CS 270 Algorithms **Spring 2021**

Lecture 18: Approximate Maximum Flow in Undirected Graphs

Lecturer: Prasad Raghavendra Scribe: Ankit Agarwal and Magzhan Gabidolla

18.1 Summary

The goal of this lecture is to find a way to approximate maximum flow in an undirected graph in near linear time. In order to do this, we will first review the multiplicative weights (MW) algorithm and analyze its run-time in terms of the gain bound. Then, we will look at two separate oracles for max-flow namely the shortest path oracle and the electric flow oracle.

By using these two oracles, and our Laplacian solver from the previous lecture, we will be able to construct $O(m^{3/2})$ algorithm to solve the max-flow problem approximately in undirected graphs.

18.2 The Max Flow Problem

Recall that a flow f on graph $G = (V, E)$ of value F from starting vertex s to ending vertex t is a mapping $E \to \mathbb{R}$. We denote the flow on edge $e = (i, j)$ as $f_e = f_{(i,j)}$. Flows also have the constraint that for all $j \in V$:

$$\begin{cases} \sum_{(i,j)\in E} f_{(i,j)} = 0 & j \notin \{s, t\} \\ \sum_{(i,j)\in E} f_{(i,j)} = -F & j = s \\ \sum_{(i,j)\in E} f_{(i,j)} = F & j = t \end{cases}$$

.

We note that flows are inherently anti-symmetric, that is $f_{(i,j)} = -f_{(j,i)}$.

For the max flow problem, we also require each edge (i, j) to have a capacity $c_{(i,j)}$ such that the flow across the edge cannot exceed $c_{(i,j)}$. The max-flow problem can now be formulated as the maximum value of F that is a flow and satisfies the edge capacity constraints.

In particular, we can write max-flow as the following linear program (LP).

$$
\begin{aligned}
\text{maximize} \quad & F \\
\text{subject to} \quad & \sum_{(i,j)\in E} f_{(i,j)} = 0 \qquad j \notin \{s,t\}, \\
& \sum_{(i,j)\in E} f_{(i,j)} = -F \qquad j = s, \\
& \sum_{(i,j)\in E} f_{(i,j)} = F \qquad j = t, \\
& f_{(i,j)} \leq c_{(i,j)} \qquad \forall (i,j) \in E
\end{aligned}
$$

We solve this problem by fixing values of F and determining if a feasible solution to the above LP exists using multiplicative weights. This type of LP (without a minimize/maximize constraint) is known as a **feasible LP**. We then do a **binary search** on the value of F until we are within ϵ of our desired value. Since this only adds logarithmic complexity to our algorithm, this does not affect our overall runtime in terms of $\tilde{O}$.

18.3 Review of Multiplicative Weights

The multiplicative weights algorithm involves two primary entities. The MW algorithm itself, and an oracle adversary. The algorithm and adversary play a game at every time-step t. The algorithm involves a vector of **experts** indexed from 1 to m. At time t, the algorithm presents the adversary with a **weight vector** $\mathbf{p}^{(t)} = (p_1^{(t)} ... p_m^{(t)})$ which represents a probability distribution over the experts. The adversary, in turn, looks at the weights and provides a **gain vector** $\mathbf{g}^{(t)} = (g_1^{(t)} ... g_m^{(t)})$. At time-step t, the algorithm incurs a total gain of $\langle \mathbf{g}^{(t)}, \mathbf{p}^{(t)} \rangle$.

At $t = 0$, the algorithm starts off with a uniform distribution over the experts. Then, as time goes on, the algorithm follows the given update rule. Note there are many update rules but we will use this one. This corresponds to the **Hedge Algorithm**:

$$p_i^{(t+1)} = p_i^{(t)} \exp\left(\epsilon \cdot g_i^{(t)}\right)$$

.

