously routed flow. See also Figure 1.[1]

The Ford-Fulkerson algorithm repeatedly finds an s-t path P in the current residual graph G_f, and augments along p as much as possible subject to the capacity constraints of

*©2016, Tim Roughgarden.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1]We usually implicitly assume that all edges with zero residual capacity are omitted from the residual network.

the residual network.[2] We argued that the algorithm eventually terminates with a feasible flow. But is it a maximum flow? More generally, a major course theme is to understand

> How do we know when we're done?

For example, could the maximum flow value in the network in Figure 2 really just be 3?

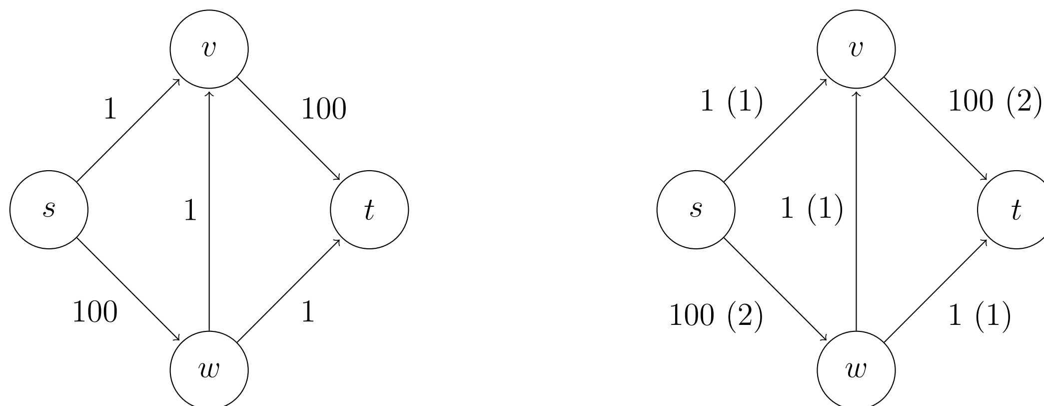

Figure 2: (a) A given network and (b) the alleged maximum flow of value 3.

2 Around the Maximum-Flow/Minimum-Cut Theorem

We ended last lecture with a claim that if there is no s-t path (with positive residual capacity on every edge) in the residual graph G_f, then f is a maximum flow in G. It's convenient to prove a stronger statement, from which we can also derive the famous maximum-flow/minimum cut theorem.

2.1 (s, t)-Cuts

To state the stronger result, we need an important definition, of objects that are "dual" to flows in a sense we'll make precise later.

Definition 2.1 (s-t Cut) An (s, t)-*cut* of a graph $G = (V, E)$ is a partition of V into sets A, B with $s \in A$ and $t \in B$.

Sometimes we'll simply say "cut" instead of "(s, t)-cut."

Figure 3 depicts a good (if cartoonish) way to think about an (s, t)-cut of a graph. Such a cut buckets the edges of the graph into four categories: those with both endpoints in A, those with both endpoints in B, those sticking out of A (with tail in A and head in B), and those sticking into A (with head in A and tail in B.

[2] To be precise, the algorithm finds an s-t path in G_f such that every edge has strictly positive residual capacity. Unless otherwise noted, in this lecture by "G_f" we mean the edges with positive residual capacity.

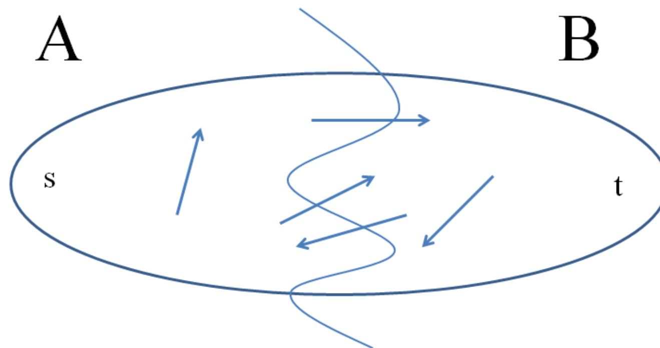

Figure 3: cartoonish visualization of cuts. The squiggly line splits the vertices into two sets A and B and edges in the graph into 4 categories.

