Preface

This is a set of lecture notes for *CS369A-B: Topics in Analysis of Algorithms* taught by Andrew Goldberg and Serge Plotkin in Spring 1992. The course was devoted to Combinatorial Optimization and aimed at preparing students to do research in the area. We pursued two goals while teaching the course. The first goal was to teach advanced background material and the second goal was to present some of the recent research results.

The notes were scribed by students in the class and edited by the lecturers. These notes are not ment to be in the polished form, and probably contain some errors and omissions. In particular, many references to the literature are missing. The reader is directed to the relevant books and papers for a formal treatment of the material.

These notes are intended as an aid in teaching and for studying advanced work in the area. Much of the material covered in these notes is very recent and is available in research papers only.

We would like to thank the students who took the course and scribed. We also would like to thank Orli Waarts who, as a Teaching Assistant for course, provided valuable help, especially with the notes.

Contents

Lecture #1: April 1, 1992
Scribe: David Karger

1 Preliminaries

1.1 The Fundamental Theorem of Linear Inequalities

This lecture draws from the book *Theory of Integer and Linear Programming* by Schrijver [14], Sections 7.1–7.3.

1.1.1 Definitions

We begin with some conventions and definitions. Throughout these notes, we will assume we are working in the vector space $\mathbb{R}^n$. Vectors in this space will be denoted by lowercase Roman letters a,b,c, etc. Scalars will be denoted by lowercase greek letters $\alpha, \beta, \gamma, \ldots$, and matrices by capital Roman letters A,B,C (capitals will also be used to denote sets). Juxtaposition will denote scalar multiplication, dot product, or matrix multiplication depending on the types of the operands involved. For sets X and Y, let $X + Y$ denote $\{x + y \mid x \in X, y \in Y\}$ and similarly let rX denote $\{rx \mid x \in X\}$. We now define some geometric objects which will appear frequently:

Definition 1 A set $C \subset \mathbb{R}^n$ is a *convex cone,* or just a cone, if $\lambda C + \mu C \subseteq C$ for any $\lambda, \mu \geq 0$. The cone *generated* by $\{x_1, \ldots, x_m\}$ is the smallest cone containing the x_i, namely $\{\sum \lambda_i x_i \mid \lambda_i \geq 0\}$, is called a *finitely generated cone*, and is denoted by $\text{cone}(\{x_i\})$.

Definition 2 A *polyhedron* is the intersection of finitely many affine half-spaces. It thus has the form $\{x \mid Ax \leq b\}$. A *polytope* is a bounded polyhedron. An alternative, equivalent definition (equivalence will follow from a lemma to be proved later) is that a polytope is the convex hull of a finite set of points.

1.1.2 The Fundamental Theorem

We first state the fundamental theorem, and then discuss its meaning.

Theorem 3 Let $a_1, \ldots, a_n \in \mathbb{R}^n$. Then $\forall b \in \mathbb{R}^n$, either

1. $b = \sum \lambda_i a_i$ for some $\lambda_1, \ldots, \lambda_n \geq 0$, or
2. $\exists c$ such that the hyperplane $\{x \mid cx = 0\}$ contains at least $t-1$ linearly independent a_i with
   - $cb < 0$,
   - $ca_i \geq 0$ for all i,
   - $t = \operatorname{rank}\{a_1, \ldots, a_n, b\}$

What this theorem is saying is that if we consider the cone generated by the a_i, then either b is in the cone, or there exists a plane which separates b from the cone. Note that the two conditions in the theorem are exclusive, since when situation 1 holds, having $cb < 0$ necessarily implies that for some i, $ca_i < 0$, violating condition 2.

Proof: The definition of t is actually a compact way to describe two cases. If b is linearly independent of the a_i, then a hyperplane satisfying the theorem is just any hyperplane which contains all the a_i but not b (clearly such a hyperplane exists). On the other hand, if b is a linear combination of the a_i, then we can restrict our attention to the t-dimensional subspace spanned by the a_i, and find a hyperplane in this space which is spanned by $t-1$ of the a_i. So to simplify matters, lets also assume the the a_i span $\mathbb{R}^n$. From the $\{a_i\}$, choose some arbitrary basis $D = \{a_{i_j} \mid j = 1, \ldots, n\}$. We will iteratively modify this basis until it provides a proof of the theorem. To do so, write $b = \sum \lambda_{i_j} a_{i_j}$. If all the $\lambda_{i_j} \geq 0$, then case 1 holds and we are done. If not, choose the smallest i_j that $\lambda_{i_j} < 0$, and throw it out. The remaining vectors define a hyperplane. Let c be the normal to this hyperplane, with the direction of c be chosen so that $cb < 0$. Now check whether for each i, $ca_i \geq 0$. If so, case 2 holds and the theorem is proved. If not, take the smallest s such that $ca_s < 0$, and add a_s to the basis. This ends one iteration.

To prove the theorem, it remains to show that this iterative procedure eventually terminates in one of the ways mentioned above. Since the number of possible bases is finite (though exponential in n), all that is necessary to show that no basis can arise more than once. Let D_i denote the basis after iteration i, and suppose $D_k = D_l$ for some $k < l$. Let r be the highest index for which a_r was removed during some iteration between k and $l-1$, say iteration p. Since $D_k = D_l$, necessarily a_r was put back at some iteration preceding l, say at iteration q. Let us concentrate on D_p and D_q. Let $D_p = \{a_{i_j}\}$, $b = \sum \lambda_{i_j} a_{i_j}$, and let c_q be the normal to the hyperplane found in iteration q. Consider

$$c_q b = \sum \lambda_{i_j} a_{i_j} c_q.$$

This quantity is less than 0, because c_q is defined so that $c_q b < 0$. However, we will show a contradiction by proving that the right hand side of this equation is positive. We do so by showing that each term in the right hand sum is nonnegative, and that one is positive.

First consider terms with an index $i_j > r$. If $a_{i_j} \in D_p$, then since no vector of index exceeding r was removed between iterations k and l, it must be the case that $a_{i_j} \in D_q$ as well. But now we need merely recall that c_q was defined so that $ca_{i_j} = 0$.

Now consider the term with index r, which must be in the sum because of the way D_p is defined. Since a_r was brought back in iteration q, necessarily $c_q a_r < 0$. Also, in iteration p

vector a_r was discarded, so it must be that $\lambda_r < 0$. Thus the term with index r in the above equation is positive.

Finally, consider $i_j < r$. $\lambda_{i_j} \geq 0$ because otherwise this index would have been selected and discarded in iteration p. On the other hand, $c_q a_{i_j} \geq 0$ because otherwise index i_j would have been used in iteration q. Thus $\lambda_{i_j} a_{i_j} c_q \geq 0$. The contradiction has thus been shown. ▮

Those who have seen the simplex algorithm may note the similarity of what we are doing here to the simplex method. In particular, consider applying the simplex method to find a feasible solution to

$$\left[\left(\; a_1 \; \right) \left(\; a_2 \; \right) \cdots \right] \Lambda = b, \text{ subject to } \Lambda \geq 0$$

where $\Lambda = (\lambda_1, \ldots, \lambda_m)$. We either find a solution (a convex combination of the a_i) or find a proof that no such solution exists.

1.1.3 Applications

We now show some applications of the Fundamental Theorem. For the first, consider the following alternative definition of cones: Let a *polyhedral cone* be a set of the form $\{x \mid Ax \leq 0\}$. It turns out that we can show the following, using the fundamental theorem just proved:

Theorem 4 A cone is polyhedral of and only if it is finitely generated.

Proof: We first prove that any finitely generated cone is polyhedral. Consider a cone generated by $x_1, \ldots, x_m \in \mathbb{R}^n$. Consider the set $\mathcal{H}$ of half-spaces through the origin which correspond to hyperplanes defined by some $n-1$ linearly independent x_i and contain every x_i. Note there are only finitely many such half-spaces. Each such hyperspace is convex which means it contains all convex combinations of the x_i and therefore contains C. Thus $C \subset \bigcap \mathcal{H}$. On the other hand, consider some $b \notin C$. By the fundamental theorem b is on the wrong side of at least one of the half-planes in $\mathcal{H}$, so $b \notin \bigcap \mathcal{H}$. It therefore follows that $C = \cap \mathcal{H}$ and is therefore a polyhedral cone.

Now consider the other direction. Suppose $C = \{x \mid a_1 x \leq 0, a_2 x \leq 0, \ldots\}$. Consider the cone generated by the a_i. By the previous paragraph, it is a polyhedral cone, *i.e.* it has the form $C' = \{x \mid b_1 x \leq 0, b_2 x \leq 0, \ldots\}$. Let C'' be the cone generated by the b_j. We claim $C = C''$. To prove $C'' \subset C$, it suffices to show that each $b_j \in C$. But since $a_i \in C'$ by definition, it is necessarily the case that $a_i b_j \leq 0$; therefore $b_j \in C$ follows. To prove $C \subset C''$, suppose $x \in C$ but $x \notin C''$. Then the fundamental theorem tells us there is a hyperplane separating x from the b_j, *i.e.* a vector c such that $cx > 0$ but $cb_j \leq 0$ for each j. From this deduce that $c \in C'$. Since C' is generated by the a_i, we have $c = \sum \lambda_i a_i$ where the λ_i are positive. Then $cx = \sum \lambda_i a_i x$. Since $x \in C$, we have $a_i x \leq 0$ for each i. Thus $cx \leq 0$, contradicting the fact that $cx > 0$. ▮

Note that a polyhedral cone is a more restricted entity than a general cone, because a general cone can have sides which are not planes (*e.g.* the cone in $\mathbb{R}^3$ defined by $x^2 + y^2 < z$).

Lemma 5 Any polyhedron $P = \{x \mid Ax \leq b\}$ is of the form $Q + C$ for some polytope Q and cone C.

Proof: Consider the vectors $\{\binom{x}{\lambda} \mid x \in \mathbb{R}^n, \lambda \in \mathbb{R}, \lambda \geq 0, Ax \leq \lambda b\}$. This is a polyhedral cone in $\mathbb{R}^{n+1}$. It is thus generated by some set $A' = \{\binom{x_1}{\lambda_1}, \binom{x_2}{\lambda_2}, \ldots\}$, where we can normalize these vectors so that each λ_i is either 1 or 0. Let $X_0 = \{x_i \mid \lambda_i = 0\}$ and $X_1 = \{x_i \mid \lambda_i = 1\}$. We claim that C can be taken as the cone generated by X_0 and Q the polytope generated by X_1. To prove this, consider $p \in P$. We have

$$\begin{aligned} p \in P \quad &\Longleftrightarrow \quad Ap \leq b \\ &\Longleftrightarrow \quad \begin{pmatrix} p \\ 1 \end{pmatrix} = \sum_{x \in X_1} \lambda_x \begin{pmatrix} x \\ 1 \end{pmatrix} + \sum_{x \in X_0} \mu_x \begin{pmatrix} x \\ 0 \end{pmatrix} \end{aligned}$$

(At this point, moving in either direction in the implication, we can deduce that $\sum \lambda_x = 1$ and $\mu_x \geq 0$.)

$$\begin{aligned} &\Longleftrightarrow \quad p = q + c \quad \text{where} \quad q = \sum_{x \in X_1} \lambda_x x, c = \sum_{x \in X_0} \mu_x x \\ &\Longleftrightarrow \quad p = q + c \text{ with } q \in Q \text{ and } c \in C \end{aligned}$$

∎

Lecture #2: April 6, 1992
Scribe: Michael Goldwasser

1.2 Farkas' Lemma

In this lecture we will consider the Farkas' lemma which characterizes a necessary and sufficient condition for the existence of a feasible non-negative solution to a set of linear inequalities with the form $Ax = b$. From this, we will construct an alternate form of Farkas' lemma which gives a necessary and sufficient condition for the existence of any feasible solution to a set of linear inequalities with the form $Ax \leq b$.