Given this update rule, the following theorem is true:

Theorem 18.1. *Given that the gains* $g_i^{(t)} \in [-\gamma, \rho]$ *for all* t, i*, and for all* ϵ*, there exists an integer* T *such that:*

$$\frac{1}{T}\sum_{t=1}^{T}\langle \mathbf{p}^{(t)}, \mathbf{g}^{(t)}\rangle \geq \max_{i\in\{1\ldots m\}} \frac{1}{T}\sum_{t=1}^{T} g_i^{(t)} - \epsilon$$

and $T = O\left(\frac{\gamma\rho\ln(m)}{\epsilon^2}\right)$.

18.4 Solving an LP with Multiplicative Weights

Now that we have reviewed the MW algorithm, let us understand how this algorithm relates to max-flow. Since max-flow can be treated as a feasible LP, we will discuss a general way to solve LP's using the MW algorithm.

In our case, we will divide our constraints into two types. Flow constraints which determine a flow, and capacity constraints, which ensure that each edge stays under capacity. For our purposes, the flow constraints will be our *easy* constraints and our capacity constraints are our *hard constraints*. The easy constraints will always be satisfied by our oracle, but the hard ones will not always be.

The setup is to run the MW algorithm. The idea is that at each stage, the MW algorithm gives us a probability distribution over our hard constraints. This probability distribution creates a single new constraint in place of the original hard constraints. We note that we give the new constrain $\epsilon > 0$ room. An example is shown below:

Example: Suppose our constraints are $c_1 - 8 \leq 0$ and $c_2 - 4 \leq 0$. Then a probability distribution of $\mathbf{p}^{(t)} = (0.25, 0.75)$ would give the constraint that $0.25(c_1-8)+0.75(c_2-4) \leq 0$ or that $0.25c_1+0.75c_2-5 \leq \epsilon$.

Our oracle adversary now only has to find a feasible solution in the LP with the easy constraints and the single hard constraint. We call this single hard constraint the **average gain** of the algorithm at that time-step. We note that since the average gain is by nature a linear combination of the individual hard constraints, failure to find a feasible solution for this LP implies that the original LP is not feasible.

Once the oracle finds a feasible solution, it is able to return the values for each individual hard constraint as its gain vector. We note that in general, each gain component which is a linear constraint can be written as $g_i^{(t)} = \langle a_i, \mathbf{x}^{(t)}\rangle - b_i$ where $\mathbf{x}^{(t)}$ is the vector of variables of the LP at time-step t. The corresponding constraint to this gain component is then $\langle a_i, \mathbf{x}^{(t)}\rangle - b_i \leq 0$. For example, using the constraints in the example above, if the oracle found the solution $c_1 = 0$, $c_2 = 6$, which satisfy the simplified LP, it would return $c_1 - 8 = -8$ and $c_2 - 6 = 2$ or $(-8, 2)$ as its gain vector.

We are now ready to do some analysis. For each t we know that the oracle is able to find a feasible solution. So by the multiplicative weights algorithm:

$$\epsilon \geq \langle \mathbf{p}^{(t)}, \mathbf{g}^{(t)} \rangle \quad \forall t$$

$$\epsilon \geq \frac{1}{T} \sum_{t=1}^{T} \langle \mathbf{p}^{(t)}, \mathbf{g}^{(t)} \rangle \quad \text{average over all } t$$

$$\geq \max_{i \in \{1 \ldots m\}} \frac{1}{T} \sum_{t=1}^{T} g_i^{(t)} - \epsilon \quad \text{By Multiplicative Weights Theorem}$$

$$2\epsilon \geq \frac{1}{T} \sum_{t=1}^{T} g_i^{(t)} = \frac{1}{T} \sum_{t=1}^{T} \langle a_i, \mathbf{x}^{(t)} \rangle - b_i = \langle a_i, \frac{1}{T} \sum_{t=1}^{T} \mathbf{x}^{(t)} \rangle - b_i \quad \forall i \in \{1 \ldots m\}$$

.

In other words, $\sum_{t=1}^{T} \mathbf{x}^{(t)}$ solves each linear constraint up to 2ϵ, which since ϵ is made as small as possible, solves the LP. This demonstrates how the multiplicative weights algorithm can solve linear programs given a proper oracle that can solve the simplified program.

