Massachusetts Institute of Technology
Debayan Gupta, Aleksander Madry, and Bruce Tidor

Recitation 5: Max Flow and Linear Programming

1 Max Flow Review

To recap from last week's recitation, we have the following definitions for the maximum flow problem.

Definition 1 *A **flow network** is a directed graph $G = (V, E)$ where each edge has a non-negative capacity and the following property holds: for $u, v \in V$, if $(u, v) \in E$ then $(v, u) \notin E$.*

We are interested in the flow from a source $s \in V$ to a target $t \in V$:

Definition 2 *A **flow** is a real-valued function that maps an edge $(u, v) \in E$ to a non-negative real number $f(u, v)$ such that: (a) $f(u, v)$ is bounded by the capacity of (u, v) (this is known as the* capacity constraint*) and (b) the net flow is conserved (i.e.* flow conservation*) - i.e., for all $u \in V - \{s, t\}$,*

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

*We define the **value** of a flow f, denoted $|f|$, as the difference in the flow out of the source and the flow into the source:*

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

We can then formulate the ***maximum-flow problem,*** *which is given a flow network G, with* source *s and* sink *t, find a flow that maximizes value.*

The intuition behind solving most maximum flow problems is the idea that as long as there is a path from s to t in the network where none of the edges are filled to capacity, we can push additional flow along this path. We can thus define an ***augmenting path*** as a path from *s-t* in which all the edges have positive residual capacity. We can augment flow along an augmenting path by increasing the flow on each edge on the path by the same amount, while still preserving the conservation constraint. However, given a graph G, augmenting flow along all possible augmenting paths doesn't always give the correct answer (Exercise: why?).

Given a flow network and a flow, we formalize the notion of the remaining capacity in the network after we deduct the flow as follows:

Definition 3 *Given a graph $G = (V, E)$ and a flow f, the **residual network** is a graph G_f with vertices V and edges $E_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}$.*

FORD-FULKERSON($G = (V, E, c)$)

1: Initialize flow f to 0
2: **while** there is an augmenting path, p, in G_f **do**
3: Augment flow f along p
4: **end while**
5: **return** f

Assuming that all the capacities are integral, Ford-Fulkerson runs in $O(|f|E)$ time. If the capacities are real, termination is not guaranteed (Exercise: Why?). This is a pseudo-polynomial time algorithm, since it runs proportional to *value* of the input rather than the *size* of the input.

To convert this to a truly polynomial-time algorithm, we can choose a particular way to find augmenting paths. One possibility is to find augmenting paths by using BFS - this is the idea behind Edmonds-Karp, which is covered in the Appendix.

2 Maximum Bottleneck Capacity Augmenting Paths

Another intuitive way is to choose the augmenting path with the maximum bottleneck capacity, each time we search for augmenting paths. How can we find this path?

Given a particular guess, u, for the maximum bottleneck capacity, we can check for a path from s to t in G_f where all the edges with capacity less than u are removed. If there is no such path, then we know u is too large. We want to find the largest possible value of u, where such a path from s to t exists. For a particular value of u, this takes $O(m)$ time.

How can we find such guesses? We know that the maximum bottleneck capacity must be the residual capacity of one of the edges. If there are m such edges, we can sort all the edges in $O(mlogm) = O(mlogn)$ time. We can do binary search on the sorted array in $O(logn)$ time to find values of u to test. Thus, in $O(mlogn)$ time, we can find the path with the maximum bottleneck capacity.

However, this is the run time for a single augmentation. To calculate the run time, it remains to bound the number of required augmentations.

By using the flow decomposition lemma, we know that any flow can be decomposed into at most m paths. We can use an averaging argument to claim that at least one path in the residual graph carries at least $\frac{1}{m}$-part of the remaining flow. Since we are choosing the path which augments the flow the most, we know that the remaining flow must go down by a factor of at least $(1 - \frac{1}{m})$.

After $m\ ln|f| + 1$ augmentations, the remaining flow would become less than 1. If our capacities are integral, this means our solution is optimal.