Lemma 6 (Farkas' Lemma) Given a matrix A and vector b, there exists some $x \geq 0$ satisfying $Ax = b$ if and only if for any vector y, $yA \geq 0$ implies $yb \geq 0$.

It follows that given any linear programming problem, there is an easy proof of the feasibility (exhibit a feasible point) *or* non-feasibility (exhibit a witness y with $yA \geq 0$ but $yb < 0$) of the solution.

Proof: If there exists a solution x such that $Ax = b$, then consider y with $yA \geq 0$. Since $yb = y(Ax) = (yA)x$, and since by hypothesis $yA \geq 0$ and $x \geq 0$, it follows that $yb \geq 0$. On the other hand, suppose there is no solution. Consider A as a row of column vectors $a_1, \ldots, a_m$ and consider the cone generated by the a_i. Since there is no solution, it follows that b is not in the cone generated by the a_i (since this is what $Ax = b$ with $x \geq 0$ means). Using the fundamental theorem, we know there exists some y such that $ya_i \geq 0$ for every i (so that $yA \geq 0$), but such that $yb < 0$. ∎

Lemma 7 (Farkas' Lemma, alternate form) Given a matrix A and a vector b, there exists some vector x satisfying $Ax \leq b$ if and only if for any vector $y \geq 0$, $yA = 0$ implies $yb \geq 0$.

Proof: Examine the condition $\exists x : Ax \leq b$. Although x is not necessarily non-negative, we can rewrite any x such that $x = x_1 - x_2$ with $x_1, x_2 \geq 0$. Therefore there exists a solution to $[A]\,[x] \leq b$ if and only if there exists a solution to

$$\begin{bmatrix} A & -A \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq b, \text{ where } x_1, x_2 \geq 0$$

Now, since the right side of the equation dominates, we can create an equality by adding a non-negative vector x_3 to the left side. These variables are called *slack variables.*

$$\underbrace{\begin{bmatrix} A & -A & I \end{bmatrix}}_{A'} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = b, \text{ where } x_1, x_2, x_3 \geq 0$$

Now the existence of x such that $Ax \leq b$ is equivalent to the existence of non-negative $x' \geq 0$ such that $A'x' = b$. By Lemma 6,

$$\begin{aligned} (\exists x' \geq 0 : A'x' = b) &\Leftrightarrow (\forall y : yA' \geq 0 \Rightarrow yb \geq 0) \\ &\Leftrightarrow (\forall y : yA \geq 0, -yA \geq 0, y \geq 0 \Rightarrow yb \geq 0) \\ &\Leftrightarrow (\forall y : yA = 0, y \geq 0 \Rightarrow yb \geq 0) \end{aligned}$$

∎

Lemma 8 (Farkas' Lemma, yet another form) Given a matrix A and a vector b, the system $\{Ax \leq b \text{ and } ax < b'\}$ is infeasible if and only if one of the following conditions is true:

- $\exists y \geq 0$ such that $yA = 0$, $yb < 0$.
- $\exists y \geq 0$ such that $yA + a = 0$, $yb + b' \leq 0$.

Proof: Since the second constraint is a strict inequality, we can not directly apply any of the previous forms of Farkas' Lemma to derive the necessary and sufficient conditions for infeasibility. Observe that the conditions stated in the lemma are sufficient to prove that the system is infeasible. All that is left is to show that if the system $\{Ax \leq b \text{ and } ax < b'\}$ is infeasible, then there exists such y.

Observe that the original system is infeasible if and only if the following system is infeasible for all $\epsilon > 0$.

$$\begin{bmatrix} Ax \leq b \\ ax \leq (b' - \epsilon) \end{bmatrix}$$

Now we can apply Lemma 7, which states that this system is infeasible if and only if $\exists (y, \lambda) \geq 0$ such that $yA + \lambda a = 0$ and $yb + \lambda(b' - \epsilon) < 0$.

If $\lambda = 0$ for this example, then we have that $y \geq 0$, $yA = 0$ and $yb < 0$, and so this y satisfies the first condition in the lemma.

Assume that we can not find $y \geq 0$ such that $yA = 0$ and $yb < 0$, *i.e.* $Ax \leq b$ is feasible. In this case we need to prove that $\exists (y, \lambda), y \geq 0, \lambda > 0$ such that $yA + \lambda a = 0$ and $yb + \lambda b' \leq 0$. Scaling by λ, we see that we need to prove that $\exists y \geq 0$ such that $yA + a = 0$ and $yb + b' \leq 0$. We know that for any $\epsilon > 0$, there exists $y \geq 0$, such that $yA + a = 0$ and $yb + b' - \epsilon < 0$. Consider $\inf\{yb + b' | yA + a = 0\}$. Since the set that we minimize over is non-empty, either the bound is $-\infty$ or it is less than 0. In both cases, we are done. ∎

1.3 LP Duality

1.3.1 Standard form of Duality Theorem

To solve a linear program $max\{\, cx : Ax \leq b \,\}$, we create a second linear program inherently related to the first. We will call the original program the *primal* program, and we will create what is called the *dual* program. For the above linear program, the dual is $min\{\, yb : yA = c, y \geq 0 \,\}$. The importance of the dual is brought out by the following theorem

Theorem 9 (Duality Theorem) Given matrix A and vectors b and c, $\max\{cx \mid Ax \leq b\} = \min\{yb \mid y \geq 0, yA = c\}$ so long as both limits exist.

Proof: Proving the duality theorem is equivalent to proving the following two conditions hold:

(1) For all feasible x and y, $cx \leq yb$.

(2) There exists some feasible $\hat{x}$ and $\check{y}$, such that $c\hat{x} \geq \check{y}b$.

Condition (1) follows directly from the definition of the primal and dual constraints. Since $c = yA$ then $cx = yAx$ for any feasible x and y. Also, since $Ax \leq b$ and $y \geq 0$, then $yAx \leq yb$. Combining these two, $cx \leq yb$.

Proving condition (2) is the more difficult feat. We will use Farkas' lemma to do so. We want to show the existence of an x and y such that all of the following conditions hold: $Ax \leq b$, $y \geq 0$, $yA = c$, and $cx \geq yb$. These equations can be combined into the following linear system:

$$\underbrace{\begin{bmatrix} A & 0 \\ -c & b^t \\ 0 & A^t \\ 0 & -A^t \\ 0 & -I \end{bmatrix}}_{\Lambda} \begin{pmatrix} x \\ y^t \end{pmatrix} \leq \underbrace{\begin{pmatrix} b \\ 0 \\ c^t \\ -c^t \\ 0 \end{pmatrix}}_{\beta}$$

We must show that there exists a solution to this system. To apply Lemma 7 we must show that for all $\gamma \geq 0 : \gamma\Lambda = 0 \Rightarrow \gamma\beta \geq 0$. For $\gamma = (\begin{array}{ccccc} u & \lambda & v & w & q \end{array})$, the condition $\gamma\Lambda = 0$ is,

$$\begin{pmatrix} u & \lambda & v & w & q \end{pmatrix} \begin{bmatrix} A & 0 \\ -c & b^t \\ 0 & A^t \\ 0 & -A^t \\ 0 & -I \end{bmatrix} = 0$$

Using Farkas' lemma, to show the existence of a solution to this problem, we must show that the condition ($uA - \lambda c = 0$ and $\lambda b^t + (v - w)A^t \geq 0$) implies ($ub + (v - w)c^t \geq 0$). To do so, we consider two cases.

- Case (1) is when $\lambda > 0$. Using the two assumptions, we can write $ub = \lambda^{-1}(\lambda b^t)u^t \geq \lambda^{-1}(w - v)A^t u^t = \lambda^{-1}(w - v)\lambda c^t$. Thus $ub + (v - w)c^t \geq 0$.

- Case (2) is when $\lambda = 0$. Assume that $\gamma \geq 0, \gamma\Lambda = 0$ does not imply $\gamma\beta \geq 0$ for all γ. Then there exists such a γ so that $uA = 0$, $(v - w)A^t \geq 0$, and $ub + (v - w)c^t < 0$. However if we consider this same system without the constraint that $cx \geq yb$, together with this same γ without the λ component, we can apply Farkas' Lemma to show that even this reduced system is infeasible. However this system is just the combination of the primal and dual systems, and so this would imply that one of those systems was infeasible, which contradicts our assumptions.

For either case, we proved that for all $\gamma \geq 0 : \gamma\Lambda = 0 \Rightarrow \gamma\beta \geq 0$, and so by Farkas' lemma, we have shown the existence of such solutions $\hat{x}$ and $\check{y}$. ∎

1.3.2 Alternate Forms of LP Duality

There are similar duality theorems for various other forms of linear programs. These allow many forms of primal programs to be translated into related dual programs.

Theorem 10 (Alternate Duality Theorems) Given matrix A and vectors b and c, the following equalities hold, whenever the involved minimums and maximums exist.

- $\max\{cx \mid Ax \leq b\} = \min\{yb \mid y \geq 0, yA = c\}$
- $\max\{cx \mid x \geq 0, Ax \leq b\} = \min\{yb \mid y \geq 0, yA \geq c\}$
- $\max\{cx \mid x \geq 0, Ax = b\} = \min\{yb \mid yA \geq c\}$

Proof: The first form is exactly that which was proven in the original theorem. Given the first equation, to derive the second equality, we must introduce the constraint $x \geq 0$. Apply the first equation with a modified A':

$$\underbrace{\begin{bmatrix} A \\ -I \end{bmatrix}}_{A'} (x) \leq \underbrace{\begin{pmatrix} b \\ 0 \end{pmatrix}}_{b'}$$

Notice that $\{x \geq 0, Ax \leq b\} = \{A'x \leq b'\}$, and so if the first has a feasible solution so does the second. Now we can apply the first duality statement to get that $\max\{cx \mid A'x \leq b'\} = \min\{y'b' \mid y' \geq 0, y'A = c\}$ where $y' = \begin{pmatrix} y \\ y_0 \end{pmatrix}$. This program is equivalent to $\min\{yb \mid y \geq 0, yA \geq 0\}$, and so we have the second form of duality.

