CME 305: Discrete Mathematics and Algorithms
Instructor: Professor Aaron Sidford (sidford@stanford.edu)
January 25, 2018

Lecture 6 - Maximum Flow & Applications[1]

In this lecture we continue our discussion of the maximum flow problem. We show how to obtain faster algorithms than those achieved in the last lecture and we discuss several applications of the maximum flow problem.

1 Recap

Let's briefly recall the problem from last lecture. We have directed graph $G = (V, E)$, a source $s \in V$, sink $t \in V$, and edge capacities $u \in \mathbb{R}^E_{\geq 0}$. Our goal is to compute $f \in \mathbb{R}^E_{\geq 0}$ that is a s-t flow, meaning for all $v \notin \{s, t\}$ we have that the net flow out of v or the imbalance at v is 0 i.e.

$$\text{im}(f, v) = \sum_{e=(v,u)\in E} f(e) - \sum_{e=(u,v)} f(e) = 0$$

and that is feasible, meaning it also meets the capacity constraints, $f(e) \in [0, u_e]$ for all $e \in E$. Moreover, we wish to compute a maximum s-t flow, meaning a feasible s-t flow of maximum value, $v(f) = \text{im}(f, s)$.

Last class we saw the following augmenting flow meta-algorithm.

Algorithm 1 Augmenting Flow Meta Meta Algorithm

Input: capacitated graph $G = (V, E, u)$ and vertices $s, t \in V$
Let $f_0 = \vec{0}$ and $i = 0$
while t is reachable from s in G_{f_i} **do**
  Compute s-t flow g_i in G_{f_i}
  Let f_{i+1} be s-t flow of value $v(f_i) + v(g_i)$ by combining f and g
end while
Return f_i as a maximum flow and R_i the set of vertices reachable from s in G_{f_i} as the minimum cut.

In each iteration i of this algorithm there is a flow f_i and a residual graph G_{f_i} is constructed by for every edge $e \in E$ decreasing the capacity of e by $f_i(e)$ and adding an edge in the reverse direction of e with capacity $f_i(e)$. A flow in this residual graph, g_i, is computed and we update our flow f_i by essentially adding g_i. As we saw the residual graph was constructed precisely so that the set of feasible flows in the residual graph correspond to the set of flows that can be added to f_i while preserving feasibility.

We saw by the linearity of flow value that each iteration $v(f_{i+1}) = v(f_i) + v(g_i)$ and that f_i stays feasible throughout the algorithm. Furthermore, we saw that the value of s-t cuts in G_{f_i} is exactly its value in G minus $v(f_i)$ and therefore, when s cannot reach t in G_{f_i} then the set of vertices reachable from s in G_{f_i}, denoted R_i, has capacity 0 and therefore is a s-t cut in G with value equal to $v(f_i)$. Since of a flow is at most the value of any cut this proved that the flow and cut returned by the algorithm were optimal.

Consequently, if the g_i picked in every iteration of the meta-algorithm is an arbitrary s-t path this algorithm, known as Ford Fulkerson, solved the maximum flow problem in $O(mv_*)$ time. Moreover, this algorithm proved that if the capacity are integer then there is always a maximum flow where the flow values are integer. In this lecture we show how to improve.

[1] These lecture notes are a work in progress and there may be typos, awkward language, omitted proof details, etc. Moreover, content from lectures might be missing. These notes are intended to converge to a polished superset of the material covered in class so if you would like anything clarified, please do not hesitate to post to Piazza and ask.

2 Improving the Running Time for Maximum Flow

So how do we improve upon Ford Fulkerson? Well, one obvious possible inefficiency in Ford Fulkerson is that the s-t path found in the residual graph was completely arbitrary. A natural thing we could do would be to pick the s-t path that allows us to send as much flow as possible. As we saw in early lectures, this path will only approximate the maximum flow, but so long as it approximates it well we can use this to obtain a better bound on the number of iterations of our meta algorithm.

Informally, for a simple s-t path $P = (e_1, ..., e_k)$ with $e_i \in E$ we say that the value of the path is the maximum value of α such that sending α-units of flow along the path is feasible. Formally, we say $p \in \mathbb{R}^E$ is the *indicator vector of the path* $P = (e_1, ..., e_k)$ if for all $e \in E$ we have

$$p(e) = \begin{cases} 1 & e = e_i \text{ for } i \in [k] \\ 0 & \text{otherwise} \end{cases}.$$

Now, we say αp is the flow corresponding to sending α units on path P and we say the the value of the path P is the max α such that αp is feasible. Clearly αp is a s-t flow for all α and it is feasible when $\vec{0} \leq \alpha p \leq u$, entrywise. Consequently, αp is feasible when $0 \leq \alpha \leq \min_{i \in [k]} u_{e_i}$. In other words, given a path the most flow we can send along that path while having a feasible flow is the minimum capacity of an edge on the path and we call that the value of the path.

