MIT 6.854/18.415: Advanced Algorithms Spring 2016

Lecture 7 – February 24, 2016

Prof. Ankur Moitra *Scribe: Andrew He, Brian Shimanuki, Jerry Wu*

1 From last time: Nearest Neighbor Search and LSH

Nearest Neighbor Search is an example of a problem whose naive runtime is exponential in dimension. We can use dimensionality reduction techniques such as locality-sensitive hashing to approximately solve this problem. Locality-sensitive hashing *exploits collisions* to achieve sublinear search time.

2 Max Flow

2.1 Motivation and setup

Definition 1 (Max Flow Problem)**.** Our input is:

- Directed graph $G = (V, A)$ with vertices V and arcs (directed edges) A.
  - Arcs $a \in A$ are ordered pairs of vertices.
- Capacity function $u : A \to \mathbb{R}^+$
  - Note that we use u here so we can reserve c for cost, which we'll look at in later lectures.
- Source s and sink t ($s, t \in V$), which are special vertices.

Informally, we would like to send as much flow as possible from s to t.

Formally, we consider all functions $f : A \to \mathbb{R}^{\geq 0}$, and we would like to maximize the "magnitude of the flow":

$$|f| := \sum_{(s,a)\in A} f(s,a) - \sum_{(a,s)\in A} f(a,s)$$

subject to the following two constraints:

- *Conservation of flow*: for all vertices $v \neq s, t$,

$$\sum_{(v,a)\in A} f(v,a) - \sum_{(a,v)\in A} f(a,v) = 0 \;;$$

- *Capacity constraint*: for all $(a, b) \in A$,

$$0 \leq f(a,b) \leq u(a,b) \;.$$

These constrain the flow to behave physically. The conservation of flow constraint ensures that except for the source and sink nodes, the total flow entering a node is equal to the total flow exiting a node. The capacity constraint limits the amount of flow that any arc can contain.

Note that t does not show up in this formulation because the net flow out of s equals the net flow into t by the conservation constraint, and we could just have easily have defined $|f|$ in terms of net flow into t.

Example 2 (Example of a Maximum-Flow Problem)**.** Consider the following flow network:

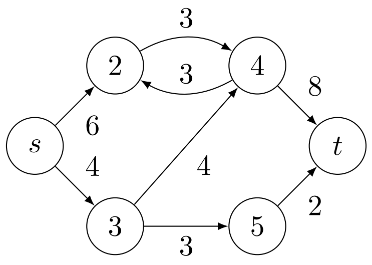

Figure 1: An example flow network

Then, a maximum flow is drawn in red.

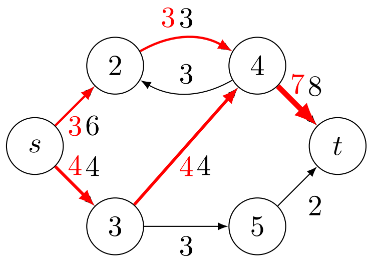

Figure 2: A maximum flow

Some example use cases of this problem include:

- Computer networks
- Transportation networks
- Matching problems
- Scheduling problems
- Partitioning problems

Fact 3. Without loss of generality, we may assume that no arc into s has flow, and no arc out of t has flow; this gives the same max flow for any graph.

2.2 Flow decomposition

We now present a way of simplifying flows.

Theorem 4 (Flow decomposition)**.** *Any s-t flow can be decomposed into at most $m = |A|$ different cycles and s-t paths.*

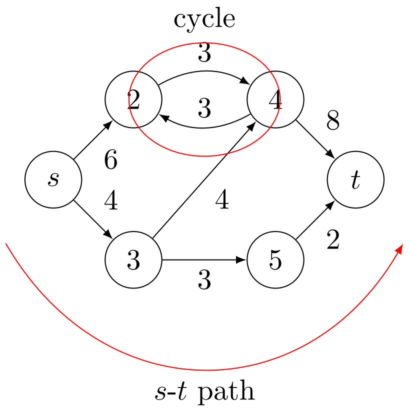

Figure 3: An example flow decomposition

Remark. s-t paths and cycles are the simplest possible flows which obey the constraints, so they form a simple set of "building blocks" for us to use. Note that we must include cycles in our decomposition even though they accomplish no net flow, as they cannot be decomposed into paths.