The third version is exactly symmetric to the first in terms of x and y. However, we could also derive it from the second by again considering a new matrix A' and applying the second equation,

$$\underbrace{\begin{bmatrix} A \\ -A \end{bmatrix}}_{A'} \begin{pmatrix} x \end{pmatrix} \leq \underbrace{\begin{pmatrix} b \\ -b \end{pmatrix}}_{b'}$$

Notice $\{x \geq 0, A'x \leq b'\} = \{x \geq 0, Ax = b\}$, and since the feasibility is equivalent for the two, we can apply the second version of duality to get, $\max\{cx \mid x \geq 0, Ax \leq b'\} = \min\{y'b' \mid y \geq 0, yA \geq c\}$ where $y' = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$. This program is equivalent to $\min\{yb \mid yA \geq c\}$ for $y = (y_1 - y_2)$, and so we have derived the third version of duality. ∎

We would like to show that if a program has a feasible solution which is bounded, then there exists a feasible solution which can be written as a non-negative linear combination of the inequalities bounding the problem.

Theorem 11 Let $Ax \leq b$ have at least one solution. Suppose $cx \leq \delta$ holds for all vectors x lying in $Ax \leq b$. Then there exists $\delta' \leq \delta$ such that $cx \leq \delta'$ is a non-negative combination of inequalities in $Ax \leq B$. That is there exists a vector y which contains the coefficients for this combination, such that $yA = c$, $yb = \delta'$, and $y \geq 0$.

Proof: From the duality theorem, if both limits are defined, we know that $\max\{cx \mid Ax \leq b\} = \min\{yb \mid y \geq 0, yA = c\}$, and this theorem is satisfied by any such optimal dual solution.

To derive a contradiction, assume that there is no feasible solution to the dual problem. Then by applying a form of Farkas' lemma, there must exist some vector z, such that both $Az \leq 0$, and $cz > 0$. However there exists a feasible solution to the primal, and so consider the effect of adding z to this solution. Since $Az \leq 0$, adding this vector will not violate any of the constraints, and so it will still be a feasible solution to the primal. However since $cz > 0$, then the cost value of this solution will be strictly greater after adding in z. And since we can continually add in more copies of z, then the primal problem cannot have any δ which bounds the primal solution. Therefore if the primal problem is non-empty and bounded, then the dual problem must have some feasible solution.

Since both problems have feasible solutions, then by the duality theorem, there is some optimal dual solution $\breve{y}$ such that $\breve{y}A = c$, $\breve{y} \geq 0$, and $\breve{y}b = \delta'$ where δ' is the maximal cost value of the primal. This $\breve{y}$ satisfies this theorem. ∎

Given a primal program and its dual, this reasoning and the duality theorem ensures us that there are only four possible scenarios.

1. The primal and dual programs are both non-empty and bounded, and thus the optimal values are equal.

2. The primal is non-empty but unbounded, and thus the dual is empty, in which case the optimal value for both problems is $+\infty$.

3. The dual problem is non-empty but unbounded, and thus the primal is empty, in which case the optimal value for both problems is $-\infty$.

4. Both the primal and dual problems are empty, and thus the optimal values for both are undefined.

1.3.3 Complementary Slackness

Earlier, we introduced the idea of a slackness variables to turn inequalities to equalities. For instance, if $Ax_0 \leq b$, we added a non-negative slackness variable x_1, so that $Ax_0 + x_1 = b$. The following theorem relates the value of the slackness variables $(b - Ax_0)$ of a solution to the corresponding dual variables.

Theorem 12 (Complimentary Slackness) Given the primal linear program $\max\{cx \mid Ax \leq b\}$ and the dual $\min\{yb \mid y \geq 0, yA = c\}$, and given feasible solution: $\hat{x}$ and $\breve{y}$, the following are equivalent:

(1) $\hat{x}$ and $\breve{y}$ are optimum.

(2) $c\hat{x} = \breve{y}b$.

(3) $\forall i : (\breve{y}_i > 0 \rightarrow a_i\hat{x} = b_i)$.

Proof: The equivalence of statements (1) and (2) is exactly the duality theorem. So assume statement (2), $c\hat{x} = \breve{y}b$. Since $c = \breve{y}A$, this equality can be written, $\breve{y}A\hat{x} = \breve{y}b$, and so $\breve{y}(A\hat{x} - b) = 0$. Since both $\breve{y}, (A\hat{x} - b) \geq 0$, then each component in the dot product is non-negative. For the sum to be equal to 0, each component must have one (or both) of $\breve{y}_i$ and $(a_i\hat{x} - b_i)$ equal to 0. Therefore, for any i, if $\breve{y} > 0$, then $(a_i\hat{x} = b_i)$. Therefore (2) implies (3).

To see that (3) implies (2), assume that $\forall i : (\breve{y}_i > 0 \rightarrow a_i\hat{x} = b_i)$. Clearly, if $\breve{y}_i = 0$, then $\breve{y}_i a_i\hat{x} = \breve{y}_i b_i$. Also if $\breve{y}_i > 0$, then by the assumption $a_i\hat{x} = b_i$, and thus $\breve{y}_i a_i\hat{x} = \breve{y}_i b_i$. So $\forall i : \breve{y}_i a_i\hat{x} = \breve{y}_i b_i$. Since this is true for all i, $\breve{y}A\hat{x} = \breve{y}b$, and since $c = \breve{y}A$, then $c\hat{x} = \breve{y}b$. Therefore (3) implies (2), and all the statements are equivalent. ∎

Carefully consider the complimentary slackness theorem. Given an optimal primal and dual solution, that theorem states that if the optimum value of the dual variable y_i is strictly greater than zero, then the corresponding constraint must be tightly satisfied by any optimum primal solution. However the converse is not necessarily true. That is, if a constraint is tight, the corresponding dual variable might still equal zero. The following generalized theorem shows that if the constraint is tight, even though the dual variable might still equal zero, there must exist *some* optimal dual solution whose corresponding variable will be positive.

Theorem 13 Given the primal linear program $\max\{cx \mid Ax \leq b\}$ and the dual $\min\{yb \mid y \geq 0, yA = c\}$, if both limits are defined, then exactly one (but not both) of the following statements is true:

(a) The primal has an optimal solution $\hat{x}$ with $a_i\hat{x} < b_i$.

(b) The dual has an optimal solution $\breve{y}$ with $\breve{y}_i > 0$.

Proof: We already know that (a) and (b) must exclude each other from Theorem 12. Since $\hat{x}$ and $\breve{y}$ are optimal, then condition (b), $\breve{y}_i > 0$ implies $a_i\hat{x} < b_i$. So assume that (a) is not satisfied, *i.e.* that is there is no optimum solution $\hat{x}$ with $a_i\hat{x} < b_i$. If we assume that the optimum value is δ, then the following system of equations has no solution:

$$\begin{bmatrix} Ax \leq b \\ -cx \leq -\delta \\ a_i x < b_i \end{bmatrix}$$

If we consider the dual vector $\begin{pmatrix} y & \lambda & \lambda' \end{pmatrix}$, then we can apply the form of Farkas' Lemma stated in Lemma 8.

One of the conditions of Lemma 8 must be true. Assume the first condition is true, that is $\exists y, \lambda \geq 0$ such that $yA - \lambda c = 0$, $yb - \lambda\delta \geq 0$. Considering only the first two constraints, the existence of such a vector (y, λ), when combined with Farkas' lemma, shows the infeasibility of $\{Ax \leq b, -cx \leq -\delta\}$. However since we assumed that there was a feasible solution obtaining this maximal value δ, this cannot be the case.

Therefore, the second condition from Lemma 8 must be true, that is $\exists y', \lambda \geq 0$ such that $y'A - \lambda c + a_i = 0$ and $y'b - \lambda\delta + b_i \leq 0$. From this we build vector y_0 by taking y' but increasing the i^{th} component by 1. Consider the two cases depending on whether or not λ is strictly greater than 0.

If $\lambda > 0$, then consider $\breve{y} = \lambda^{-1}y_0$. We know that $\breve{y}$ is feasible since $\breve{y}A = \lambda^{-1}y_0A = \lambda^{-1}(y'A + a_i) = \lambda^{-1}(\lambda c) = c$. Also notice that $\breve{y}b = \lambda^{-1}y_0b = \lambda^{-1}(y'b + b_i) \leq \lambda^{-1}(\lambda\delta) = \delta$. Since $\breve{y}b \leq \delta$, then this must be an optimal solution, and since the i^{th} component is strictly positive, this solution satisfies condition (b).

If $\lambda = 0$, then $y'A + a_i = 0$ and $y'b + b_i \leq 0$. Since the limit for the dual existed there exists some optimal y for the dual program. We claim that $\breve{y} = (y + y_0)$ also attains the minimum, and has a positive i^{th} component. Since y is feasible, then $\breve{y}A = yA + y_0A = yA + 0 = c$, and so $\breve{y}$ is also a feasible solution. Also, $\breve{y}b = yb + y_0b \leq yb$ and so $\breve{y}$ is an optimal dual solution. However by construction, $\breve{y}$ has an i^{th} component that is strictly positive. ∎

1.3.4 Intuition of Duality

To better understand the intuition in the previous theorem, let's view the constraint problem from a physicist's point of view. Each of the hyperplanes that are constraints, are represented as walls. Our solution is a ball floating in space. Assuming the program has some feasible solution, then the ball is always placed inside the set of walls. The objective function is a

Figure 1: The Physicist's View

directional force pushing the ball. The goal is to take all feasible positions, x, and to maximize the objective function over the set.

We know that in the physical model, the objective function will be maximized when it is against a face, or in a corner. At this point the object will stop moving, because the objective force will be exactly balanced by the combined forces of the walls that the object is up against.

In Figure 1, the a_i's represent the directions of the constraints, and the vector c represents the objective function. At the optimal point, the force c will exactly be balanced by the restraining forces of the walls. Since the walls only provide a force in the direction opposite of the a_i's, then $-c + \sum_i y_i(a_i) = 0$, where $y_i \geq 0$. In other words, c can be expressed as a non-negative combination of the rows of A. That is $c = y_1a_1 + \ldots + y_ma_m$ where $y_i \geq 0$. These coefficients correspond to the dual variable y. Furthermore, the intuition of slackness can be seen. The distance between a position x and a wall corresponds to the slackness at that constraint. This value is equal to $b - a_ix$. In the physical model, a specific wall can only provide a balancing force if the position x is up against the wall, that is if the system is tight at that constraint. Therefore, if $y_i > 0$, then that wall is providing some non-zero balancing force, and therefore the position x must be against that wall. In terms of the complimentary slackness conditions, $\forall i, (y_i > 0 \Rightarrow a_ix = b_i)$. However it is possible that the ball could be up against a wall that is not contributing any force. If the objective function runs parallel to the constraint, the ball could be resting in a corner against that wall, even though the wall is providing no support. In terms of the slackness conditions, it is possible to have both $(y_i = 0$ and $a_ix = b_i)$. However by slightly rotating everything, this degeneracy can always be removed.

Lecture #3: April 8, 1992
Scribe: Chris Wilson

1.4 Applications of LP Duality

As we have already seen, LP duality implies that:

$$\max\{cx|Ax \leq b\} = \min\{yb|y \geq 0, yA = c\}$$

For each inequality on the left, there is an inequality on the right restricting the corresponding y_i to be non-negative. When looking at the constraint between Ax and b on the left side, each inequality corresponds to a non-negative variable on the right, and each equality corresponds to an unrestricted variable on the right.

