CS 224: Advanced Algorithms Fall 2-14

Lecture 21 — November 11, 2014

Prof. Jelani Nelson *Scribe: Nithin Tumma*

1 Overview

In the previous lecture we finished covering the multiplicative weights method and its application to solving LP's and we introduced the maximum flow problem and the Ford-Fulkerson algorithm to solve it.

In this lecture we discuss two new methods to solve the maximum flow problem

- Scaling
- Blocking flows

that improve upon the runtime of FF.

2 Maximum Flow Problem

Recall the maximum flow problem. We are given a directed graph G with edges $E, |E| = m$, vertices $V, |V| = n$ and capacities $c(e) \in \{0, 1, \dots U\}$. We are also given a source vertex s and sink vertex t. A feasible flow is a vector f_e (for each $e \in E$) such that the flow out of the source is equal to the flow into the sink, for every $v \in V, v \notin \{s, t\}$ the incoming flow is equal to the outgoing flow. and the flow through every edge is no more than the edge's capacity. The capacity of a flow is defined to be the sum of flow units leaving s. Our goal is to find the maximal flow for G with capacities $c(e)$.

Last lecture we discussed the Ford-Fulkerson algorithm to solve max flow, whose runtime is $O(mf^*)$. Before we describe the techniques we will use to get better algorithms, we'll mention the current state of the art in max flow. James Orlin gave an $O(nm + m^{31/16} \log^2 n)$ algorithm for solving the max-flow problem in [8]. The previous best max-flow algorithm was due to King, Rao, and Tarjan ([5]) and had runtime $O(nm \log_{m/(n \log n)} n)$. Coupling these approaches (choosing which algorithm based on the sparsity of G) achieves $O(nm)$ runtime for max flow, also described in [8].

It is also possible to achieve bounds depending on U. Along these lines, [3] achieves time $O(m \cdot \min\{m^{1/2}, n^{2/3}\} \log(n^2/m) \log U)$, and [6] achieves $\tilde{O}(m\sqrt{n} \log^2 U)$ (the $\tilde{O}$ hides polylogarithmic factors).

In unit capacity graphs, [2] achieves time $O(m \cdot \min\{m^{1/2}, n^{2/3}\})$, and [7] achieves $\tilde{O}(m^{10/7})$, which is better for sufficiently sparse graphs. If one is satisfied with finding an *approximate* maximum flow (i.e. some flow with value at least $1 - \varepsilon$ times the maximum possible), then [4, 9] can do so in time $\text{poly}(1/\varepsilon) \cdot m^{1+o(1)}$.

2.1 Algorithm Runtime Terminology

Here we take the number of inputs to our algorithm to be n integers in the range $\{0, 1, \ldots, U\}$. We assume that we are working in the Word-RAM model, where standard arithmetic operations can be computed in constant time.

Definition 1. *An algorithm is* ***strongly polynomial*** *if its runtime is poly*(n).

Definition 2. *An algorithm is* ***weakly polynomial*** *if its runtime is poly*$(n, \log U)$.

Definition 3. *An algorithm is* ***pseudopolynomial*** *if its runtime is poly*(n, U).

Example: The Ford-Fulkerson algorithm has runtime $O(mf^*)$, but we have $f^* \leq (n-1)U$, so it is psuedopolynomial.

Example: Orlin's algorithm has runtime that is independent of U and is polynomial in n and m, so it is strongly polynomial.

Here, we will see one weakly and one strongly polynomial algorithm for the maximum flow problem.

3 Scaling Capacities

Recall that the Ford-Fulkerson algorithm has runtime $O(mf^*)$, where f^* is the maximum flow. We want to improve on the factor of f^*. The idea is to think of each capacity $c(e) \in \{1, 2 \ldots U\}$ as a bit integer $c(e) \in \{0,1\}^{\lceil \log_2 U \rceil}$. Then, we will "process" each $c(e)$ one bit at a time (from left to right), maintaining capacities $c'(e) = c(e)_{n \ldots (n-k)}$ (th k most significant bits of c) and a flow f' such that f' is feasible on $c'(e)$. We will actually ensure that f' is optimal at the beginning of each iteration. Then, when we finish, $c'(e) = c(e)$ and we will have $f' = f^*$. The complete scaling algorithm is given below.