Proof. We induct on the number of arcs with non-zero flow.

Let a, b be any arc with $f(a, b) > 0$. Trace backwards from a and forwards from b. More specifically, because there is flow out of a along (a, b), there must be flow into a, and vice versa for b. Thus, we can trace backwards edges starting from a with positive flow or forwards edges starting at b with positive flow, until one of the following two conditions is reached:

1. We reach a cycle (which may or may not contain (a, b))
2. We trace backwards into s and forwards into t.

Note that it's possible for us to no longer be able to trace backwards from a but still be able to trace forwards from b (or vice versa) while neither of the two conditions above has been reached, but only when at least one condition has been reached will we be unable to do either.

Let w be the arcs in the cycle or s-t path, and let $\Delta(w) = \min_{(a,b)\in w} f(a, b)$.

Decrease the flow along all arcs in w by $\Delta(w)$. This is still a valid s-t flow, as we took the minimum, and since it's an s-t path or cycle the conservation constraint is still observed. But there is at least one new edge which has zero flow. Thus, our induction is complete. □

Corollary 5. *We can decompose any flow f into an equivalent flow (i.e. with the same magnitude) which is the sum of only s-t paths.*

Proof. Note that cycles contribute no net flow, so we may simply discard them from the flow decomposition of f. □

2.3 Minimum cuts

Flow decomposition gives an easy way to show that the maximum flow in a graph can be upper bounded by constructing graph *cuts*.

Definition 6 (s-t cuts)**.** We define an *s-t cut* to be a subset $S \subset V$, where $s \in S$, and $t \in V \setminus S$. The capacity of the cut is defined as

$$\text{cap}(S, V \setminus S) = \sum_{\substack{(a,b)\in A \\ a\in S, b\in V\setminus S}} u(a,b)$$

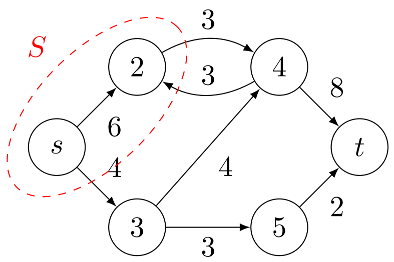

Figure 4: A capacity-7 s-t cut

Lemma 7 (Max flow $\leq$ Min cut)**.** *The maximum s-t flow through a directed graph $G = (V, A)$ is at most the minimum capacity of any s-t cut.*

Remark. Another common proof given of this involves defining the flow from a cut to be the net flow from vertices of the cut to vertices of its complement, and using the conservation law to prove this equals the net flow from s, and thus equals $|f|$ itself; then, we would use the fact that the flow from any cut is bounded by the capacity from that cut. But here we'll use flow decomposition instead.

Proof. Consider any s-t flow f, and any s-t cut $(S, V \setminus S)$. Decompose f into s-t paths and cycles, and discard the cycles, as they do not change the magnitude of the flow. Consider any s-t path which contributes p flow. Then, it uses up at least p capacity of the cut, as it must pass from S to $V \setminus S$ at some point. (Note that there's no restriction preventing a path from passing from S to $V \setminus S$ and back multiple times, but that doesn't matter since it still only contributes p flow, but uses more capacity.) □

This stems from the weak duality for linear programming, which we will explore in a later lecture.

2.4 Algorithms for Max-Flow

We first consider a greedy algorithm: while we can find an "augmenting path" from s to t, use it to increase the flow.

If w is a path, let $\delta(w) = \min_{(a,b)\in w} u(a,b) - f(a,b)$ represent the remaining capacity available along this path. Then, our algorithm is as follows:

Algorithm 1 Greedy algorithm

```
while $\exists$ an $s$-$t$ path $w$ with $\delta(w) > 0$ do
    increase the flows on each arc in $w$ by $\delta(w)$.
end while
```

Essentially, we prune out edges which are saturated, and try to find more augmenting paths. However, this greedy algorithm can become blocked too early.