Last time we also talked about complementary slackness. We showed that when we have optimal solutions for both the primal and dual then if the restricted variable is non-zero, the corresponding inequality must hold with equality. We use x^* and y^* to denote optimal solutions. Complementary slackness, applied to the problem above gives

$$y_i{}^* > 0 \Rightarrow a_i x^* = b_i$$

where a_i is the i-th row of A.

The reverse implication isn't quite as strong. We can only say that if there exists an optimal x^* such that $a_i x^* = b_i$, then there exists *some* optimal y^* with $y_i{}^* > 0$.

1.4.1 Min-Cut/Max Flow Theorem

Now we will show that complementary slackness directly implies the max-flow/min-cut theorem. Let $G = (V, E)$ be a graph. For now we will consider the directed case. An instance of the problem consists of G, two distinguished nodes, $s, t \in V$ (the source and the sink, respectively), and a capacity function U from E to the non-negative reals. A feasible flow is a real-valued function X on E satisfying the following conditions:

$$
\begin{aligned}
X(e) &\le U(e) && \forall e \in E\\
-X(e) &\le 0 && \forall e \in E\\
\sum_{u:(u,v)\in E} X(uv) - \sum_{u:(v,u)\in E} X(vu) &\le 0 && \forall v \in V, v \ne t\\
-\sum_{u:(u,v)\in E} X(uv) + \sum_{u:(v,u)\in E} X(vu) &\le 0 && \forall v \in V, v \ne s
\end{aligned}
$$

The problem is to find a flow X which maximizes $\sum_{u:(s,u)\in E} X(su) - \sum_{u:(u,s)\in E} X(us)$, the flow out of the source.

An equivalent formulation is to add an arc from t to s with unlimited capacity (so we should really say that the capacities are functions from E to the non-negative reals or infinity). Then we can force the flow to be conserved at every node and simply maximize the backflow along the new arc from t to s. The new edge set, augmented with ts, is denoted E'. The conditions for a feasible flow are now (they are all in the form l.h.s. $\le$ r.h.s. so it will be easier to get the dual later)

$$
\begin{array}{rcll}
X(e) & \le & U(e) & \forall e \in E\\
-X(e) & \le & 0 & \forall e \in E\\
\sum_{u:(u,v)\in E'} X(uv) - \sum_{u:(v,u)\in E'} X(vu) & \le 0 & & \forall v \in V\\
-\sum_{u:(u,v)\in E'} X(uv) + \sum_{u:(v,u)\in E'} X(vu) & \le 0 & & \forall v \in V
\end{array}
\tag{1}
$$

The objective is to find X that satisfies 1 that maximizes $X(ts)$. Note that we have not explicitly added a constraint to insure the capacity of ts is non-negative because it's automatic in this problem anyway. Such a constraint could be added and the ultimate results would be the same.

A cut is simply a partition of the set V of vertices into two parts, S and $\overline{S}$. We denote the value of the cut by $\text{cut}(S, \overline{S})$ and define it to be the sum of the capacities of all arcs from S to $\overline{S}$. Note that the direction matters here, so $\text{cut}(S, \overline{S})$ is not the same as $\text{cut}(\overline{S}, S)$ in general.

Now consider all the cuts in which $s \in S$, $t \in \overline{S}$. Clearly the value of each such cut is an upper bound on all feasible flows. The Min-cut/Max-flow Theorem states that in fact the minimum value of all these cuts is equal to that of the maximum feasible flow.

Since the value of any cut is an upper bound on the value of any feasible flow, to prove the Min-cut/Max-flow Theoremit suffices to show that there exists a flow of value equal to that of some cut. The proof here will use duality (see [Ford and Fulkerson] for a more direct proof). Once the problem has been put in the correct form, the complementary slackness will automatically give the result we want.

In forming the dual, we have one dual variable for each equation in the primal. For equations that hold for each edge, there correspond vectors of length equal to the number of edges, and similarly for equations holding for every vertex. Here are the equations of the primal with their corresponding dual variables:

$$
\begin{array}{rcll|l}
X(e) & \leq & U(e) & \forall e \in E & Y_1(e) \\
-X(e) & \leq & 0 & \forall e \in E & Y_2(e) \\
\sum_{u:(u,v)\in E'} X(uv) - \sum_{u:(v,u)\in E'} X(vu) & \leq & 0 & \forall v \in V & Z_1(v) \\
-\sum_{u:(u,v)\in E'} X(uv) + \sum_{u:(v,u)\in E'} X(vu) & \leq & 0 & \forall v \in V & Z_2(v)
\end{array}
$$

The objective function to be maximized is $X(ts)$, which can be written in the form cX (X by itself denotes the column vector with components $X(e)$ indexed by $e \in E'$) by taking c to be the row vector $(0, 0, \dots, 0, 1)$. We have assumed here that we order the edges so that ts is the last one. Hence the dual has the form:

$$
\begin{array}{lrcll}
(2) & Y_1(uv) - Y_2(uv) + Z_1(v) - Z_1(u) - Z_2(v) + Z_2(u) & = & 0 & \forall uv \neq ts \\
(3) & Z_1(s) - Z_1(t) - Z_2(s) + Z_2(t) & = & 1 &
\end{array}
$$

Since the constraints of the primal are inequalities, the dual variables, Y_1, Y_2, Z_1, and Z_2 are all constrained to be non-negative.

Note that we could formulate the objective of this dual, but we don't need it since all we want is complementary slackness. The dual objective would be $U \cdot Y_1$. In some sense we could think of this as a length. $Z(v) = Z_1(v) - Z_2(v)$ is then a sort of distance. Equation 2 above becomes:

$$Y(uv) + Z(v) \geq Z(u)$$

This is the natural constraint on Y as edge-length when Z is distance.

Now we will take Y_1, Y_2, Z_1, and Z_2 as optimal solutions to the dual. We would usually write Y_1^*, Y_2^*, etc., but since we will only be considering optimal solutions here we can make the notation a bit simpler.

Now we will use the dual to construct a saturated cut. Define a subset V' of V as follows: $V' = \{v \in V | Z(v) \geq Z(s)\}$. Now consider any w and w' in V such that $w \in V'$, $w \notin V'$. We have:

$$Z(w) \geq Z(s) > Z(w') \tag{4}$$

and equation 2 gives

$$Y_1(ww') - Y_2(ww') + Z(w') - Z(w) = 0 \tag{5}$$

By equation 4, the sum of the last two terms in equation 5 is strictly negative. The second term, $-Y_2(ww')$ is non-positive, so for the equation to hold the first term, $Y_1(ww')$ must be positive, i.e. $Y_1(ww') > 0$. Hence by complementary slackness, the corresponding primal inequality must hold with equality. That inequality was

$$X(ww') \leq U(ww')$$

so we have that

$$X(ww') = U(ww')$$

This is true for all arcs ww' from V' to its complement.

Conversely, if we take any arc ww' to V' from its complement (so $w \notin V'$ and $w' \in V'$), we have

$$Z(w) < Z(s) \leq Z(w')$$

so the last two terms of equation 5 now have a strictly positive sum, and $Y_1(ww')$ is non-negative, so $Y_2(ww')$ must be strictly greater than zero this time. Complementary slackness says that the corresponding inequality,

$$-X(ww') \leq 0$$

holds with equality, so $X(ww')$ is zero.

Hence all arcs out of V' are at their capacities and all arcs into V' have zero flow, so the value of the flow must be equal to the capacity of this cut. This completes the proof of the Min-cut/Max-flow Theorem.

1.4.2 Doubly Stochastic Matrices

A doubly stochastic matrix is a matrix A with α_{ij} as the i, j coordinate satisfying the following conditions:

$$\begin{array}{rcl} \alpha_{ij} & \geq & 0 \\ \sum_j \alpha_{ij} & = & 1 \quad \forall i \\ \sum_i \alpha_{ij} & = & 1 \quad \forall j \end{array} \tag{6}$$

A permutation matrix is a special case of a doubly stochastic matrix in which all α_{ij} are integers (which means that there is a single 1 in each row and in each column and all other entries are zero). Note that these conditions imply that A must be square.

Theorem 14 A is doubly stochastic iff A is a convex combination of permutation matrices.

Proof: The "$\Leftarrow$" direction is trivial — any convex combination of a doubly stochastic matrix is doubly stochastic.

The proof of the other direction will be by induction on the dimension n of the matrix A. We will think of the class of all n by n doubly stochastic matrices as the n^2-dimensional vectors that satisfy 6. Notice that this set of vectors is a polytope that we will denote by P.

It is sufficient to prove that the vertices of P are convex combinations of permutation matrices since everything else is a convex combination of the vertices.

There are $n^2 + 2n$ constraints in 6. Notice that the row and column-constraints are dependent, and hence we have at most $n^2 + 2n - 1$ independent inequalities. The dimension of the space is n^2, so n^2 constraints define a vertex. Of these n^2, at most $2n - 1$ can be summation of rows or columns and the other $n^2 - 2n + 1$ must be of the form $\alpha_{ij} \geq 0$. The constraints that define a vertex hold with equality for that vertex, so for each vertex, there are at least $n^2 - 2n + 1$ constraints of the form $\alpha_{ij} \geq 0$ holding with equality. Hence at a vertex at least $n^2 - 2n + 1$ of the α_{ij} must be zero, so at most $2n - 1$ can be non-zero.

At a given vertex, if every column had at least two non-zero α_{ij}, at least $2n$ such non-zero entries would be required. Hence this cannot be the case; every vertex must have one column with only a single entry non-zero. Since the sum of entries in the column is one, that single non-zero entry must be one. Hence there are i', j' such that $\alpha_{i'j'} = 1$ and the rest of row i and the rest of column j must be zero. If we permute the indices so that this 1 is in the upper left corner, we have a doubly stochastic matrix of the form

$$A = \begin{pmatrix} 1 & 0 \\ 0 & A' \end{pmatrix}$$

Now the fact that A is doubly stochastic implies that A' is doubly stochastic of dimension one smaller. Hence, by inductive hypothesis, A' is a convex combination of permutation matrices. Adding a 1 to the upper left hand corner of each of these and increasing their dimensions by ones gives a convex combination of permutation matrices that form A. Hence we have shown that each vertex of P is a convex combination of permutation matrices and the theorem follows. ∎

Now we can use the theorem we just proved to get a nice property of bipartite graphs.

Lemma 15 A regular bipartite graph has a perfect matching (regular means that all the nodes have the same degree).

Proof: Define A as follows:

$$\begin{aligned} \alpha_{ij} &= \frac{1}{\text{degree}} \text{ if } (i,j) \in E \\ &= 0 \text{ otherwise} \end{aligned}$$

By inspection, A is doubly stochastic so we can apply our previous result; A is a convex combination of permutation matrices. Hence there is a permutation matrix β_{ij} such that for all i and j, $\beta_{ij} \neq 0 \Rightarrow \alpha_{ij} \neq 0 \Rightarrow (i,j) \in E$. Hence β_{ij} corresponds to a perfect matching. ∎

Lecture #4: April 13, 1992
Scribe: Mario J. Córdova

2 Max Flow and Min Cuts

The goal of the next several lectures is to describe two new algorithms for finding a minimum cut in a given graph. We will start with presenting several definitions, then we will present the push/relabel max-flow algorithm, and then we will show how to use this algorithm to comute a min-cut in the graph.