Consequently, if we want to find the path that lets us send the most flow, since $v(\alpha p) = \alpha \cdot v(p)$ by linearity, we see that we want to compute the s-t path where the minimum capacity of an edge on the path is largest. This is known as the "fattest path" or "widest path" and we call setting g_i in the meta-algorithm to be the widest path in G_{f_i} the "widest augmenting path" heuristic.

Now before, we analyze this heuristic, we first note that we can easily compute the widest path in nearly linear time.

Lemma 1. *We can compute the widest path in time $O(m \log n)$.*

Proof. Note that the value of the widest path is maximum value of α for which there is a s-t path using only edges e which have capacities $u_e \geq \alpha$. To find this value of α we can simply perform binary search. First we sort the edge capacities which can be done in $O(m \log n)$ time. Let $e_1, ..., e_m$ be the edges sorted in order of increasing capacity and let $u_1, ..., u_m$ be there capacities. We then check if there is a s-t path in the graph consisting of edges with capacities larger then $u_{m/2}$, which take $O(m)$ time. If so we know that optimal value of α lies in $[u_{m/2}, u_e]$ if not we know the optimal value of α lies in $[0, u_{m/2}]$ repeating by querying the midpoint of the next interval and so on, we halve the possible values of $\alpha$and thus find the optimal value of α in $O(m \log m) = O(m \log n)$ time (assuming the graph is simple). Once we have the optimal value of α we find any s-t path in the graph of edges with capacities at least α and we obtain the desired path. □

Now to analyze how well this heuristic does, let's reason a bit more about the structure of s-t flows. We want to show that the maximum value s-t flow contains a significant amount of flow relative to the value of the optimal flow. We will do this by showing that any s-t flow, including the maximum one can be decomposed into a bounded number of paths and a *circulation* (a s-t flow of value 0, i.e. $f \in \mathbb{R}^E$ with $\text{im}(f, v) = 0$ for all $v \in V$.

Lemma 2. *If $f \in \mathbb{R}^E_{\geq 0}$ is a s-t flow then $f = \sum_{i \in [k]} \alpha_i p_i + f_0$ where each p_i is the indicator of a s-t path, $\alpha_i \geq 0$ for all $i \in [k]$, $\sum_{i \in [k]} \alpha_i = v(f)$, $f_0 \in \mathbb{R}^E_{\geq 0}$ is a s-t flow with $v(f_0) = 0$, and $k \leq m$. Moreover, we can compute the α_i,p_i, and f_0 in $O(m^2)$ time.*[2]

[2] The running time can be improved to $O(mn)$ but we leave the ingredients for this to homework.

Proof. Consider the capacitated graph $G = (V, E, u)$ where $u_e = f_e$ for all $e \in E$ (and all edges with 0 capacity are removed). Either $v(f) = 0$ and the claim follows trivially with $k = 0$ and $f_0 = f$ or otherwise we know there must be a s-t path in G. Let p_1 be any simple s-t path in G and let α_1 be the maximum value such that $\alpha_1 p_1 \leq u$ entrywise. Decrease u_e by $\alpha_1 p_1(e)$ for all e and repeat on $f - \alpha_1 p_1(e)$. We take the sum of the $\alpha_1 p_1$ compute this way as the p_i and the final flow of value 0 as the f_0. Note that every time we subtracted $\alpha_1 p_1(e)$ the value of f decreased by $\alpha_1 p_1(e)$ and at least one edge had its flow set to 0 (by the maximality of α_1). Consequently, this procedure terminates in m iterations and $k \leq m$ and $\alpha_i \geq 0$ and $\sum_{i \in [k]} \alpha_i = v(f)$ as desired. Note the algorithm applied here can easily be executed in $O(m^2)$ time. □

Why did the algorithm in the above proof work ? We know that such a greedy scheme does not in general compute a maximum s-t flow, but it did when we ran in it in this proof. The key here is that we ran the algorithm on the graph induced by a s-t flow itself and unlike with a graph, there is no way for a path to cross a minimum s-t cut more than once. In other words we showed that greedy does work get a flow equal to the out-degree of s minus the in-degree of s provided that in-degree equals out-degree for every other vertex in a graph. Note that nothing too surprising is happening here as this is only gauranteeing a flow of value 0 on the directed graph induced by an undirected graph.