For instance, if each edge of the graph below has unit capacity, and our first path is the path marked in red, we cannot find any more augmenting paths, but the maximum flow should be 2. To

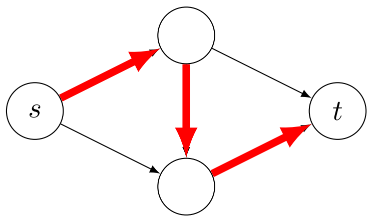

Figure 5: A counterexample for the greedy algorithm

reach the optimal, we need to "undo" the flow through the center edge and use that as part of our augmenting path.

This motivates the definition of a residual graph.

Definition 8 (Residual graph)**.** Given $G = (V, A)$ and s-t flow f, we construct the *residual graph* H with the same vertices as G, but with arcs constructed as follows:

For each arc $(a, b) \in A$, construct two arcs in H: an arc (a, b) with capacity $u(a, b) - f(a, b)$, and an arc (b, a) with capacity $f(a, b)$ (we may merge edges as appropriate and discard edges with capacity 0).

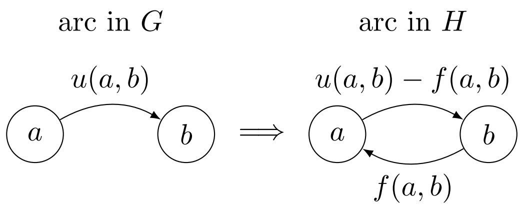

Figure 6: Construction of the residual graph

Thus, we have a refined version of our greedy algorithm.

For residual graph H, let $\Delta_H(w) = \min_{(a,b)\in w} u_H(a, b)$, where u_H denotes capacity in H.

Algorithm 2 Ford-Fulkerson [1]

```
while $\exists$ an $s$-$t$ path $w$ in $H$ with $\Delta_H(w) > 0$ do
    "increase" the flows on each arc in $w$ by $\Delta_H(w)$.
end while
```

Remark. Note that "increasing" the flows along w, where w is an s-t path in H, may actually be decreasing the flows in the corresponding arcs in G, since w may point backwards along an arc (with nonzero flow) in G. On the other hand, the net flow $|f|$ is guaranteed to increase by some amount each time.

2.5 Proof of correctness and the Max-Flow Min-Cut Theorem

We now show that Ford-Fulkerson must terminate on a maximum flow (if it in fact terminates).

Theorem 9. *For any s-t flow f with residual graph H, the following three conditions are equivalent:*

1. *f is a max flow;*
2. *There is no s-t path in H, with $\Delta_H(w) > 0$;*
3. *$|f| = \mathrm{cap}(S, V \setminus S)$ for some s-t cut.*

Proof. We prove (3) $\implies$ (1), (1) $\implies$ (2), (2) $\implies$ (3).

(3) $\implies$ (1) - See Lemma 7.

(1) $\implies$ (2) - We prove instead $\neg(2) \implies \neg(1)$, the contrapositive. This is straightforward, because if there existed such a path, we could augment along it to increase $|f|$.

(2) $\implies$ (3) - For clarity, let A_H denote the arcs in H, whereas A denotes arcs in G. On the other hand, we don't need to distinguish between vertices in G and H since they're all the same.

Consider the set S of vertices reachable in H from s, and the corresponding s-t cut $(S, V \setminus S)$. Then we claim:

- Every arc $(a,b) \in A$ with $a \in S, b \in V \setminus S$, satisfies $f(a,b) = u(a,b)$;
- Every arc $(b,a) \in A$ with $a \in S, b \in V \setminus S$, satisfies $f(b,a) = 0$;

Both claims are true because otherwise $(a,b) \in A_H$ and b would be in S.

Thus,

$$\mathrm{cap}(S, V \setminus S) = \sum_{\substack{(a,b)\in A \\ a\in S, b\in V\setminus S}} u(a,b) = \sum_{\substack{(a,b)\in A \\ a\in S, b\in V\setminus S}} f(a,b) - \sum_{\substack{(b,a)\in A \\ a\in S, b\in V\setminus S}} f(b,a) = |f|$$

by conservation of flow.