It is important to notice the difference between the definition of a *min-cut* and the *min* $s-t$ cut. Let $G = (V, E)$ be an undirected graph with node set V and edge set E. (We can regard an undirected graph as a symmetric directed graph.) We denote the size of V by n and the size of E by m. We assume that $m = \Omega(n)$. Each edge $e \in E$ has a capacity $u(e)$.

Definition 16 A *cut* is a partition of the node set $\{S, T\}$ where $V = S \cup T$, $S \cap T = \emptyset$, and S, $T \neq \emptyset$. A *minimum cut* is a cut with minimum capacity. A *minimum s-t cut* is a cut with minimum capacity s.t. s belongs to S and t belongs to T.

The following theorem establishes a relation between the min cut and the min s-t cut problems.

Theorem 17 A min cut can be found in $(n-1)$ min s-t cut computations.

Proof: Given the nodes s and t, either (i) s and t are on opposite sides of a min cut, or (ii) s and t are on the same side. In Case (i), min s-t cut = min cut, and we are done. In Case (ii), min cut $\leq$ min s-t cut.

Let $v_1, \ldots, v_n$ be the nodes in the graph. Compute min s-t cut for each of the pairs v_1, v_k. Clearly, the number of these min s-t cut computations is at most $n-1$. Now, in the min cut, V is partitioned into S and T, where $S \cup T = V$. Hence, v_1 is either in S or T; without loss of generality we assume $v_1 \in S$. Since $T \neq \emptyset$, $\exists k, 2 \leq k \leq n$ such that $v_k \in T$. In this case, v_1 and v_k are on the opposite sides of the min cut, so min cut = min $v_1 - v_k$ cut, and we are done. ∎

In fact, it is not known whether computing the min cut is harder than computing the min s-t cut, or vice versa. The best currently known algorithm to compute the former is faster than the best known algorithm to compute the latter. In any case, as shown above, to compute min cut in polynomial time, we only have to know how to compute min s-t cut in polynomial time. The rest of the lecture will show an efficient algorithm to obtain the latter.

2.1 Max Flow Algorithm

This sections is a review of the paper "A new Approach to the Maximum Flow Problem" by Goldberg and Tarjan [3]. We talk about directed flows. Given a directed flow, we can easily convert it into a symmetric flow which corresponds to undirected flow of the same value.

2.1.1 Definitions

In the previous lecture we defined a *flow* as a function on the arcs satisfying capacity, anti-symmetry and conservation constraints. The *value* of a flow f is the net flow into the sink. A *maximum flow* is a flow of maximum value. We also observed that given s and t, the maximum flow in a network equals to the capacity of a min s-t cut. Today we introduce several new definitions.

Definition 18 A *preflow* is a flow with relaxed conservation constraints for all $v \neq s, t$:

$$e_f(v) = \sum_{(u,v)\in E} f(u,v) - \sum_{(v,w)\in E} f(v,w) \geq 0.$$

Definition 19 The *residual capacity* u_f of an arc (v,w) is $u_f(v,w) = u(v,w) - f(v,w)$.

Observe that there are two ways an arc (v,w) can have positive residual capacity: Either (v,w) is an arc with flow less than its capacity (arc (v,w) is said to be *unsaturated*), or (w,v) is an arc with positive flow.

Definition 20 The *residual graph* E_f is the graph composed of the arcs whose residual capacities are greater than zero. We say that t (s) is *reachable* from v if it is reachable from v in the current residual graph.

Definition 21 A *valid labeling* d is a function from the nodes to the nonnegative integers, such that $d(s) = n$, $d(t) = 0$, $d(v) \leq d(w) + 1$ for every residual arc (v,w).

Roughly speaking, the labeling function is a locally meaningful estimate on the distance from the nodes to the sink.

Definition 22 We say that a node v is *marked* if it can not reach the sink t in the residual graph. A node $v \neq t$ is *active* if $e_f(v) > 0$ and v it is not marked. The arc (v,w) is *admissible* if $u_f(v,w) > 0$ and $d(v) = d(w) + 1$.

Push(v, w)

Applicability: v is active and (v, w) is admissible.
Action: $\delta \leftarrow \min \{e_f(v), u_f(v, w)\}$.
Increase $f(v, w)$ by δ.

Figure 2: The push operation

Relabel(v)

Applicability: v is active and $\forall w$ such that $(v, w) \in E_f$, $d(w) \geq d(v)$.
Action: $d(v) \leftarrow 1+ \min \{d(w) | (v, w) \in E_f\}$. (If this min is empty, mark v.)
If $d(v) \geq n$, mark v.

Figure 3: The relabel operation

2.1.2 The Push-Relabel Algorithm

We want to compute a maximum flow in a given network. The algorithm solves this problem by manipulating a preflow f on the network. The preflow algorithm works by examining nodes other than s and t with positive flow excess and pushing excess from them to nodes estimated to be closer to the sink s, with the goal of getting as much excess as possible to t.

The maximum flow algorithm begins with a preflow f that is equal to the arc capacity on each arc leaving the source and zero on all other arcs. (Hence, after the first step, the source is marked.) In the beginning, the labeling is a valid labeling where $d(v) = 0$ for all nodes except for the source whose label is n. The algorithm then repeatedly performs, in any order, the basic operations, ***push*** and ***relabel***, appearing in Figures 2 and 3. The algorithm maintains a preflow and a valid labeling function at any given time. The preflow is updated using the push operation, and the labels are updated using relabel. When there are no active nodes, the algorithm terminates.

Intuitively, when relabel is applicable it means that the previous estimate on the distance from v to the sink was too low.

Observe that the algorithm may mark nodes explicitly. To stay consistent with the definition of a marked node above (a marked node was defined as a node from which the sink is not reachable), we need that when the algorithm explicitly marks a node, the sink is not reachable from it. This property indeed follows from the fact that the algorithm maintains valid labeling at all times.

To compute the running time of the algorithm, observe that when a node is relabeled, its label increases. Therefore, there can be at most n relabelings for each node. Every time a node is relabeled, its arc list must be scanned. Consequently, the total cost of the relabeling operations

Discharge(v)

Applicability: v is active.
Action: Repeat
push or relabel v
if w becomes active during this push/relabel operation,
then add it to the rear of Q.
until either $e_f(v) = 0$ or v is relabeled.
If v is still active add it to the end of the queue.

Figure 4: The discharge operation

is $O(nm)$. Next define a push(v, w) as a *saturating* push if after the push $u_f(v, w) = 0$. It is easy to see that between every two saturating pushes ¿From v to w on the arc (v, w), both v and w must be relabeled. Therefore, we have a total of $O(nm)$ saturating pushes. The number of nonsaturating pushes is however $O(n^2m)$ (we won't prove it here).

To improve the number of nonsaturating pushes, we exploit the freedom in selecting nodes for push/relabel operations by ordering them in a specific way. At the low level, the operations are ordered by the *discharge* operation described in Figure 4. The operation consists of applying push/relabel operations to an active node until its excess becomes zero or the label of the node increases. Note that there is at most one nonsaturating push per discharge.

It remains to specify strategies for selecting the next active node to be discharged. One such strategy if to select an active node with the maximum distance label. Note that if this strategy is used, then after a node is discharged, it is not discharged again until at least one relabeling occurs. It follows that there are at most $O(n^3)$ discharges and therefore at most $O(n^3)$ nonsaturating pushes. Using the bucket data structure, the algorithm can be implemented to run in $O(n^3)$ time. Cheriyan and Maheshwari [1] improve the bounds on the number of nonsaturating pushes and the running time to $O(n^2\sqrt{m})$.

Proof of Correctness

This section sketches the proof of correctness of the above algorithm. More details are in the paper.

Observe that as long as there is some active node with zero excess, either push or relabel is applicable. Therefore, when the algorithm terminates, all unmarked nodes except the sink must have excess 0. Recall that a node is marked iff the sink is not reachable from it. Thus when the algorithm terminates, the nodes are partitioned into two sets: the first contains nodes from which the sink is not reachable, and the other contains the remaining nodes. All nodes in the second set except for the sink have excess zero and each arc from the first to the second set is saturated.

The two sets define a cut, say cut C. We claim that this cut is a minimum cut and hence the maximum flow value is the current net flow into the sink. By the flow decomposition theorem, flow excess at the nodes other then t (all of which are marked) can be moved back to s without changing the flow across our cut. Clearly, the cut C can be found in linear time (for example, by starting a breadth first search from t in the residual graph whose arcs are reversed).

Use of Dynamic Trees

The algorithm mentioned in the previous subsection can be improved with the introduction of the *dynamic tree* data structure of Sleator and Tarjan. Each tree arc is regarded as directed from child to parent. The trees support a number of useful operations, among which are:

1. *find-root(v):* Find and return the root of the tree containing node v.
2. *find-min(v):* Find and return the ancestor w of v such that the residual flow on the arc from w to its parent in the tree is minimum. In case of a tie, choose the node close to the root.
3. *link(v, w):* Combine the trees containing nodes v and w by making w a parent of v.
4. *cut(v):* Break the tree containing v into two trees by deleting the arc from v to its parent.

The cost of l dynamic trees operations is $O(l \log k)$ where k is the maximum size of a tree.

When used by our algorithm, the trees maintain two invariants: (i) All arcs in the tree are admissible; and (ii) All flow excesses are at tree roots (except for when we are in the middle a push/relabel operation).

By using appropriate tree operations we can push flow along an entire path in a tree, either causing a saturating push or moving flow excess from some node in the tree all the way to the tree root. The heart of the algorithm is the procedure *send(v)*. It pushes excess from a nonroot node v to the root of its tree, cuts arcs saturated by the push, and repeats these steps until $e_f(v) = 0$ or v is a tree root.

At the top level, the dynamic tree algorithm is exactly the same as the first-in first-out algorithm described above. We maintain a queue Q of active nodes and repeatedly perform discharging operations until Q is empty. However, the push/relabel operations are replaced with tree-push/relabel operations appearing in Figure 5.

To compute the running time of the algorithm, first observe that a cut operation is performed on (v, w) only when we either relabel w or perform a saturating push on (v, w). Therefore, there are $O(nm)$ cuts. The number of links performed is no larger than the number of cuts plus $n-1$, which is bounded again by $O(nm)$. The number of send operations is at most the number of links which gives $O(nm)$ again. Consequently, the algorithm requires $O(nm)$ tree operations, yielding a total of $O(nm \log n)$ running time. A more careful implementation in which the size of the trees is bounded by $k = n^2/m$, gives a time bound of $O(nm \log(n^2/m))$.

Tree-push/relabel

Pick an active node v
If $\exists$ an admissible arc (v, w), then link(v, w) and send(v);
Otherwise, relabel v, and cut all tree arcs into v.