Thus, this algorithm runs in $O(m^2\ log(n)\ log(|f|))$

3 Problem-Solving: Bipartite Matching

Problem: You are in charge of a distributed computing system. Every day, you get a list of n jobs, each of which has a "power requirement". You have access to n processors. That is job i, with a power requirement r_i can only be done on a processor j that has processing power p_j, if $p_j > r_i$. Your goal is to maximize the number of jobs that can be done concurrently - i.e. each processor can only handle one job.

Solution: We can formulate this problem as a bipartite graph and the goal becomes to find the maximum bipartite matching.

To maximize the number of jobs that can be done concurrently, we are trying to maximize the matching on the graph G. The hardest part of the problem is finding how to represent the problem as a graph, on which we can run our maximum flow algorithm.

We create our set of vertices V as follows:

- We add a source s and a sink t
- We add one vertex in the left side L for every job
- We add one vertex in the left side R for every processor

Then, we can create a set of edges E as follows:

- We create an edge (s, u) for every vertex $u \in L$
- We create an edge (v, t) for every vertex $v \in R$
- We create an edge (u, v) between $u \in L$ and $v \in R$ if $r_u \leq p_v$

All the edges have unit capacities and maximizing flow is equivalent to maximizing the number of jobs we can accomplish concurrently.

If we can push one unit of flow through a path p, it must pass through some edge (u, v) where $u \in L$ and $v \in R$. By construction, this edge would only exist if we can assign job u to processor v. Once we have run our max flow algorithm, we can find all such edges (u, v) to find the exact maximal matching between processors and jobs.

4 Linear Programming Review

Linear programming is a powerful tool that will allow us to optimize a linear objective function subject to a number of linear constraints. A linear program (LP) with n variables (x_j) and m constraints (not counting nonnegativity constraints) can be written in the form:

$$
\begin{aligned}
\max \quad & \sum_{j=1}^{n} c_j x_j && && \text{(objective function)} \\
\text{subject to} \quad & && && \\
& \sum_{j=1}^{n} A_{ij} x_j && \leq b_i \quad \text{for } i = 1, 2, \ldots, m && \text{(constraints)} \\
& x_j && \geq 0 \quad \text{for } j = 1, 2, \ldots, n && \text{(nonnegativity constraints)}
\end{aligned}
$$

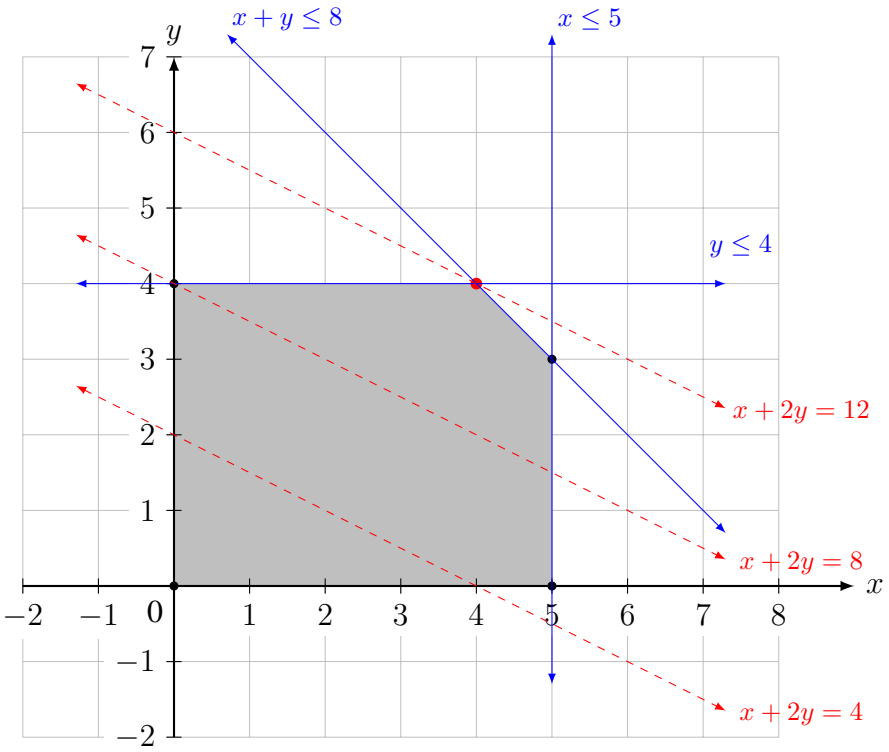
Linear programs written in the above form are said to be in *standard form.* Any solution that optimizes (maximizes, for a standard form LP) the objective value is an *optimal solution*.