Now in the last lemma we showed that a s-t flow is essentially the sum of at most m paths and a circulation. However, the widest augmenting path picks the best path and thus it should do as well as one of these m. We show this in the following:

Lemma 3. *The value of the widest augmenting path is at least v_*/m.*

Proof. Let W be the value of the widest augmenting path and let f_* be any maximum s-t flow. By the previous lemma we know that $f_* = \sum_{i \in [k]} \alpha_i p_i + f_0$ where each p_i is the indicator of a s-t path, $\alpha_i \geq 0$ for all $i \in [k]$, $\sum_{i \in [k]} \alpha_i = v(f_*)$, and $k \leq m$. Consequently,

$$v_* = v(f_*) = \sum_{i \in [k]} \alpha_i \leq kW \leq mW$$

since each $\alpha_i \in [0, W]$ by virtue of the fact that $\alpha_i p_i$ is feasible and the widest augmenting path is of the form αp for the largest possible α where αp is feasible. □

Using this we can obtain an improved running time for computing maximum flow in many settings.

Lemma 4. *If $G = (V, E, u)$ is a capacitated graph with integer capacities then the meta-algorithm with widest augmenting path heuristic can be used to compute a maximum flow in time $O(m^2 \log v_* \log n)$ and if the maximum capacity of the graph is U it can compute it in time $O(m^2 \log(nU) \log(n))$.*

Proof. For all i in the meta algorithm let $G_i = G_{f_i}$ and let $v(G_i)$ be the value of the maximum s-t flow in G_i.We know that $v(G_i) = v(G_0) - v(f_i) = v_* - v(f_i)$. Consequently, by the previous lemma $v(g_i) \geq \frac{1}{m}[v_* - v(f_i)]$ and we have that

$$v_* - v(f_{i+1}) - = v_* - v(f_i) - v(g_i) \leq \left(1 - \frac{1}{m}\right)[v_* - v(f_i)]$$

therefore by induction, for all i we have

$$v_* - v(f_i) \leq \left(1 - \frac{1}{m}\right)^i [v_* - v(f_0)] \leq \exp\left(-\frac{i}{m}\right) v_* \,.$$

Therefore once $i > m \log(v_*)$ we have $v_* - v(f_i) < 1$. However, capacities stay integer throughout this algorithm so when this happens $v_* - v(f_i) = 0$. Thus the algorithm terminates in $O(m \log(v_*))$ iterations and the result follows. □

Note that this is a weakly polynomial running time. If the capacities are written in binary we still obtain a polynomial time algorithm as we depend on U logarithmically. This improves upon our previous running which was technically not polynomial.

How do we obtain a strongly polynomial time algorithm? That is one that does not depend on the capacities at all. We will not prove it here, but if instead of picking the widest augmenting path, we pick the path with the fewest number of edges then it can be shown that every m iterations the number of edges in the shortest s-t path in the residual must go up by at least one. Since a simple path cannot have length larger then n and we can compute the path with the fewest edges in $O(m)$ time by BFS this implies we can solve the maximum flow problem in strongly polynomial time, $O(m^2n)$.

Faster algorithms can be obtained by augmenting by more than a single path in each iteration. For example, if the augmenting flow is chosen by applying something greedy until there are no more paths to be found, the resulting flow is known as a blocking flow. It is outside the scope of this class, but there are algorithms which can compute a blocking flow in nearly linear time. Applying to carefully chosen subgraphs can yield even faster algorithms. For a long time the fastest algorithms for maximum flow were strongly polynomial running times of the form $\tilde{O}(mn)$ and weakly polynomial running time of $\tilde{O}\left(m \cdot \min\{m^{1/2}, n^{2/3}\} \log U\right)$ due to Goldber and Rao in 98. These were improved recent and now the fastest strongly polynomial running time is due to Orlin and is $O(mn)$ and the fastest weakly polynomial time algorithm is $\tilde{O}(m\sqrt{n}\log^{O(1)} U)$ due to Yin Tat Lee and myself. There is also an almost weakly polynomial time algorithm that runs in $\tilde{O}(m^{10/7}U^{1/7})$ due to Madry and it is an active area of research to improve these even further.

3 Applications

In the remainder of this lecture we discuss a few applications of maximum flow. There are many more applications than those listed here and the reading in Kleinberg and Tardos has many.

3.1 Disjoint Paths

The first application is a rather natural one. We are given an unweighted directed graph $G = (V, E)$ and we wish to compute the maximum number of edge disjoint paths from s to t. We also want to compute the fewest number of edges to remove to make it so s cannot reach t.