Figure 5: The tree-push/relabel operation

Lecture #5: April 15, 1992
Scribe: Kathleen Fisher

2.2 Cut Algorithms: Hao-Orlin

In a previous lecture, we showed that the min-cut of an undirected graph can be found by computing $n-1$ s-t cuts. With the Hao-Orlin algorithm [?], we will still compute $n-1$ s-t cuts, but now the running time of the entire algorithm will be comparable to the running time required to compute a single s-t cut. We will take as our starting point the push-relabel algorithm of last time; however, we need to make some modifications to it.

Modifications to the Push-Relabel Algorithm

1. Set the distance label for the original sink node to $2n$ instead of n. That is, $d(s) = 2n$. Since we are going to keep the distance labels throughout the $n-1$ iterations of the algorithm, we have to make this change to prevent the distance label of any sink from ever getting within n of $d(s)$.

2. After computing an s-t cut, we don't really have to merge s and t. Instead, relabel $d(t)$ to $2n$ and saturate all edges leaving t. These actions effectively make t another source node.

3. After relabeling t, we have to select the next sink node, t_i. In choosing t_i, we need to make sure that the distance constraints given last time hold. In particular, for any sink node t, we have to guarantee that $d(s) - d(t) \geq n$. Let t_i represent the i-th sink chosen, and $\overline{d}(t_i)$ be t_i's distance label at the time it became a sink. Initially we set

   $$\overline{d}(t_i) = 0 \tag{7}$$

   Each sink that we pick will satisfy the property that

   $$\overline{d}(t_i) \leq 1 + \max_{j<i}(\overline{d}(t_j)) \tag{8}$$

Equations (7) and (8) make it easy to prove by induction on i that $\forall\ 1 \le i \le n,\ \overline{d}(t_i) < i$. Since $d(s) = 2n$, we have that

$$\forall\ 1 \le i \le n,\ d(s) - \overline{d}(t_i) \ge 2n - 1 \ge n$$

Thus as long as each sink node that we select satisfies (8), the required distance constraints will hold throughout the algorithm.

4. With the dynamic tree implementation, all dynamic tree edges into a sink must be cut when the sink becomes a source node for the next iteration.

Selection of the Next Sink

Consider the following graph:

The numbers labeling the edges are the capacities; those labeling the nodes are the distance labels. If t is selected as the next sink, then the push-relabel algorithm will terminate in the following state:

At this point, to restart the algorithm, $d(t)$ will become $2n$. We need a sink node whose distance label is less than or equal to one. No such node exists: we are stuck. This example illustrates the care with which sink nodes must be selected.

Auxiliary Data Structures: Occupancy Array

To select the next sink, we are going to use a occupancy array. The occupancy array keeps track of how many nodes are labeled with each distance label. If only one node v has a given label, and v is about to be relabeled, we know not only that v can't reach the sink, but also that no nodes with a higher label than v's can reach the sink either. This array gives us a way to detect sets of nodes, none of which can reach the current sink.

Auxiliary Data Structures: Stack of 'Dormant' Nodes

We are also going to maintain a stack of nodes to help restart the algorithm. All source nodes are at the bottom of the stack. Each time the gap relabeling scheme detects a set of nodes V_i

that can't reach the current sink, the distance labels of all nodes in V_i are frozen and the set is pushed onto the stack. Note that the node whose imminent relabeling created V_i is *not* actually relabeled. Pictorially, we have:

where T denotes the set of nodes not on the stack.

Properties of the Node Sets on the Stack

In running an iteration of the algorithm, the nodes in the sets on the stack are nodes that we do not look at. They were placed on the stack because they could not reach any node in T at the time they were stacked. Since all sets placed on the stack after a set V_i was in V_i's T set, nodes in any set on the stack cannot reach any nodes above them on the stack either.

Lemma 23 When a set of nodes V_i is stacked, there are no residual arcs from V_i to T.

Proof: Suppose there were a residual arc from some node $w' \in V_i$ to some node $u' \in T$. We know that $\forall w \in V_i, d(w) \geq d(v)$ and $\forall u \in T, d(u) < d(v)$ where v is the node that caused V_i's creation. Thus $d(w') > d(u')$, and the gap relabeling scheme could not have raised $d(w')$ above d(u'). ▮

Since there are no residual arcs from any node on the stack to any node in T, we can ignore nodes on the stack when computing the value of the current s-t cut. In the dynamic tree implementation, when a set is frozen, there are no dynamic tree edges leaving the sets because the dynamic tree edges are admissible.

Selection of New Sink: The Algorithm

To pick a new sink to replace t_i, we do the following:

1. Delete t_i from T
2. If $T = \emptyset$, pop the top set from the stack and make T equal to it.
3. Select t_{i+1} to be the element of T with the smallest distance label.

Lemma 24 This algorithm for selecting the sink t_i satisfies the constraint $\overline{d}(t_i) \leq 1 + \max_{j<i} \overline{d}(t_j)$

Proof:

Case 1: $T \neq \emptyset$ (after t_i is removed)

Consider the occupancy array just before t_i is relabeled to $2n$. Suppose for a proof by contradiction that there are no nodes with distance labels $\overline{d}(t_i)$ or $\overline{d}(t_i) + 1$. (Any nodes with labels smaller than these must have been selected before t_i.) In this case, there is a gap in the occupancy array. Since $T \neq \emptyset$, there must be some node $u \in T$ such that $d(u) > \overline{d}(t_i) + 1$. To produce the gap between $d(u)$ and $d(t_i)$, a node w with label $\overline{d}(t_i) + 1$ must have been relabeled when it was the only node with this distance label. Such a relabeling, however, cannot occur: instead, w and all nodes with higher distance labels would have been frozen on the stack. Thus such a gap cannot exist and there must be some node $x \in T$ such that $d(x) \leq \overline{d}(t_i) + 1$.

Case 2: $T = \emptyset$

In this case, the top set on the stack, say V_i, is going to be unfrozen. Consider the occupancy array at the time that V_i was pushed onto the stack. Let v be the node with the unique distance label d_v that caused V_i to be frozen. Note that there must be at least one node $u \in T$ such that $d(u) = d_v - 1$. Since the distance labels of V_i were frozen when V_i was pushed, $d(v) = d_v$. Note that the algorithm will select all nodes in T before unfreezing V_i, and that distance labels never decrease. Thus $\max_{j<i+1} \overline{d}(t_j) \geq d(u)$. Since $v \in V_i$ and $d(v) = d(u) + 1$, $d(v) \leq 1 + \max_{j<i+1} \overline{d}(t_j)$. Thus v satisfies the distance constraint. Since v has the smallest distance label of any node in V_i by V_i's construction, v the algorithm will select v as t_{i+1}. ∎

Time Complexity

With these modifications, the time complexity analysis done last class for the single s-t cut problem goes through for the min-cut problem. Since we only do $O(n^2)$ work maintaining the stack to restart the push-relabel algorithm, that work can be charged to relabeling as well. Thus we have an $O(n^3)$ algorithm for the min-cut problem. Using the dynamic tree implementation, a bound of $O(nm \log(n^2/m))$ can be obtained.

2.3 Min Cut Algorithm of Nagamochi and Ibaraki

Before describing the minimum-cut algorithm of Nagamochi and Ibaraki [?], it is interesting to note that this algorithm dos not seem to have anything to do with flows. It achieves a bound of $O(nm + n^2 \log(n))$, which is slightly better than the bound of the previous section. The intuition behind this algorithm's improvement over algorithms that run a max-flow algorithm $n - 1$ times is that some s-t cuts may be difficult to compute. The algorithm described in the previous section picks a source and a sink at each iteration, so it might be stuck computing a difficult s-t cut. The Nagamochi-Ibaraki algorithm, however, picks pairs of nodes as a source and a sink after it already knows the min-cut between them.

The Edge Connectivity Problem

We are first going to solve a simplified version of the min-cut problem where all edge capacities are 1. This problem is called the edge-connectivity problem. It arises in reliability theory, for example: how many edges in a network can fail and have the network remain connected?

We will show an $O(nm)$ bound on the edge connectivity problem, which we will then improve to $O(n^2\lambda)$, where λ is the value of the min-cut of the graph. After developing the algorithm for the edge connectivity problem, we will generalize it to work in the case where edge capacities can be any positive integer. To facilitate this generalization, we will allow multiple edges between nodes.

Tools for the Algorithm: Maximal Forests

Given a graph $G = (V, E)$, compute a maximal forest F_1 embedded in G, and then delete the edges of F_1 from G. Repeat this process until G contains no more edges. This process produces a sequence $F_1, F_2, \ldots, F_t$ of maximal forests. Define $(G_i = (V, \bigcup_{j\leq i} F_i))$.

Properties of Maximal Forests

First some notation: define $\lambda(x, y; G)$ to be the connectivity between nodes x and y in graph G.

Lemma 25 below will enable us to show the improved bound of $O(n^2\lambda)$ for the edge connectivity problem.

Lemma 25 $\lambda(x, y; G_i) \geq \min(i, \lambda(x, y; G))$.

Proof: The lemma follows from the fact that if $\lambda(x, y; G_j) = \lambda(x, y; G_{j+1})$ for some $1 \leq j < t$, then $\lambda(x, y; G_j) = \lambda(x, y; G)$. To see the latter, assume for contradiction that $\lambda(x, y; G_{j+1}) < \lambda(x, y; G)$.

Let $A \subset E$ be a minimal set of edges such that adding A to G_{j+1} increases the connectivity between x and y, and let $a \in A$. By the definition of A, $a \notin \bigcup_{r\leq j+1} F_r))$, and $F_{j+1} \bigcup \{a\}$ is a forest. This contradicts the maximality of F_{j+1}. ∎

Lemma 26 If $x, y \in F_i$, then $\lambda(x, y; G) \geq i$.

Proof: If a path connects x and y in F_i, then by the construction of the maximal forts, a path must have connected x and y in all previous forests. Thus each maximal forest up to F_i must contribute at least one to $\lambda(x, y; G)$. Hence, $\lambda(x, y; G) \geq i$. ∎

The Algorithm: An Overview

This algorithm constructs a set of maximal forests with the properties described above. From these forests, it can immediately determine a min s-t cut for some s and t. (In this way it avoids selecting a pair of nodes whose s-t cut is difficult to compute; it selects pairs of nodes whose s-t cut it has already computed.) If the current s-t cut is the smallest seen so far, the cut and its nodes are stored for future reference. The nodes s and t are merged, and the process is repeated, for a total of $n-1$ iterations. At the end of that time, the smallest cut calculated is guaranteed to be the min-cut of the graph.

The Algorithm: The Details

The algorithm works as follows:

Input: An undirected graph $G = (V, E)$.

Output: A set of maximal forests $F_1, F_2, \ldots, F_k$.

Mark all nodes as unscanned;
Mark all edges as unscanned;
Make all edges undirected;
$\forall w \in V$ set $r(w) = 0$
While unscanned nodes remain {
    Select unscanned node v with largest r-value: $r(v)$;
    Mark v as scanned;
    For all unscanned $\{v, w\} \in E$ {
        Mark $\{v, w\}$ as scanned;
        Direct $\{v, w\}$ from v to w;
        $r(w) := r(w) + 1$;
        Insert $\{v, w\}$ into $F_{r(w)}$;
    }
}

Note that scanning a node is an atomic operation: once a node is selected, all unscanned incident edges must be scanned before another node can be selected. Note also that when the algorithm terminates, each node and each edge will have been scanned exactly once, and each edge will have been placed in exactly one forest.