All variable settings that satisfy all the constraints are *feasible solutions*. If a solution violates any of the constraints, then the solution is *infeasible*. If there are no feasible solutions, the linear program is *infeasible*. If the linear program does not have a finite optimal objective value, then it is said to be *unbounded*.

For example, consider the following standard form linear program:

$$
\begin{aligned}
\max \quad & x + 2y && \\
\text{subject to} \quad & y && \leq 4 \\
& x && \leq 5 \\
& x + y && \leq 8 \\
& x, y && \geq 0
\end{aligned}
$$

This linear program is small enough that we can solve it by hand, but the solution to a linear program also has a geometric interpretation.



In this two dimensional example, the feasible region is a convex polygon (gray area), and the level sets of the objective function are parallel lines (red dashed lines). The level sets here are all the lines $x + 2y = k$ (i. e. for each k, a line going through all the solutions that have objective value k).

With this geometric interpretation, the optimal solution can be found by "sliding" the line representing the objective function as far as possible while it still intersects the feasible region.

Therefore, the optimal objective value is achieved at either a vertex or an edge of the polygon. In particular, this means that the optimal value of a linear program is always achieved at some edge of the polyhedron it defines.

In the example, the line $x + 2y = 12$ contains all the solutions which have an objective value of 12. You cannot slide this line further up without having it leave the feasible region, so we cannot do better than 12 for our objective value.

To solve this linear program by hand, we try all the vertices of the feasible region: $(0, 0)$, $(5, 0)$, $(0, 4)$, $(4, 4)$, and $(5, 3)$. Plugging each into the objective function will give us values of 0, 5, 8, 12, and 11 in that order. Because one of the vertices must contain the optimal value, we know that the optimal value is 12 at $(4, 4)$.

5 Duality

One of the most important properties of linear programs is that of duality. Take some LP written in standard form:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \\ & \sum_{j=1}^{n} A_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \\ & x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n \end{aligned}$$

The linear program we are taking the dual of is referred to as the *primal* LP. The *dual* of the above maximization LP is a minimization LP with m variables and n constraints (not counting nonnegativity):

$$\begin{aligned} \min \quad & \sum_{i=1}^{m} b_i y_i \\ \text{subject to} \quad & \\ & \sum_{i=1}^{m} A_{ij} y_i \geq c_j \quad \text{for } j = 1, 2, \ldots, n \\ & y_i \geq 0 \quad \text{for } i = 1, 2, \ldots, m \end{aligned}$$

A summary of how to get the minimization dual from a maximization primal in standard form:

- Every variable in the primal corresponds to a constraint in the dual. A nonnegative variable yields a $\geq$ inequality in the dual.
- Every constraint in the primal corresponds to a variable in the dual. A constraint of the form $\leq b_i$ yields a variable that must be nonnegative (≥ 0).
- The primal's constraint bounds (b_i values) become the coefficients of the new objective function in the dual.
- The primal's coefficients in the objective function (c_j values) become the constraint bounds in the dual.
- A maximization primal becomes a minimization dual.

Strong duality tells us that the optimal objective value of the primal linear program equals the optimal objective value of the dual linear program. *Weak duality* says that the optimal objective value of the primal maximization linear program is upper bounded by the optimal objective value of the dual minimization linear program.