18.5 Setting up MW for Max-Flow

We have reduced solving max-flow to an LP and reduced solving and LP via multiplicative weights to finding an oracle that that given a probability distribution $\mathbf{p}^{(t)}$ over the edges, solves the LP below:

$$\begin{aligned}
\text{maximize} \quad & 1 \\
\text{subject to} \quad & \sum_{(i,j) \in E} f_{(i,j)} = 0 \qquad j \notin \{s, t\}, \\
& \sum_{(i,j) \in E} f_{(i,j)} = -F \qquad j = s, \\
& \sum_{(i,j) \in E} f_{(i,j)} = F \qquad j = t, \\
& \sum_{(i,j) \in E} p_{(i,j)}^{(t)} f_{(i,j)} \leq \sum_{(i,j) \in E} p_{(i,j)}^{(t)} c_{(i,j)} + \epsilon
\end{aligned}$$

For simplicity, we keep $c_{(i,j)} = 1$ for our edges. Thus our final constraint becomes:

$$\sum_{(i,j)\in E} p_{(i,j)}^{(t)} f_{(i,j)} \leq 1 + \epsilon$$

18.6 The Shortest Path Oracle

We now turn to a simple oracle that solves this linear program. In general, we can think about this program as finding a way to route F flow through the graph with the average gain constraint. Our algorithm will be as follows:

Algorithm 1: Shortest Path Oracle Algorithm

Result: Finds a flow on graph G given a probability distribution $\mathbf{p}^{(t)}$ over the edges
treat p_e as an edge weight;
find the shortest path P in graph G with these weights;
route F units of flow on this path;

Let P be the recovered shortest path. Then if $\text{len}(P) \leq \frac{1}{F}$, then the total cost over each edge which is $\sum_{e\in E} f_e p_e = \sum_{e\in P} f_e p_e = \sum_{e\in P} F p_e =\leq \frac{1}{F} \cdot F = 1$. So we have found a path that satisfies the constraint. Otherwise if $\text{len}(P) \geq \frac{1}{F}$, then since every flow is a linear combination of paths, if we were to route F units of flow from s to t by the minimality of P, it is at least the same as routing F units of flow from s to t along the shortest path P which is such that $\sum_{e\in P} f_e p_e = \sum_{e\in P} F p_e \geq \frac{1}{F} \cdot F = 1$. Thus no flow satisfies the constraint and we have proved the in-feasibility of the LP.

Thus this oracle solves the above LP.

18.7 Runtime with SPT Oracle

To know the runtime we need to estimate the two parameters γ and ρ in the MW algorithm.

$$\gamma = \min \text{violation} = \min_i \left(\langle \mathbf{a_i}, \mathbf{x}\rangle - b_i\right) = \min_e f_e - 1 > -1 \quad \text{(flows are nonnegative)}$$

So $\gamma = -1$.

$$\rho = \max \text{violation} = \max_e f_e - 1 \approx F - 1 \approx F$$

because in the worst case we can send F units of flow along a single edge. Since all edge capacities are one, $F \leq m$. So $\rho = m$. The number of iterations in MW algorithm is therefore:

$$O\Big(\frac{\gamma\rho \ln m}{\epsilon^2}\Big) = O\Big(\frac{(1)(m)\ln m}{\epsilon^2}\Big) = \tilde{O}(m)$$

At each iteration SPT Oracle takes linear time, so the total runtime is $\tilde{O}(m^2)$.

18.8 Electric Flow Oracle

One disadvantage of SPT oracle is its high width $\rho = m$, because it uses only a single path. We want to find flow which spreads out the mass more across the graph. In other words, we want to improve the ρ quantity, so that the runtime is better. This can be achieved by constructing an oracle using electrical flow. Note that the computation of electrical flow is fast ($\tilde{\Theta}(m)$) and it minimizes the energy. We use $r_e = p_e + \epsilon/m$ resistances to compute $\hat{f}$ electrical flows.