Lemma 27 The F_i constructed by the above algorithm are maximal forests of the given graph G.

Proof: First show that the F_i's constructed by the algorithm are forests, and then that they are maximal with respect to the subgraphs.

Notice that there are no cycles in any of the F_i. Suppose there were. Consider the first node in the cycle that the algorithm scanned, say node v. When v was scanned, all edges incident on it were directed out of it, so no cycle can contain v. Thus F_i can contain no cycles. Note also that in each of the F_i, the in-degree of every node is at most one. Hence the F_i are indeed forests.

Suppose the forests are not maximal with respect to the subgraphs for a proof by contradiction. Then there exist two trees T_1 and T_2 in a forest F_i such that there exists an edge {v,w} in a forest F_j such that $j > i$ and $v \in T_1$ and $w \in T_2$. The algorithm puts edges into the forests one by one. Suppose without loss of generality that the first edge it put into F_i was in T_1. That is, a node from T_1 was scanned first. Because of the way the algorithm manages the r-values, all of the nodes in T_1 will be scanned before any of the nodes in T_2. In particular, v will be scanned before w, so $\{v, w\}$ will be directed toward w. Consider what $r(w)$ can be when v is scanned. If $r(w) \geq i$, then there is already an edge incident on w in F_i. Since no nodes in T_2 have been scanned, this edge must have come from T_1, but that contradicts the assumption that T_1 and T_2 aren't connected. Hence $r(w) < i$. Thus when v is scanned, $\{v, w\}$ cannot be inserted into F_j for any $j > i$. ∎

So Where's the Min-Cut?

Surprise! After running this algorithm, we can easily find a min s-t cut. Consider the last edge to be scanned: $\{v, w\}$. Suppose we scanned this edge from node v and that we put the edge into F_i. We know that w couldn't have been scanned already. (If it had been, $\{v, w\}$ would have already been scanned from w.) Since $\{v, w\}$ was the last edge scanned, all edges incident on w must be directed toward w. Because $\{v, w\}$ was inserted into F_i, w must have i edges incident on it. Thus the s-t cut between v and w must be at most i. By Lemma 26, $\lambda(v, w; G) \geq i$, so the s-t cut between v and w is exactly i.

After the algorithm finds the s-t cut between v and w, these two nodes are contracted, and the whole process is repeated until only one node remains.

Complexity

The complexity of a single iteration of this algorithm is just $O(n + m)$ since we scan every edge and every node exactly once and do a constant amount of work for each edge and node. The only possible increase in complexity comes when we select the node with the largest r-value. We can avoid this problem by keeping the nodes in an array of buckets indexed by r-values and by keeping track of the largest active r-value. The only time we could run into trouble is scanning down the array to find the next largest r-value after processing the largest. Since $\sum_{v \in V} r(v) \leq m$, however, we will never have to move backwards more that m times. Thus this structure allows us to track the largest r-values in $O(m)$ time.

Since we are assuming that the graph is initially connected (otherwise $\lambda = 0$, trivially), $O(m+n)$ is equivalent to $O(m)$. The running time for the entire algorithm (all $n-1$ iterations) is thus $O(nm)$.

Improved Bound for the Edge Connectivity Problem

Notice that Lemma 25 tells us that if a graph G's connectivity $\lambda < k$ for some constant k, then the connectivity of G is the same as the connectivity of G_k. This fact allows us to find λ in $O(n^2\lambda+m)$. Given k, we build G_k using one iteration of the above algorithm. This construction requires $O(n+m)$ or $O(m)$ since we are assuming that G is connected. The above algorithm is then used to determine the min-cut of G_k. Since G_k has at most kn edges by its construction, the time to find its min-cut is just $O(n^2k)$. By Lemma 25, if $\lambda(G_k) < k$, then $\lambda(G_k) = \lambda(G)$ and we are done. Otherwise, we double k and repeat the process. Because of k's exponential increase, the running time of the final iteration dominates the process. Since $\lambda = O(\text{last } k)$, we have a bound of $O(n^2\lambda + m)$.

Weighted Case: Capacities are Positive Integers

We will now generalize the above algorithm to the case where each edge i can have an arbitrary positive integer edge capacity c_i. Given a graph G and a capacity for each edge, we first modify G by replacing each edge with capacity c_i by c_i edges with capacity 1. Our edge connectivity algorithm, when run on the resultant graph G', will produce the correct answer; however, G' may have exponentially many edges, giving an unacceptable running time. Notice though, that nothing in the original algorithm specified the order in which edges incident to a node must be considered. We can now specify that the c_i edges must be considered in sequence and handle them together in a constant amount of time.

Complexity of the Weighted Case

As in the edge connectivity problem, the only problem arises in tracking the node with the largest r-value. Because r-values can now increase by arbitrarily large amounts, however, the bucket scheme used above no longer works efficiently. Instead, we must use a heap data structure. Regular heaps give a bound of $O(nm\log(n))$. Fibonacci heaps offer an improvement. The two operations we need are *ExtractMax* and *IncreaseKey(amount)*. The algorithm invokes *ExtractMax* n times for a total amortized cost of $O(n\log(n))$ and *IncreaseKey(amount)* m times for a total amortized cost of $O(m)$. With this implementation, then, the total cost is $O(n^2\log(n) + nm)$.

Rational Capacities

The algorithm easily handles the case where the capacities are rational numbers: increment the r-values by the rational edge capacities and otherwise proceed the same. The only question is whether or not the cut this process produces is the min-cut of the graph. We claim that it is. Consider what would happen if the rational capacities were first scaled by their least common denominator. Because of the way the algorithm works, it would still find the same cut. Since in this case the capacities are all positive integers and we know the algorithm works on positive integer edge capacities, the algorithm must work on rational edge capacities as well.

Real Capacities

Now let us consider what happens when the edge capacities c_i can be real. We will show that the same algorithm will still produce the min-cut of the graph. To do this, we will show that there exists a set of rational edge capacities q_i such that the algorithm makes exactly the same decisions given the q_i as it does given the c_i. Thus the cut determined in the two cases is the same. Since we already know that the algorithm finds the min-cut in the rational case, it must also find it in the real case.

To find these rational capacities q_i, we will treat the edge capacities as variables, x_i. All the algorithm does with these capacities is add them and compare linear combinations of them to compute new r-values and to select the next node to scan. (Note that the r-value of a node w is now the sum of the capacities of all arcs directed into w.) The results of these comparisons determine a path through the decision tree of the algorithm. Each node represents a decision point where the algorithm compares the r-values of two nodes. The leaves are the cuts determined by the path taken to reach them.

If we follow the path down the tree determined by the real capacities c_i, we will certainly end up at a cut. The only possibility for error is that we don't end up at the min-cut. That is, the algorithm produces a cut C_1 and there exists a cut C_2 such that $cC_1 > cC_2$ where c is a vector of the real capacities. We can scale the capacities in the above inequality to produce the modified inequality:

$$c'C_1 \geq c'C_2 + 1 \tag{9}$$

Given these edge capacities c', the algorithm will still produce the same cut.

Now we trace the path taken by the algorithm through the decision tree when run with the scaled capacities c'. Collect the inequalities that hold between the linear combinations of the variable capacities at each node on the path to the cut determined, C_1. The coefficients of the x_i's must be integral since each non-zero coefficient corresponds to the selection of an edge of a particular capacity for a cut. To these inequalities, add the inequality determined above in Equation 9:

$$xC_1 \geq xC_2 + 1$$

where x is the vector of capacity variables. Note that all coefficients are integral here as well. Taken together, these inequalities give us a system of linear equations:

$$Ax \geq b$$

where all entries in both A and b are integral.

We know that the original real coefficients c satisfy this linear system, so it must define a non-empty polyhedron. Consider any vertex v of this polyhedron. Let x_v be the solution at v. At the vertex we have that:

$$A'x_v = b'$$

where A' is a submatrix of A and b' is a subcolumn of b. Since A' forms a basis, it must have an inverse: ${A'}^{-1}$. Thus

$$x_v = {A'}^{-1}$$

By Kramer's Rule, ${A'}^{-1}$ has rational coefficients. Hence x_v is a rational solution to the linear system.

Thus if the rational capacities x_v were assigned to the edges of G, the algorithm would follow exactly the same path through the decision tree as it does with the real capacities c. Therefore the algorithm will find the same cut, C_1, in both cases. This contradicts the correctness of the algorithm for rational capacities.

Lecture #6: April 20, 1992
Scribe: Rakesh Nigam

2.4 Gomory-Hu Trees

Let $G = (V, E)$ be an undirected graph with node set V of size n and edge set E of size m. Each edge $e \in E$ has a capacity $u(e)$. In this lecture we will consider ways to compute the capacity of a min cut between *every* two nodes in the graph. (This in turn gives us the values of the maximum flows between all pairs of nodes.) Clearly this can be accomplished in polynomial time, by solving $n(n-1)/2$ min cut problems (one for each pair of distinct nodes). The solution obtained by Gomory and Hu, however, requires solving only $n-1$ min cut problems.

Definitions and Properties of Gomory-Hu Trees

Definition 28 A *Gomory-Hu tree* is a weighted spanning tree T where for every edge (x, y), the two components of $T - \{x, y\}$ give a minimum cut between x and y in G, and the weight of the edge gives the cut capacity.

Denote by $c(x, y)$ the capacity of a min $x-y$ cut.

Lemma 29 Let T be a Gomory-Hu tree. Then for each pair of nodes x, y in G, $c(x, y) = min\{v_{i1}, v_{i2}, \ldots, v_{ir}\}$, where the v_{ij}'s are values of the series of edges connecting the two points in T. Moreover, the edge that gives the minimum also defines a min cut between x and y (that is, the two components we get when removing this edge, determine a min $x-y$ cut).

Proof: Each v_{ij} as above is the capacity of some cut between x and y. Therefore, clearly, $c(x, y) \leq min\{v_{i1}, v_{i2}, \ldots, v_{ir}\}$. Next we show that $c(x, y) \geq min\{v_{i1}, v_{i2}, \ldots, v_{ir}\}$. Let $\{X, Y\}$ be a min $x-y$ cut. Consider the nodes along the path from x to y in T. There must be two nodes a, b adjacent in this path such that $a \in X, b \in Y$. But then the capacity of min $a-b$ cut is the same as the one of min $x-y$ cut, and hence $c(x, y) = c(a, b)$, which is the weight of edge (a, b) in the tree, and which is $\geq min\{v_{i1}, v_{i2}, \ldots, v_{ir}\}$. Finally, since the two components of $T - (a, b)$ define a cut between x and y whose capacity equals to the capacity of min $x-y$ cut, these components define a min $x-y$ cut, and we are done. ∎

Clearly if a weighted spanning tree T satisfies the property that for each pair of nodes x, y in G, $c(x, y) = min\{v_{i1}, v_{i2}, \ldots, v_{ir}\}$, where the v_i's are values of the series of edges connecting the two points in T, and the edge that gives the minimum also defines a min $x-y$ cut, then this tree is a Gomory-Hu tree. Lemma 29 shows that also the opposite direction is true. Consequently, the above lemma provides an alternative definition for Gomory-Hu trees.