The *capacity* of an (s,t)-cut (A,B) is defined as

$$\sum_{e \in \delta^+(A)} u_e.$$

where $\delta^+(A)$ denotes the set of edges sticking out of A. (Similarly, we later use $\delta^-(A)$ to denote the set of edges sticking into A.)

Note that edges sticking in to the source-side of an (s,t)-cut to do not contribute to its capacity. For example, in Figure 2, the cut $\{s,w\}$, $\{v,t\}$ has capacity 3 (with three outgoing edges, each with capacity 1). Different cuts have different capacities. For example, the cut $\{s\}$, $\{v,w,t\}$ in Figure 2 has capacity 101. A *minimum cut* is one with the smallest capacity.

2.2 Optimality Conditions for the Maximum Flow Problem

We next prove the following basic result.

Theorem 2.2 (Optimality Conditions for Max Flow) *Let f be a flow in a graph G. The following are equivalent:*[3]

(1) f is a maximum flow of G;

(2) there is an (s,t)-cut (A,B) such that the value of f equals the capacity of (A,B);

(3) there is no s-t path (with positive residual capacity) in the residual network G_f.

Theorem 2.2 asserts that any one of the three statements implies the other two. The special case that (3) implies (1) recovers the claim from the end of last lecture.

[3] Meaning, either all three statements hold, or none of the three statements hold.

Corollary 2.3 *If f is a flow in G such that the residual network G_f has no s-t path, then the f is a maximum flow.*

Recall that Corollary 2.3 implies the correctness of the Ford-Fulkerson algorithm, and more generally of any algorithm that terminates with a flow and a residual network with no s-t path.

Proof of Theorem 2.2: We prove a cycle of implications: (2) implies (1), (1) implies (3), and (3) implies (2). It follows that any one of the statements implies the other two.

Step 1: (2) implies (1): We claim that, for every flow f and every (s,t)-cut (A,B),

$$\text{value of } f \leq \text{capacity of } (A,B).$$

This claim implies that all flow values are at most all cut values; for a cartoon of this, see Figure 4. The claim implies that there no "x" strictly to the right of the "o".

Figure 4: cartoon illustrating that no flow value (x) is greater than a cut value (o).

To see why the claim yields the desired implication, suppose that (2) holds. This corresponds to an "x" and "o" that are co-located in Figure 4. By the claim, no "x"s can appear to the right of this point. Thus no flow has larger value than f, as desired.

We now prove the claim. If it seems intuitively obvious, then great, your intuition is spot-on. For completeness, we provide a brief algebraic proof.

Fix f and (A,B). By definition,

$$\text{value of } f = \underbrace{\sum_{e \in \delta^+(s)} f_e}_{\text{flow out of } s} = \sum_{e \in \delta^+(s)} f_e - \underbrace{\sum_{e \in \delta^-(s)} f_e}_{\text{vacuous sum}}; \tag{1}$$

the second equation is stated for convenience, and follows from our standing assumption that s has no incoming vertices. Recall that conservation constraints state that

$$\underbrace{\sum_{e \in \delta^+(v)} f_e}_{\text{flow out of } v} - \underbrace{\sum_{e \in \delta^-(v)} f_e}_{\text{flow into of } v} = 0 \tag{2}$$

for every $v \neq s,t$. Adding the equations (2) corresponding to all of the vertices of $A \setminus \{s\}$ to equation (1) gives

$$\text{value of } f = \sum_{v \in A} \left(\sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e \right). \tag{3}$$

Next we want to think about the expression in (3) from an edge-centric, rather than vertex-centric, perspective. How much does an edge e contribute to (3)? The answer depends on which of the four buckets e falls into (Figure 3). If both of e's endpoints are in B, then e is not involved in the sum (3) at all. If $e = (v, w)$ with both endpoints in A, then it contributes f_e once (in the subexpression $\sum_{e \in \delta^+(v)} f_e$) and $-f_e$ once (in the subexpression $-\sum_{e \in \delta^-(w)} f_e$). Thus edges inside A contribute net zero to (3). Similarly, an edge e sticking out of A contributes f_e, while an edge sticking into A contributes $-f_e$. Summarizing, we have

$$\text{value of } f = \sum_{e \in \delta^+(A)} f_e - \sum_{e \in \delta^-(A)} f_e.$$

This equation states that the net flow (flow forward minus flow backward) across every cut is exactly the same, namely the value of the flow f.