As an example, the dual of the linear program in the previous section is:

$$\begin{array}{lrl} \min & 4a + 5b + 8c & \\ \text{subject to} & b + c & \geq 1 \\ & a + c & \geq 2 \\ & a, b, c & \geq 0 \end{array}$$

Strong duality tells us that this linear program has the same optimal value as the primal (12). If you were to calculate it out, you'd find that the optimal occurs at $(a, b, c) = (1, 0, 1)$, giving us the objective value of 12 we were expecting.

In addition, using the coefficients $(a, b, c) = (1, 0, 1)$ that yield the dual optimal solution give us a certificate of optimality for our primal solution. If we multiply the inequalities from our primal by 1, 0, and 1, and add them together, we get:

$$\begin{array}{r} (1)(x + y) \leq (1)(8) \text{ and} \\ (0)(x) \leq (0)(5) \text{ and} \\ (1)(y) \leq (1)(4) \\ \Rightarrow x + 2y <= 12 \end{array}$$

This tells us that the maximum possible objective value we can get in the primal is 12. Because we found a way to get 12 in the primal LP, we can safely say there is no better objective value.

6 Application: LP of Max Flow and Min $s - t$ Cut

Model 1

The input will be a graph $G = (V, E)$ with source node s, sink node t, and c_{uv} as the capacity of the edge (u, v). For each edge $(u, v) \in E$, let's define a variable f_{uv} representing the flow along edge (u, v). For simplicity, we'll assume that $c_{uv} = 0$ if (u, v) is not in E and that E has no antiparallel edges (if an edge exists, its reverse will not).

The max flow linear program is:

$$
\begin{aligned}
\max \quad & \sum_{v:(s,v)\in E} f_{sv} - \sum_{v:(v,s)\in E} f_{vs} \\
\text{subject to} \quad & \\
& \sum_{u:(u,v)\in E} f_{uv} = \sum_{w:(v,w)\in E} f_{vw} \quad \forall v \in V - \{s,t\} \\
& f_{uv} \leq c_{uv} \quad \forall (u,v) \in E \\
& f_{uv} \geq 0 \quad \forall (u,v) \in E
\end{aligned}
$$

There is a constraint for each vertex (except the source and sink) based on the conservation of flow: the total flow going into a node equals the flow leaving the node. In addition, you'll need constraints that bound each edge's flow based on its capacity. The objective function is to maximize the net amount of flow leaving the source.

Model 2

Now, we'll look into an alternate way to formulate max flow as a linear program. Let P be the set of all paths from s to t in F, and let c_e be the capacity of edge e. For each path p in P, we'll define a variable f_p that represents the flow along path p. We want to maximize the total flow going from s to t (the sum of all the f_p values) but we are constrained by the fact that each edge has a capacity. Expressing this in a linear program yields:

$$
\begin{aligned}
\max \quad & \sum_{p\in P} f_p \\
\text{subject to} \quad & \\
& \sum_{p \text{ containing } e} f_p \leq c_e \quad \forall e \in E \\
& f_p \geq 0 \quad \forall p \in P
\end{aligned}
$$

Note that we can show that any feasible solution to the LP in Model 1 corresponds to a feasible solution to the LP in Model 2 with the same cost, by using the flow decomposition lemma, as covered in lecture.

Minimum $s-t$ Cut Model

This LP has an exponential number of variables, but taking the dual is now simpler. The objective function coefficients are all 1, and the constraint bounds are the capacities. For each variable in the primal (f_p), we should have a corresponding constraint in the dual. For each constraint in the primal, we should have a corresponding variable in the dual. Because there was a primal constraint for each edge e, we will call the corresponding variable in the dual x_e. The matrix A (coefficients of terms in the constraints) has a row for each edge and a 1 entry for each path that uses that edge. In our dual, we will use the columns of A instead of the rows. A has a column for each path and a 1 entry for each edge along that path. This gives us the following dual linear program:

$$
\begin{aligned}
\min \quad & \sum_{e \in E} c_e x_e \\
\text{subject to} \quad & \\
& \sum_{e \in p} x_e \geq 1 \quad \forall p \in P \\
& x_e \geq 0 \quad \forall e \in E
\end{aligned}
$$

We can interpret this linear program as a different graph problem that we have encountered before. That is, x_e is non-zero if edge e should be cut from the graph. The objective function minimizes the total weight of the edges that are cut, and the constraints require that at least one edge along each path has been cut. If one edge along each path has been cut, then we have a proper $s - t$ cut of G (one that blocks s from reaching t). Thus, we have shown that taking the dual of the max flow problem gives us the minimum $s - t$ cut problem formulation.