Suppose f^* is the optimal feasible maximum flow.

$$\begin{aligned}
\mathtt{Energy}(f^*) &= \sum_e (f_e{}^*)^2 r_e \\
&\le \sum_e (1)^2 (p_e + \frac{\epsilon}{m}) \quad \text{(capacity constraints)} \\
&\le \sum_e p_e + \frac{\epsilon}{m} m \\
&= 1 + \epsilon
\end{aligned}$$

Because electrical flow minimizes the energy, $\mathtt{Energy}(\hat{f}) \le \mathtt{Energy}(f^*) \le 1 + \epsilon$. So we have

$$\begin{aligned}
&\sum_e (\hat{f}_e)^2 r_e \le 1 + \epsilon \\
&\sum_e (\hat{f}_e)^2 (p_e + \frac{\epsilon}{m}) \le 1 + \epsilon
\end{aligned} \tag{18.1}$$

We can use this inequality to bound the width and the average gain. From eq. 18.1 it follows that

$$\forall \text{ edges } (\hat{f}_e)^2 \cdot \frac{\epsilon}{m} \le 1 + \epsilon \implies \hat{f}_e \le \sqrt{\frac{\epsilon}{m}}$$

Therefore $\rho = \text{width} = \max_e \hat{f}_e = \sqrt{\frac{\epsilon}{m}}$ (an improvement from m to $\sqrt{m}$). Now, bounding the average gain:

$$\begin{aligned}\sum_e p_e f_e &\le \Big(\sum_e p_e\Big)^{1/2}\Big(\sum_e p_e f_e^2\Big)^{1/2} \quad \text{(Cauchy–Schwarz)}\\ &\le (1)\cdot(1+\epsilon) \quad \text{(using eq. 18.1)}\\ &\le 1+\epsilon\end{aligned}$$

So the electrical flow oracle satisfies the average gains constraint, and it has better width.

18.9 Runtime with Electric Flow Oracle

The number of iterations in MW algorithm is:

$$O\Big(\frac{\gamma\rho \ln m}{\epsilon^2}\Big) = O\Big(\frac{1\sqrt{m}\ln m}{\epsilon^2}\Big) = \tilde{O}(\sqrt{m})$$

At each iteration we need to solve the Laplacian system to calculate electrical flow, which takes $\tilde{O}(m)$ time. So the total runtime is:

$$\tilde{O}(\sqrt{m})\cdot\tilde{O}(m) = \tilde{O}(m^{3/2})$$

18.10 Extensions of the algorithm

When this algorithm was discovered in 2010, there already existed an algorithm with the same runtime from 1990s which works even for directed graphs (the techniques of electrical flow only work for undirected graphs). However, using the same techniques we can obtain $O(m^{4/3})$ algorithm, which improves upon all the other combinatorial techniques. But the algorithm is approximate and only works for undirected graphs. The main idea of it is as follows: we monitor the maximum violation ρ in electrical flow, and if $\exists\, e \in E\ \ \hat{f}_e > m^{(1/3)}$ then we delete that edge, and recalculate the electrical flow, and continue the algorithm. By doing so, we enforce the bound on $\rho < m^{(1/3)}$. The key to the proof is to show that deletions do not affect significantly the max flow and that there are not too many deletions. The proof uses the upper bound on the effective resistance R_{eff} between s and t to show that the change in R_{eff} from deletions cannot exceed that bound.

There now exist near linear time algorithms for approximate max flow in undirected graphs. Those algorithms do not use electrical flow, but many ideas are derived from it. The critical idea behind many algorithms for max flow can be stated in the realm of gradient descent as:

$$\min_{x} ||Bx||_{\infty}$$

It is known that gradient descent can be accelerated with some form of preconditioning. When the matrix B comes from graph problems, then we can use combinatorial objects for preconditioning. Near linear time algorithms are based on some form of combinatorial preconditioning and some form of gradient descent.