Finally, using the capacity constraints and the fact that all flows values are nonnegative, we have

$$\begin{aligned} \text{value of } f &= \sum_{e \in \delta^+(A)} \underbrace{f_e}_{\leq u_e} - \sum_{e \in \delta^-(A)} \underbrace{f_e}_{\geq 0} \\ &\leq \sum_{e \in \delta^+(A)} u_e \qquad (4) \\ &= \text{capacity of } (A, B), \qquad (5) \end{aligned}$$

which completes the proof of the first implication.

Step 2: (1) implies (3): This step is easy. We prove the contrapositive. Suppose f is a flow such that G_f has an s-t path P with positive residual capacity. As in the Ford-Fulkerson algorithm, we augment along P to produce a new flow f' with strictly larger value. This shows that f is not a maximum flow.

Step 3: (3) implies (2): The final step is short and sweet. The trick is to define

$$A = \{v \in V : \text{there is an } s \rightsquigarrow v \text{ path in } G_f\}.$$

Conceptually, start your favorite graph search subroutine (e.g., BFS or DFS) from s until you get stuck; A is the set of vertices you get stuck at. (We're running this graph search only in our minds, for the purposes of the proof, and not in any actual algorithm.)

Note that $(A, V - A)$ is an (s, t)-cut. Certainly $s \in A$, so s can reach itself in G_f. By assumption, G_f has no s-t path, so $t \notin A$. This cut must look like the cartoon in Figure 5, with no edges (with positive residual capacity) sticking out of A. The reason is that if there *were* such an edge sticking out of A, then our graph search would not have gotten stuck at A, and A would be a bigger set.

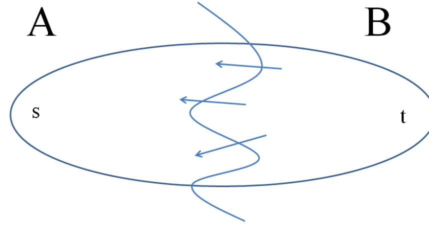

Figure 5: Cartoon of the cut. Note that edges crossing the cut only go from B to A.

Let's translate the picture in Figure 5, which concerns the residual network G_f, back to the flow f in the original network G.

1. Every edge sticking out of A in G (i.e., in $\delta^+(A)$) is saturated (meaning $f_u = u_e$). For if $f_e < u_e$ for $e \in \delta^+(A)$, then the residual network G_f would contain a forward version of e (with positive residual capacity) which would be an edge sticking out of A in G_f (contradicting Figure 5).

2. Every edge sticking into in A in G (i.e., in $\delta^-(A)$) is zeroed out ($f_u = 0$). For if $f_e < u_e$ for $e \in \delta^+(A)$, then the residual network G_f would contain a forward version of e (with positive residual capacity) which would be an edge sticking out of A in G_f (contradicting Figure 5).

These two points imply that the inequality (4) holds with equality, with

$$\text{value of } f = \text{capacity of } (A, V - A).$$

This completes the proof. ■

We can immediately derive some interesting corollaries of Theorem 2.2. First is the famous *Max-Flow/Min-Cut Theorem.*[4]

Corollary 2.4 (Max-Flow/Min-Cut Theorem) *In every network,*

$$\textit{maximum value of a flow} = \textit{minimum capacity of an } (s,t)\textit{-cut.}$$

Proof: The first part of the proof of Theorem 2.2 implies that the maximum value of a flow cannot exceed the minimum capacity of an (s, t)-cut. The third part of the proof implies that there cannot be a gap between the maximum flow value and the minimum cut capacity. ■

Next is an algorithmic consequence: the minimum cut problem reduces to the maximum flow problem.

Corollary 2.5 *Given a maximum flow, and minimum cut can be computed in linear time.*

[4]This is the theorem that, long ago, seduced your instructor into a career in algorithms.

Proof: Use BFS or DFS to compute, in linear time, the set A from the third part of the proof of Theorem 2.2. The proof shows that $(A, V - A)$ is a minimum cut. ■

In practice, minimum cuts are typically computed using a maximum flow algorithm and this reduction.