3.1 Scaling Max-Flow Algorithm

- initialize $c'(e) = 0, f'_e = 0$ for all $e \in E$.
- for $k = (\lceil \log_2 U \rceil - 1) \ldots 0$
  - for each $e \in E$
    - set $c'(e) = 2c'(e)$ and $f'_e = 2f'_e$ (add trailing 0 to bit vectors of c', f')
  - for each $e \in E$
    - set $c'(e) = c'(e) + c(e)_k$ (add k'th bit of c to c')
  - augment f' such that it is a max flow for G with capacities $c'(e)$

3.2 Analysis

Claim 4. *At the beginning of every iteration of the scaling algorithm, $f' = (f')^*$, the maximum flow on G with capacities $c'(e)$.*

Proof. At the beginning of the first iteration $c'(e) = 0\ \forall e \in E$ so $(f')^* = 0 = f'$. Now, suppose that $f' = (f')^*$ before we update $c'(e)$. Then, there must exist some saturated $S - T$ cut in (G, c'). After we double $c'(e)$ and f'_e, the cut is still saturated. After we add $c(e)_k$ to c', f' may not be maximal, but it is still feasible. But then we augment f' so that it must equal $(f')^*$ on the new capacities. □

At the k'th iteration of the algorithm $c'(e)$ is the k most significant bits of $c(e)$, so after $\lceil \log_2 U \rceil$ iterations $c'(e) = c(e)$. Hence, by the claim, the output f' of the scaling algorithm will be the maximal flow for G with capacities $c'(e) = c(e)$, as desired.

The scaling algorithm runs for $O(\log_2 U)$ iterations, each iteration dominated by the time to augment f' into a maximal flow. If we use Ford-Fulkerson, we know that this time will be $O(mf^*)$. What can we say about f^*?

Claim 5. *In each iteration of the scaling algorithm, $f' = f^* \leq m$.*

Proof. At the beginning to the current iteration $f' = (f')^*$, so there must exist some saturated $S - T$ cut of (G, c'). Now, as before, after we double $c''(e), f'_e$, this cut will still have 0 capacity. After we set $c'(e) = c(e)_k$, we can have increased $c'(e)$ by at most 1. Since the cut cannot contain more than m edges, its residual capacity is $\leq m$ now, so the minimum cut in (G, c') must have capacity $\leq m$ which implies that $f^* \leq m$ (by the max-flow min-cut theorem). □

Hence, if we use Ford-Fulkerson to augment f', each iteration takes $O(m + mf^*) = O(m^2)$ steps, so we can achieve a runtime of $O(m^2 \log U)$ for maximum flow using the scaling algorithm. Hence, the scaling algorithm is weakly polynomial.

4 Blocking Flows

4.1 Setup

Recall from the previous lecture that the residual graph G_f is defined on the same edge set as G (augmented to include reverse edges) with capacities $c'(e) = c(e) - f_e + f_{\bar{e}}$. We will need some definitions before we begin.

Definition 6. *For each vertex v in a residual graph G_f, define $l(v)$ as the length of the shortest $s \to v$ path.*

Definition 7. *An edge $e = (u, v)$ is **admissible** if $l(v) = l(u) + 1$.*

By Def. 7 every admissible edge belongs to a shortest $s \to v$ path.

Definition 8. *The **level graph** $L \subset G_f$ is the subgraph of the residual graph containing the admissible edges of G.*

Definition 9. *A path is admissible if all of its edges are admissible and a flow is admissible if it decomposes into augmenting admissible paths.*

Definition 10. *f is a **blocking flow** if it is an admissible flow such that every admissible $s \to t$ path has at least one edge saturated by f (i.e., $c(e) = f_e$ for some $e \in (s \to t)$).*

Example: A max flow f^* is always a blocking flow. If it was not, we could increase the flow through some $s \to t$ path without violating capacity constraints, a contradiction. On the other hand, not every blocking flow is a maximum flow.

Note: Below, when we say that we augment along a blocking flow f', we mean we set $f := f + f'$.

Claim 11. *After augmenting along a blocking flow, the shortest path distance from $s \to t$ in the new level graph L' strictly increases. So, we have*

$$d_L(s,t) < d_{L'}(s,t)$$

Proof. Consider the level graph L before augmentation. Recall that every edge will belong to a shortest $s \to v$ path. It is not hard to see that L must be a DAG with edges $e = (u, v)$ where $u \in V_i$ and $v \in V_j$ with $i < j$. Partition the vertices into $V_i = \{v \in V; l(v) = i\}$ for $i = 1 \dots (n-1)$. Now, we must have that every edge in $e = (u, v)$ in L has $u \in V_i$ and $v \in V_{i+1}$. Suppose not; then we can find a shorter $s \to v$ path by going through e, giving $d(s, v) \leq d(s, u) + 1 = i + 1$ (since the edge 'skips' at least one level), a contradiction.