To solve each of these we can just compute a maximum s to t flow in the graph where all capacities are 1. We can get a flow where the flow values are integral, i.e. just 0 and 1 and decompose it into paths using the algorithm discussed for decomposing flows into paths. This gives a number of disjoint paths which is equal to the maximum flow and therefore optimal (since if a graph has k disjoint $s-$t paths the maximum s-t flow value is at least k, since the k paths constitute a flow). Furthermore, we know the edges in the minimum cut are the fewest edges to remove to make it so s cannot reach t.

3.2 Project Selection

Now let's see a more sophisticated application of maximum flow to solve a problem that may not look like the maximum flow problem on the surface. Consider the following project selection problem. We have projects $p_1, ..., p_n$. Each project has a reward r_i for completing it. This reward may be either positive or negative, depending on whether we are truly rewarded or penalized for completing the project. However, each project also has a set of required projects $D_i \subseteq [n]$ where for each project p_i to complete it we must also complete p_j

for all $j \in D_i$. Now subject to requirement constraints we want to compute the set of projects to complete that yield a maximum total reward.

The motivation behind this example is that we may have some activities we do not want to do, i.e. they have negative reward, but they enable us to do something with positive reward and we wish to optimize the total reward.

Now, on the surface the difficulty in modeling this by maximum flow is that the rewards can be positive or negative. We have not discussed anything to handle negative costs, we only have positive capacities. However, we fix this by being careful about connecting edges to s or to t.

We show how to use maximum flow to solve this problem as follows. We will construct a graph where minimizing over s-t cuts corresponds to maximizing over the reward of valid project selections.

Create a graph with vertices $p_1, .., p_n$ as well as an additional vertex s and a vertex t. For each $i \in [n]$ and $j \in D_i$ add an edge (p_i, p_j) with infinite capacity. Further, for each $i \in [n]$ if $r_i \geq 0$ add a (s, p_i) edge of capacity r_i and if $r_i \leq 0$ add a (p_i, t) edge of capacity $-r_i$.

Now consider the capacity of a cut s-t cut S, i.e. $u(S)$. Now if for some $j \in D_i$ for $i \in [n]$ it is the case that $p_i \in S$ but $p_j \notin S$ then $u(S) = \infty$. Consequently, a cut is of finite value if and only if the projects in it respect the dependence constraints. Further, there is always a s-t cut of finite value since we can $S = \{s\}$. Now supposing the cut value is finite, then

$$u(S) = \sum_{p_i \in S, r_i \leq 0} (-r_i) + \sum_{p_i \notin S, r_i \geq 0} r_i = \sum_{i \in [n]: r_i \geq 0} r_i - \left[\sum_{i \in S} r_i \right].$$

Consequently $u(S)$ is just a fixed value that doesn't depend on S, $\sum_{i \in [n]} r_i$, minus the reward for picking the projects in S, $\sum_{i \in S} r_i$. Thus the projects in a minimum s-t cut are projects whose selection maximizes reward while meeting the dependency constraints.

3.3 Bipartite Matching

Let $G = (V, E)$ be an undirected unweighted bipartite graph, i.e. $V = L \cup R$ for disjoint L and R and for every edge $\{i, j\} \in E$ we have $i \in L$ and $j \in R$ or $i \in R$ and $j \in L$. Now we call $M \subseteq E$ a *matching* if for every edge $e, e' \in M$ with $e \neq e'$ we have $e \cap e' = \emptyset$. In other words, a matching is any subset of edges where each vertex is incident to at most one edge. We wish to compute a maximum matching in a bipartite graph. Here we may think that we are assigning students to schools and the edges denote pairs that both parties approve of and we wish to match as many as possible. We call a matching *perfect* if every vertex is incident to exactly one edge.

We can compute a maximum matching easily using maximum flow. We simply add a vertex s and a vertex t, and a (s, l) edge for every $l \in L$ and a (r, t) edge for every $r \in R$ and then make every undirected edge be orient from L to R. Now a s to t path in this graph has length exactly three, containing just one edge in the middle and we can easily see that there is a bijection between matchings in G and disjoint paths from s to t in the directed graph we just made by associating them through the edge between L and R. Consequently, by computing the maximum number of disjoint paths between s and t we can compute a maximum matching in G.

In the next lecture we will look at matching a bit more. One question we will ask is how to solve this same problem without the assumption that G is bipartite. Unfortunately, there is not a direct way to do this with maximum flow (or any concisely written linear program of standard form) and thus we will instead use algebraic techniques based on computing the determinant of linear algebraic objects associated with the graph.