2.3 Backstory

Ford and Fulkerson published in the max-flow/min-cut theorem in 1955, while they were working at the RAND Corporation (a military think tank created after World War II). Note that this was in the depths of the Cold War between the (then) Soviet Union and the United States. Ford and Fulkerson got the problem from Air Force researcher Theodore Harris and retired Army general Frank Ross. Harris and Ross has been given, by the CIA, a map of the rail network connecting the Soviet Union to Eastern Bloc countries like Poland, Czechoslovakia, and Eastern Germany. Harris and Ross formed a graph, with vertices corresponding to administrative districts and edge capacities corresponding to the rail capacity between two districts. Using heuristics, Harris and Ross computed both a maximum flow and minimum cut of the graph, noting that they had equal value. They were rather more interested in the minimum cut problem (i.e., blowing up the least amount of train tracks to sever connectivity) than the maximum flow problem! Ford and Fulkerson proved more generally that in *every* network, the maximum flow value equals that minimum cut capacity. See [**?**] for further details.

3 The Edmonds-Karp Algorithm: Shortest Augmenting Paths

3.1 The Algorithm

With a solid understanding of when and why maximum flow algorithms are correct, we now focus on optimizing the running time. Exercise Set #1 asks to show that the Ford-Fulkerson algorithm is not a polynomial-time algorithm. It is a "pseudopolynomial-time" algorithm, meaning that it runs in polynomial time provided all edge capacities are polynomially bounded integers. With big edge capacities, however, the algorithm can require a very large number of iterations to complete. The problem is that the algorithm can keep choosing a "bad path" over and over again. (Recall that when the current residual network has multiple s-t paths, the Ford-Fulkerson algorithm chooses arbitrarily.) This motivates choosing augmenting paths more intelligently. The *Edmonds-Karp algorithm* is the same as the Ford-Fulkerson algorithm, except that it always chooses a *shortest* augmenting path of the residual graph (i.e., with the fewest number of hops). Upon hearing "shortest paths" you may immediately think of Dijkstra's algorithm, but this is overkill here — breadth-first search already computes (in linear time) a path with the fewest number of hops.

Edmonds-Karp Algorithm

initialize $f_e = 0$ for all $e \in E$
repeat
  compute an s-t path P (with positive residual capacity) in the current residual graph G_f with the fewest number of edges
  `// takes` $O(|E|)$ `time using BFS`
  if no such path **then**
    halt with current flow $\{f_e\}_{e \in E}$
  else
    let $\Delta = \min_{e \in P}$ (e's residual capacity in G_f)
    `// augment the flow` f `using the path` P
    for all edges e of G whose corresponding forward edge is in P **do**
      increase f_e by Δ
    for all edges e of G whose corresponding reverse edge is in P **do**
      decrease f_e by Δ

3.2 The Analysis

As a specialization of the Ford-Fulkerson algorithm, the Edmonds-Karp algorithm inherits its correctness. What about the running time?

Theorem 3.1 (Running Time of Edmonds-Karp [?]) *The Edmonds-Karp algorithm runs in $O(m^2 n)$ time.*[5]

Recall that m typically varies between $\approx n$ (the sparse case) and $\approx n^2$ (the dense case), so the running time in Theorem 3.1 is between n^3 and n^5. This is quite slow, but at least the running time is polynomial, no matter how big the edge capacities are. See below and Problem Set #1 for some faster algorithms.[6] Why study Edmonds-Karp, when we're just going to learn faster algorithms later? Because it provides a gentle introduction to some fundamental ideas in the analysis of maximum flow algorithms.

Lemma 3.2 (EK Progress Lemma) *Fix a network G. For a flow f, let $d(f)$ denote the number of hops in a shortest s-t path (with positive residual capacity) in G_f, or $+\infty$ if no such paths exist.*

(a) $d(f)$ never decreases during the execution of the Edmonds-Karp algorithm.

(b) $d(f)$ increases at least once per m iterations.

[5] In this course, m always denotes the number $|E|$ of edges, and n the number $|V|$ of vertices.

[6] Many different methods yield running times in the $O(mn)$ range, and state-of-the-art algorithm are still a bit faster. It's an open question whether or not there is a near-linear maximum flow algorithm.

Since $d(f) \in \{0, 1, 2, \ldots, n-2, n-1, +\infty\}$, once $d(f) \geq n$ we know that $d(f) = +\infty$ and s and t are disconnected in G_f.[7] Thus, Lemma 3.2 implies that the Edmonds-Karp algorithm terminates after at most mn iterations. Since each iteration just involves a breadth-first-search computation, we get the running time of $O(m^2 n)$ promised in Theorem 3.1.