Now, consider the level graph after augmenting along a blocking flow, call it L'. The flow can be decomposed into $s \to t$ paths. Consider the distance $d_{L'}(s, t)$ between s and t after augmentation. The edges in L' are just some subset of the edges in L, the edges in $Rev(L)$ (because we added them during augmentation), and edges $e' = (u, v)$ with $l(u) > l(v)$ (i.e., backwards edges). But, since we are looking for the shortest paths from s to v, and $d(s, s) = 0 < d(s, v) = l(v)$, will never use a backwards edge when constructing a path. This is because the graph is connected, so we can always find a path from V_i to V_{i+1}. Hence, the shortest path from $s \to t$ in L' has length at least $d_L(s, t)$. Suppose $d_L(s, t) = d_{L'}(s, t)$. Then we must have some $s \to t$ path in L' that was in L. But we formed L' by augmenting along a blocking flow. Each blocking flow saturates an edge in every $s \to t$ path, so we cannot have a complete $s \to t$ path in both L and L' (at least one of the edges must have residual capacity 0). Hence, $d_L(s, t) \neq d_{L'}(s, t) \implies d_{L'}(s, t) > d_L(s, t)$. □

From this claim and because $d(s, t) \leq n$, it follows that if we find a blocking flow, augment along it, and repeat, we will terminate after $\leq n$ iterations. Furthermore, if at any point we cannot find an augmenting $s \to t$ path we must have a maximum flow.

Corollary 12. *n iterations of finding blocking flows and augmenting yield a max flow.*

This observation leads to Dinic's algorithm for computing max flow (discovered by Yefim Dinits and described in [1]):

- Find a blocking flow f' of $L \subset G_f$
- Augment our flow f along the paths of f'
- Repeat until we can no longer augment f

It follows from Corollary 12 that the time to find the maximum flow is $O(nB(n))$ where $B(n)$ is the runtime of finding a blocking flow.

4.2 Finding a Blocking Flow

Now, we will turn our attention to efficiently finding a blocking flow. The idea is to perform a depth first search from s on L, and when we find a $s \to t$, augment the flow along this path.

We will keep track of a current vertex v, which we will have reached via DFS from s.

- if $v = t$, augment along the current $s \to t$ path (add min residual capacity to flow along path), then delete 0-capacity edges
- if $v! = t$, advance to some neighbor u of v
- if no such u exists, retreat to previous vertex $(w, v) \in E$ from which we visited v and delete the edge $e = (w, v)$
- if $v = s$, terminate

Call the operation of traversing to a neighbor of v **advance**, moving to v's predecessor **retreat**, and adding flow

4.3 Analysis for Unit Capacity Graphs

Let us restrict ourselves to the simpler case of unit capacities (i.e., $c(e) \in \{0, 1\} \forall e \in E$.

Claim 13. *The above algorithm finds a blocking flow in O(m) steps.*

Proof. Recall that L is a DAG and that G_f is a unit capacity graph. We keep applying advance and retreat operations until we reach t. Now, we augment along this $s \to t$ path, which takes one unit of flow from $s \to t$. After we augment, since the residual capacity on all of these edges was 1, they become saturated and are removed from the graph. Clearly, each edge is only advanced to once (since L is a DAG). So, each edge that was encountered on a path from $s \to t$, was encountered twice (once during advancement and once during augmentation). Edges that are not on a $s \to t$ path will be advanced upon once during the DFS, and retreated upon once when t was not found, when it is deleted. Therefore, each edge can be augmented and retreated at most once. Hence, every edge is encountered a constant number of times, so the algorithm terminates in $O(m)$ steps. When the algorithm terminates, we can no longer find a $s \to t$ path in the residual graph, so the output flow must saturate at least one edge on every admissible $s \to t$ path. □

Coupling this with Corollary 12, this gives us a runtime of $O(nm)$ for max-flow. Recall that this is a strongly polynomial algorithms since the runtime does not depend upon U.