Since the objective function of the primal is the maximum flow we can push through the network and the objective of the dual is the minimum total capacity in our cut, the Max Flow Min Cut theorem simply becomes a special case of the strong duality theorem.

7 Appendix: Edmonds-Karp

EDMONDS-KARP($G = (V, E, c), s, t$)

```
1:  for each edge (u, v) ∈ E do
2:      (u, v).f = 0
3:  end for
4:  while there exists a pathfrom s to t in the residual network do
5:      Find such a path p using BFS
6:      Let c_p be the minimum capacity of any edge in p
7:      for each edge (u, v) in p do
8:          if (u, v) ∈ E then
9:              (u, v).f = (u, v).f + c_p
10:         else
11:             (v, u).f = (v, u).f − c_p
12:         end if
13:     end for
14: end while
```

The Edmonds-Karp algorithm is very similar to Ford-Fulkerson except that it specifies how augmenting paths are found - in particular, Edmonds-Karp uses BFS to find thse paths.. We claim that this change causes the number of augmentations to be limited to $O(|V||E|)$, for a total runtime of $O(|V||E|^2)$.

Invariant: non-decreasing source distances

For a vertex v, define the shortest distance from the source (in number of edges) in the residual graph for flow f as $\delta_f(s, v)$. We will prove the invariant that this value is non-decreasing for all vertices.

Suppose to the contrary that while augmenting the flow from f to f', some vertex's shortest-path distance decreases. Consider in particular the vertex v with the minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation. Let p be a shortest path from s to v in $G_{f'}$, and let u be the immediate predecessor of v on this path, so that $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$.

Because of how we picked v, we know that $\delta(s, u)$ did not decrease. So it must be the case that $(u, v) \notin E_f$, because otherwise we could have reached v in G_f via as short a path through u. That is, if $(u, v) \in E_f$, then $\delta_f(s, v) \leq \delta_f(s, u) + 1 \leq \delta_{f'}(s, u) + 1 = \delta_{f'}(s, v)$, which contradicts our assumption that $\delta(s, v)$ decreased.

The only way we could have (u, v) in $E_{f'}$ but not in E_f is if the augmentation pushed flow along (v, u). Now, because Edmonds-Karp always pushes flow along a shortest path, some shortest path from s to u in G_f had (v, u) as its last edge. So $\delta_f(s, v) = \delta_f(s, u) - 1 \leq \delta_{f'}(s, u) - 1 = \delta_{f'}(s, v) - 2$, which contradicts the assumption that the distance to v decreased.

Bounding the number of augmentations

We will use this fact to bound the number augmentations that Edmonds-Karp does by charging each augmentation to a minimum residual capacity edge along the augmenting path.

Consider an augmentation from f that saturates some edge (u, v). Note for later that $\delta_f(s, v) = \delta_f(s, u)+1$. The augmentation saturates (u, v) and causes it to be removed from the residual graph.

Now, for (u, v) to be charged for another augmentation, it would need to be added back to the residual graph, and this can only happen if we push flow along the back edge (v, u). Let f' be the new flow when this happens. Again, because Edmonds-Karp picks shortest paths, we have that $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$. By the invariant proved above, this is at least $\delta_f(s, v) + 1 = \delta_f(s, u) + 2$. That is, $\delta(s, v)$ increases by at least 2 before the edge (u, v) is charged again.

Because the δ's are bounded by $|V|$, we conclude that each edge is charged $O(|V|)$ times, for a total number of augmentations of $O(|V||E|)$ and a total runtime of $O(|V||E|^2)$, as claimed.