For the analysis, imagine running breadth-first search (BFS) in G_f starting from the source s. Recall that BFS discovers vertices in "layers," with s in the 0th layer, and layer $i+1$ consisting of those vertices not in a previous layer and reachable in one hop from a vertex in the ith layer. We can then classify the edges of G_f as *forward* (meaning going from layer i to layer $i+1$, for some i), *sideways* (meaning both endpoints are in the same layer), and *backwards* (traveling from a layer i to some layer j with $j < i$). By the definition of breadth-first search, no forward edge of G_f can shortcut over a layer; every forward edge goes only to the next layer.

We define L_f, with the L standing for "layered," as the subgraph of G_f consisting only of the forward edges (Figure 6). (Vertices in layers after the one containing t are irrelevant, so they can be discarded if desired.)

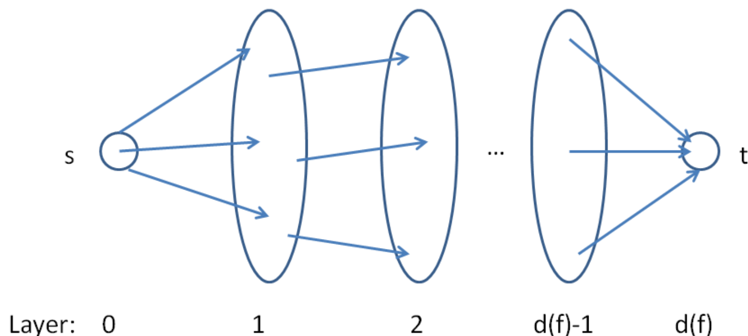

Figure 6: Layered subgraph L_f

Why bother defining L_f? Because it is a succinct encoding of all of the shortest s-t paths of G_f — the paths on which the Edmonds-Karp algorithm might augment. Formally, every s-t in L_f comprises only forward edges of the BFS and hence has exactly $d(f)$ hops, the minimum possible. Conversely, an s-t path that is in G_f but not L_f must contain at least one detour (a sideways or backward edge) and hence requires at least $d(f)+1$ hops to get to t.

[7] Any path with n or more edges has a repeated vertex, and deleted the corresponding cycle yields a path with the same endpoints and fewer hops.

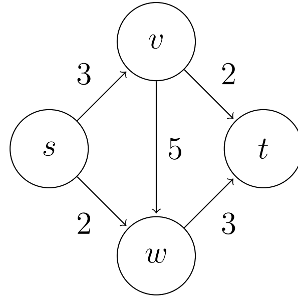

Figure 7: Example from first lecture. Initially, 0th layer is $\{s\}$, 1st layer is $\{v, w\}$, 2nd layer is $\{t\}$.

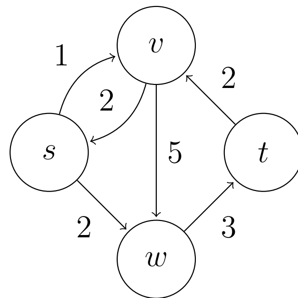

Figure 8: Residual graph after sending flow on $s \to v \to t$. 0th layer is $\{s\}$, 1st layer is $\{v, w\}$, 2nd layer is $\{t\}$.

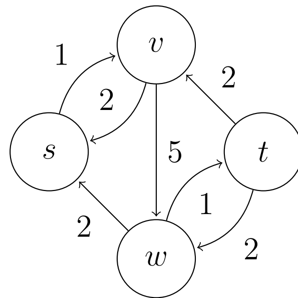

Figure 9: Residual graph after sending additional flow on $s \to w \to t$. 0th layer is $\{s\}$, 1st layer is $\{v\}$, 2nd layer is $\{w\}$, 3rd layer is $\{t\}$.

For example, let's return to our first example in Lecture #1, shown in Figure 7. Let's watch how $d(f)$ changes as we simulate the algorithm. Since we begin with the zero flow, initially the residual graph G_f is the original graph G. The 0th layer is s, the first layer is $\{v, w\}$, and the second layer is t. Thus $d(f) = 2$ initially. There are two shortest paths,

$s \to v \to t$ and $s \to w \to t$. Suppose the Edmonds-Karp algorithm chooses to augment on the upper path, sending two units of flow. The new residual graph is shown in Figure 8. The layers remain the same: $\{s\}$, $\{v, w\}$, and $\{t\}$, with $d(f)$ still equal to 2. There is only one shortest path, $s \to w \to t$. The Edmonds-Karp algorithm sends two units along this flow, resulting in the new residual graph in Figure 9. Now, no two-hop paths remain: the first layer contains only v, with w in second layer and t in the third layer. Thus, $d(f)$ has jumped from 2 to 3. The unique shortest path is $s \to v \to w \to t$, and after the Edmonds-Karp algorithm pushes one unit of flow on this path it terminates with a maximum flow.