Now, how many times do we actually have to find blocking flows in the above algorithm? We have the upper bound of n, but we can do better. The idea is to observe that in unit capacity graphs each max flow can be decomposed into edge disjoint $s \to t$ path, each pushing one unit of flow. By

Since $d(s,t)$ increases during every iteration of Dinic's algorithm, after d iterations we have $d(s,t) \geq d$. We can decompose any max flow into $s \to t$ paths, each of which must have length $\geq d$.

Claim 14. *If $d(s,t) = d$ the residual max flow is at most m/d in unit capacity graphs.*

Proof. Decompose the residual max flow into $s \to t$ paths. Since G_f is a unit capacity graph, each $s \to t$ path can only support one unit of flow. Hence, each edge is part of at most one path in the decomposition, so there can be at most m/d paths of flow. Each path delivers exactly one unit of flow from $s \to t$ so the residual max flow is at most m/d as desired. □

Since every iteration of our algorithm that augments along a blocking flow increases the current flow by 1, after d iterations of the algorithm we will need at most m/d more iterations of blocking flow augmentations. Then the total number of times we need to find blocking flows is bounded above by

$$d + \frac{m}{d}$$

steps, which is minimized when $d = \sqrt{m}$ to give $2\sqrt{m} = O(\sqrt{m})$ steps. Since it takes us $O(m)$ time to find blocking flows, this approach gives us a runtime of $O(m^{3/2})$ for max flow in unit capacity graphs.

4.4 General Capacity Graphs

Note that we can use the same algorithm to find blocking flows in general capacity graphs.

Claim 15. *Using Dinic's algorithm, we can find blocking flows in $O(nm)$ time in general capacity graphs.*

Proof. Our runtime will be the sum of the time that we spend in each of the three basic operations.

- Retreat - Every time we perform a retreat, we delete one edge. Therefore, we can perform at most m retreat operations, each of which take constant time.
- Augment - Every time we augment along a path, we must saturate at least one edge in the residual graph (the edges with minimal capacity). We delete all of the saturated edges after this step, so there are most m augment operations. Since each augment takes time proportional to the length of the $s \to t$ path, and the longest such path has length n, we spend at most $O(nm)$ time augmenting.
- Advance - Since we have only n vertices, after n advances, we must have at least one retreat or augment step. Each retreat/augment step results in the removal of at least one edge. Therefore, we call advance at most $O(nm)$ times, and each iteration takes constant time.

This gives a total runtime of $nm + nm + m = O(nm)$, as desired. The correctness follows from the same reasoning as before; at termination we can find no augmenting $s \to t$ paths, so we must be saturating at least one edge in each $s \to t$ path. □

The claim, coupled with Corollary 12 shows that we can find max flow in general capacitated graphs in $O(mn^2)$ steps. We can actually do better. Sleator and Tarjan, in [10] give an $O(mn \log n)$ max flow algorithm based on dynamic link/cut trees.

References

[1] Yefim Dinitz. Algorithm of solution to problem of maximum flow in network with power estimates. *Doklady Akademii nauk SSSR.*, 11:1277-1280.

[2] Shimon Even, Robert Endre Tarjan. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.* 4(4): 507–518, 1975.

[3] Andrew Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM (JACM).*, 45(5):783-797, 1998.

[4] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, Aaron Sidford. An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations. *SODA*, pages 217–226, 2014.

[5] Valerie King, Satish Rao, and Robert Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms.*, 17(3):447-474, 1994.

[6] Yin Tat Lee, Aaron Sidford. Following the Path of Least Resistance : An $\tilde{O}(m\sqrt{n})$ Algorithm for the Minimum Cost Flow Problem. *CoRR* abs/1312.6713, 2013.

[7] Aleksander Madry. Navigating Central Path with Electrical Flows: From Flows to Matchings, and Back. *FOCS*, pages 253–262, 2013.

[8] James Orlin and Satish Rao. Max flows in $O(nm)$ time, or better. *Proceedings of the forty-fifth annual ACM symposium on Theory of computing.*, ACM. 765-774, 2013.

[9] Jonah Sherman. Nearly Maximum Flows in Nearly Linear Time. *FOCS*, pages 263–269, 2013.

[10] Daniel Sleator and Robert Tarjan. A data structure for dynamic trees. *In Proceedings of the thirteenth annual ACM symposium on Theory of computing.*, ACM. 17(3):114-122, 1981.