□

Corollary 10. *If Ford-Fulkerson terminates, it will have achieved a max flow.*

Proof. Ford-Fulkerson will only terminate when (2) is satisfied from Theorem 9. But by the same theorem, it will have found a max flow. □

Corollary 11 (The Max-Flow Min-Cut Theorem)**.** *The maximum flow equals the minimum cut.*

Proof. From Theorem 9, in particular the equivalence of (1) and (3), the maximum flow equals some cut. But by Lemma 7, no flow can ever exceed any cut. So the maximum flow can only equal the minimum cut. □

2.6 Analysis of runtime

Ford-Fulkerson is guaranteed to produce a maximum flow if it terminates; however, it may not terminate in finite time (or even converge to the optimum value) if the capacities are allowed to be arbitrary. We have the following counterexample:

Example 12. Consider the following graph G with capacities $u(l_1) = u(l_2) = 1$, $u(l_3) = r = \frac{\sqrt{5}-1}{2}$, and $u(\text{else}) = m$, where m is any integer at least 3. Note that $r < 1, 1 - r = r^2$.

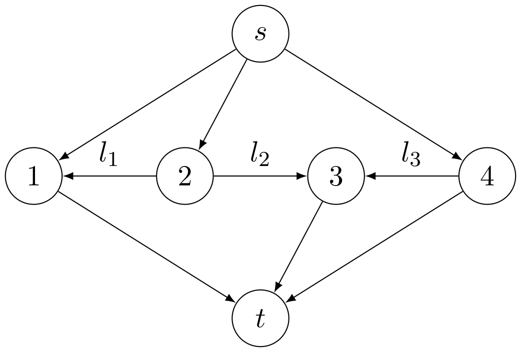

Figure 7: A bad case for Ford-Fulkerson

Then G has max flow $2m + 1$, but Ford-Fulkerson can find only $3 + 2r$ if the augmenting paths are as in Figure 8.

Path	$\Delta\|f\|$	l_1	l_2	l_3
$(s, 2, 3, t)$	1	1	0	r
$(s, 4, 3, 2, 1, t)$	r	r^2	r	0
$(s, 2, 3, 4, t)$	r	r^2	0	r
$(s, 4, 3, 2, 1, t)$	r^2	0	r^2	r^3
$(s, 1, 2, 3, t)$	r^2	r^2	0	r^3
$\vdots$				

Figure 8: Flows of the augmenting paths: the second column denotes increase in flow, and the last 3 columns denote residual capacity along those edges after that step.

The final flow converges on $1 + 2r + 2r^2 + 2r^3 + \ldots = 2\frac{1}{1-r} - 1$, which equals 3+2r for our chosen value. Thus, the Ford-Fulkerson algorithm never even approaches the correct answer even with infinitely many steps.

For integer-capacity graphs, the runtime of Ford-Fulkerson can be bounded by $O(|A|\,|f|)$, as each iteration increases the flow by at least one and takes $O(|A|)$ time to compute. (Also, for graphs with rational capacities, Ford-Fulkerson is guaranteed to terminate, as we can clear denominators.)

For an integer-capacity graph, $|f|$ can be bounded by $|A| \cdot U$ where U is the maximum capacity in G. So, we can obtain a final runtime bound that is "pseudo-polynomial", meaning that it is polynomial in the value of the inputs to the problem, but can be exponential in the size of the input representation. For example, consider the following graph with a capacity of 1 on the center arc and 2^L on every other arc. If an implementation of the Ford-Fulkerson algorithm keeps augmenting paths containing the center arc, it will take 2^L augmentations. Since capacities in this graph take at most L bits to specify, this is exponential in the number of bits required to represent the graph.

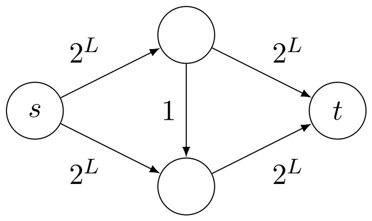

Figure 9: An integer graph which runs in pseudo-polynomial time

There are a variety of improvements we can make to reduce the runtime of this technique to polynomial, including greedily augmenting the widest flow, augmenting the shortest path flow, and capacity scaling [2]. Some of these will be covered in future lectures.

3 For next time: extensions of Max-Flow

Next time, we'll think about min-cost flow and try to do something similar: finding equivalent statements to a flow being of minimum cost.

References

[1] Ford, L. R. and Fulkerson, D. R. 1956. Maximal Flow Through a Network. *Canadian Journal of Mathematics* 8:399.

[2] Edmonds, J. and Karp, R. M. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM* 19.2:248.