Proof of Lemma 3.2: We start with part (a) of the lemma. Note that the only thing we're worried about is that an augmentation somehow introduces a new, shortest path that shortcuts over some layers of L_f (as defined above).

Suppose the Edmonds-Karp algorithm augments the current flow f by routing flow on the path P. Because P is a shortest s-t path in G_f, it is also a path in the layered graph L_f. The only new edges created by augmenting on P are edges that go in the reverse direction of P. These are all backward edges, so any s-t of G_f that uses such an edge has at least $d(f) + 2$ hops. Thus, no new shorter paths are formed in G_f after the augmentation.

Now consider a run of t iterations of the Edmonds-Karp algorithm in which the value of $d(f) = c$ stays constant. We need to show that $t \leq m$. Before the first of these iterations, we save a copy of the current layered network: let F denote the edges of L_f at this time, and $V_0 = \{s\}, V_1, V_2, \ldots, V_c$ the vertices if the various layers.[8]

Consider the first of these t iterations. As in the proof of part (a), the only new edges introduced go from some V_i to V_{i-1}. By assumption, after the augmentation, there is still an s-t path in the new residual graph with only c hops. Since no edge of of such a path can shortcut over one of the layers $V_0, V_1, \ldots, V_c$, it must consist only of edges in F. Inductively, every one of these t iterations augments on a path consisting solely of edges in F. Each such iteration zeroes out at least one edge $e = (v, w)$ of F (the one with minimum residual capacity), at which point edge e drops out of the current residual graph. The only way e can reappear in the residual graph is if there is an augmentation in the reverse direction (the direction (w, v)). But since (w, v) goes backward (from some V_i to V_{i-1}) and all of the t iterations route flow only on edges of F (from some V_i to to V_{i+1}), this can never happen. Since F contains at most m edges, there can only be m iterations before $d(f)$ increases (or the algorithm terminates). ■

4 Dinic's Algorithm: Blocking Flows

The next algorithm bears a strong resemblance to the Edmonds-Karp algorithm, though it was developed independently and contemporaneously by Dinic. Unlike the Edmonds-Karp algorithm, Dinic's algorithm enjoys a modularity that lends itself to optimized algorithms with faster running times.

[8] The residual and layered networks change during these iterations, but F and $V_0, \ldots, V_c$ always refer to networks before the first of these iterations.

Dinic's Algorithm

```
initialize f_e = 0 for all e ∈ E
while there is an s-t path in the current residual network G_f do
    construct the layered network L_f from G_f using breadth-first search,
     as in the proof of Lemma 3.2
    // takes O(|E|) time
    compute a blocking flow g (Definition 4.1) in L_f
    // augment the flow f using the flow g
    for all edges (v, w) of G for which the corresponding forward edge
     of G_f carries flow (g_vw > 0) do
        increase f_e by g_e
    for all edges (v, w) of G for which the corresponding reverse edge
     of G_f carries flow (g_wv > 0) do
        decrease f_e by g_e
```

Dinic's algorithm can only terminate with a residual network with no s-t path, that is, with a maximum flow (by Corollary 2.3). While in the Edmonds-Karp algorithm we only formed the layered network L_f in the analysis (in the proof of Lemma 3.2), Dinic's algorithm explicitly constructs this network in each iteration.

A blocking flow is, intuitively, a bunch of shortest augmenting paths that get processed as a batch. Somewhat more formally, blocking flows are precisely the possible outputs of the naive greedy algorithm discussed at the beginning of Lecture #1. Completely formally:

Definition 4.1 (Blocking Flow) A *blocking flow* g in a network G is a feasible flow such that, for every s-t path P of G, some edge e is saturated by g (i.e.,. $f_e = u_e$).

That is, a blocking flow zeroes out an edge of every s-t path.

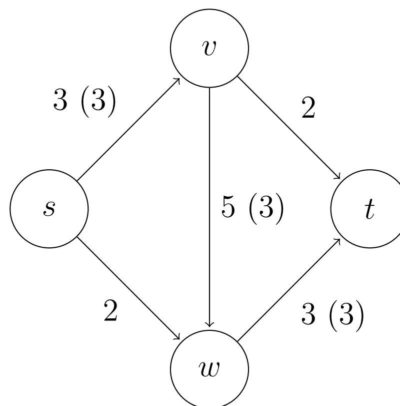

Figure 10: Example of blocking flow. This is not a maximum flow.

Recall from Lecture #1 that a blocking flow need not be a maximum flow; the blocking flow in Figure 10 has value 3, while the maximum flow value is 5. While the blocking flow in Figure 10 uses only one path, generally a blocking flow uses many paths. Indeed, every flow that is maximum (equivalently, no s-t paths in the residual network) is also a blocking flow (equivalently, no s-t paths in the residual network comprising only forward edges).

The running time analysis of Dinic's algorithm is anchored by the following progress lemma.

Lemma 4.2 (Dinic Progress Lemma) *Fix a network G. For a flow f, let $d(f)$ denote the number of hops in a shortest s-t path (with positive residual capacity) in G_f, or $+\infty$ if no such paths exist (or $+\infty$ if no such paths exist). If h is obtained from f be augmenting f by a blocking flow g in G_f, then $d(h) > d(f)$.*

That is, every iteration of Dinic's algorithm strictly increases the s-t distance in the current residual graph.

We leave the proof of Lemma 4.2 as Exercise #5; the proof uses the same ideas as that of Lemma 3.2. For an example, observe that after augmenting our running example by the blocking flow in Figure 10, we obtain the residual network in Figure 11. We had $d(f) = 2$ initially, and $d(f) = 3$ after the augmentation.

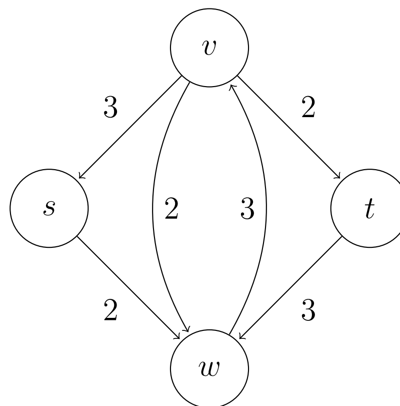

Figure 11: Residual network of blocking flow in Figure 10. $d(f) = 3$ in this residual graph.

Since $d(f)$ can only go up to $n - 1$ before becoming infinite (i.e., disconnecting s and t in G_f), Lemma 4.2 immediately implies that Dinic's algorithm terminates after at most n iterations. In this sense, the maximum flow problem reduces to n instances of the blocking flow problem (in layered networks). The running time of Dinic's algorithm is $O(n \cdot BF)$, where BF denotes the running time required to compute a blocking flow in a layered network.

The Edmonds-Karp algorithm and its proof effectively shows how to compute a blocking flow in $O(m^2)$ time, by repeatedly sending as much flow as possible on a single path of L_f with positive residual capacity. On Problem Set #1 you'll seen an algorithm, based on depth-first search, that computes a blocking flow in time $O(mn)$. With this subroutine, Dinic's

algorithm runs in $O(n^2m)$ time, improving over the Edmonds-Karp algorithm. (Remember, it's always a win to replace an m with an n.)

Using fancy data structures, it's known how to compute a maximum flow in near-linear time (with just one extra logarithmic factor), yielding a maximum flow algorithm with running time close to $O(mn)$. This running time is no longer so embarrassing, and resembles time bounds that you saw in CS161, for example for the Bellman-Ford shortest-path algorithm and for various all-pairs shortest paths algorithms.

5 Looking Ahead

Thus far, we focused on "augmenting path" maximum flow algorithms. Properly implemented, such algorithms are reasonably practical. Our motivation here is pedagogical: these algorithms remain the best way to develop your initial intuition about the maximum flow problem.

Next lecture introduces a different paradigm for computing maximum flows, known as the "push-relabel" framework. Such algorithms are reasonably simple, but somewhat less intuitive than augmenting path algorithms. Properly implemented, they are blazingly fast and are often the method of choice for solving the maximum flow problem in practice.