Finally, Lemma 29 immediately implies that given a Gomory-Hu tree, a min cut between any two nodes can be found in $\leq n-1$ steps. Moreover, the existence of these trees has the interesting consequence that among the $n(n-1)/2$ maximum flow values between all pairs of nodes there exist at most $n-1$ different values (namely, the values associated with the edges of T).

The Ideas Behind the Gomory-Hu Trees

The construction of the trees will be based on the following lemma:

Lemma 30 Let $\{S,T\}$ be a minimum cut between s and t. Let $x,y \in S$, $x \neq y$. Then there is a min $x-y$ cut $\{X',Y'\}$ such that either $X' \subseteq S$ or $Y' \subseteq S$.

Proof: Assume we are given a min $x-y$ cut, denoted by $\{X,Y\}$, that violates the lemma. We will show how to modify the cut to get a min $x-y$ cut $\{X',Y'\}$ that satisfies the lemma.

Without loss of generality assume s is in X. Consider two cases: (i) $t \in Y$; (ii) $t \in X$.

In Case (i), the cut $(T\cap Y, S\cup X)$ separates s and t, so $c(S,T) \leq c(T\cap Y, S\cup X)$, or

$$c(S\cap X, T\cap X) + c(S\cap Y, T\cap X) + c(S, T\cap Y) \leq c(S, T\cap Y) + c(T\cap X, T\cap Y)$$

and

$$c(S\cap X, T\cap X) \leq c(T\cap X, T\cap Y).$$

Because edge capacities are nonnegative, this implies

$$\begin{aligned} c(S\cap X, T\cap X) + c(S\cap X, Y) &\leq c(S\cap X, Y) + c(T\cap X, T\cap Y) \\ &\leq c(T\cap X, S\cap Y) + c(S\cap X, Y) + c(T\cap X, T\cap Y). \end{aligned}$$

1. $T \leftarrow$ a single special node containing every node of G;
2. while there is a special node containing more than a single node of G do
3. begin
4. Pick a special node U that contains nodes x, y;
5. Create a condensed graph G^c as follows:
6. Contract to a single node the nodes of G contained in each subtree of T attached to U;
7. For each $v \in U$, replace the parallel edges adjacent to v by one edge between v and the corresponding contracted node with the combined total capacity;
8. Find min $x - y$ cut, $\{X, Y\}$, in G^c;
9. Construct from T a new tree T' as follows:
10. Split U into two special nodes U_X and U_Y defined by $\{X, Y\}$;
11. Connect U_X and U_Y by a special edge labeled by the capacity of the min cut found above;
12. Reconnect every special node K adjacent to U in T to either U_X or U_Y depending on which side of the cut the node containing K was in G^c;
13. $T \leftarrow T'$
14. end

Figure 6: Gomory-Hu Tree Construction

Therefore $c(S \cap X, T \cup Y) \leq c(X, Y)$, and we can set $X' = S \cap X$, $Y' = T \cup Y$.

The proof of Case (ii) is similar. The cut $(T \cap X, S \cup Y)$ separates s and t, so $c(S, T) \leq c(T \cap X, S \cup Y)$, or

$$c(S \cap X, T \cap Y) + c(S \cap Y, T \cap Y) + c(S, T \cap X) \leq c(S, T \cap X) + c(T \cap X, T \cap Y)$$

and

$$c(S \cap Y, T \cap Y) \leq c(T \cap X, T \cap Y),$$

so

$$\begin{aligned} c(S \cap Y, T \cap Y) + c(S \cap Y, X) &\leq c(S \cap Y, X) + c(T \cap X, T \cap Y) \\ &\leq c(S \cap Y, X) + c(T \cap X, T \cap Y) + c(S \cap X, T \cap Y). \end{aligned}$$

Therefore $c(S \cap Y, T \cup Y) \leq c(X, Y)$, and we can set $X' = S \cap Y$, $Y' = T \cup X$.

∎

An important consequence of the above lemma is the following. Given two nodes x, y and a min $x - y$ cut $\{X, Y\}$, define the "condensed" network G^c as the network obtained from G

by contracting the nodes in Y to a single node u_Y and replacing all edges leading from a node $v \in X$ across the cut by a single edge between v and u_Y with capacity equal to the sum of capacities of the replaced edges. Then for every two ordinary nodes i, j of G^c (*i.e.*, i and $j \in X$), the min cut between i and j in G^c has the same value as the min cut between i and j in the original network. This is the idea behind the Gomory-Hu algorithm.

Construction of Gomory-Hu Trees

The algorithm appears in Figure 6. Its input is G and its output is a Gomory-Hu tree T. To distinguish between edges of the original graph and edges of the tree we call the later *special edges*. Similarly, the nodes of the tree will be referred to as *special nodes*.

Observe that in the beginning of the algorithm T contains one special node, and at each iteration the number of special nodes increases by one. Therefore the algorithm terminates in $n-1$ iterations.

Next we show that the algorithm outputs a Gomory-Hu tree. First note that at all stages of the algorithm T is a weighted spanning tree. The rest follows because as reasoned above, the min $x-y$ cut found for G^c in Line 8 of Figure 6 determines a min $x-y$ cut in G with the same capacity.

3 Multicommodity Flow

3.1 Derfinitions

In the discussion of flow problems so far we assumed that the flow concerns a single "commodity" that must be transported from a source node s to a target node t. The k-commodity flow problem is the natural generalization in which there are k commodities ($k \geq 2$) and k source-target pairs (s_i, t_i). The problem is to simultaneously transport d_i units of the ith commodity from s_i to t_i for $1 \leq i \leq k$ through the same network, *i.e.* under the constraint that the total flow of commodities 1 to k through each edge e is bounded by the capacity $u(e)$ of the edge.

More specifically, an instance of the problem consists of a graph $G = (V, E)$, a capacity function $U : E \rightarrow \mathbb{R}^+ \cup \{0\}$, and a specification of k commodities. The specification for commodity i consists of a source-sink pair $s_i, t_i \in V$ and a non-negative demand d_i. Define:

$$D_i(v) = \begin{cases} -d_i & \text{for } v = s_i, \\ d_i & \text{for } v = t_i, \\ 0 & \text{otherwise.} \end{cases}$$

A multicommodity flow f consists of a function $f_i(vw)$ on the edges of G for every commodity i, which represents the *flow* of commodity i on edge vw. Flow f is feasible if it satisfies the following system of inequalities:

$$f_i(e) \geq 0 \qquad \forall e \in E, 1 \leq i \leq k \tag{10}$$

$$\textstyle\sum_{u:(u,v)\in E} f_i(uv) - \sum_{u:(v,u)\in E} f_i(vu) = D_i(v) \qquad \forall v \in V, 1 \leq i \leq k \tag{11}$$

$$\textstyle\sum_{i:1\leq i\leq k} f_i(e) \leq u(e) \qquad \forall e \in E. \tag{12}$$

Equations (12) are usually called *conservation constraints* and equations (12) are called *capacity constraints.*

One of the natural ways to convert the feasibility question into an optimization question is to set as objective the computation of the maximum possible value z such that there is a feasible multicommodity flow with demands zd_i, $1 \leq i \leq k$. An equivalent formulation is to compute the minimum $\lambda = 1/z$ such that there is a feasible flow with demand d_i and capacities $\lambda u(e)$.

We get the following LP:

$$\text{minimize } \lambda \text{ subject to:}$$

$$\begin{cases} f_i(e) \geq 0 & \forall e \in E, \quad 1 \leq i \leq k \\ \sum_{u:(u,v)\in E} f_i(uv) - \sum_{u:(v,u)\in E} f_i(vu) = D_i(v) & \forall v \in V, \quad 1 \leq i \leq k \\ \sum_{i:1\leq i\leq k} f_i(e) \leq u(e) & \forall e \in E. \end{cases} \tag{13}$$

In the dual LP, we have one variable for each equation in the primal. In particular, for each node v and for each commodity i we have a variable $Z_i(v)$ that correspond to the conservation constraint inequality of commodity i at node v; for each edge vw we have dual variable $\ell(vw)$ that corresponds to the capacity constraint on edge vw. We have an inequality for each primal variable that is required to be non-negative ($f_i(vw)$ in our case), an an equality for each "unrestricted" variable (λ in our case). Since the capacity constraints are defined by an inequality (as opposed to equality), we have $\ell(vw) \geq 0$. Notice that minimizing λ in primal can be expressed by maximizing an objective function that has -1 in the coordinate that corresponds to λ, and zeroes everywhere else. Thus the dual has the following form:

$$\text{minimize} \sum_{1\leq i\leq k, v\in V} Z_i(v)D_i(v) \quad \text{subject to:}$$

$$(14) \qquad \begin{cases} \ell(uv) + Z_i(v) - Z_i(u) \geq 0 \quad \forall uv & \mid \ f_i(vw) \\ \displaystyle\sum_{vw\in E} \ell(vw)u(vw) = 1 & \mid \ \lambda \\ \ell(vw) \geq 0 \quad \forall vw \in E & \end{cases}$$

Observe that the objective function can be written as:

$$\sum_{1\leq i\leq k, v\in V} Z_i(v)D_i(v) = \sum_{1\leq i\leq k} (Z_i(t_i) - Z_i(s_i))d_i.$$

Observe that inequalities 14 indicate that we could think of ℓ as a (normalized) "length", which, in turn, implies that $Z_i(v)$ are some sort of "distances". We will formalize this intuition later. Denote by $\text{dist}_\ell(i)$ the length of the shortest path from s_i to t_i with respect to the length function ℓ, and denote the optimum primal and dual variables by $f^*, \lambda^*, Z^*, \ell^*$, respectively.

Theorem 31 $\lambda^* = \sum_{1\leq i\leq k} \text{dist}_{\ell^*}(i)d_i$.

Proof: For any feasible solution λ, f of the primal and feasible solution ℓ of the dual, we have:

$$(15) \qquad \sum_e \lambda u(e)\ell(e) \geq \sum_{e,i} f_i(e)\ell(e) = \sum_i \left[\sum_e f_i(e)\ell(e)\right] \geq \sum_i \text{dist}_\ell(i)d_i.$$

Moreover, $\sum_e \lambda u(e)\ell(e) = \lambda$. Hence, to complete the proof it is enough to show that for optimal λ^*, f^* and ℓ^*, each of the two inequalities in (15) is in fact an equality.

Recall that complementary slackness implies that for each positive dual optimum variable, the corresponding primal inequality is satisfied with equality by any primal optimum. Moreover, the above statement remains correct if we switch the meaning of "dual" and "primal" Applying to $\ell(e)$ we get that whenever $\ell^*(e) > 0$, we have $\sum_i f_i^*(e) = \lambda u(e)$. Hence the first inequality of (15) becomes an equality. Next we show that the second inequality of (15) becomes an equality as well.

Using complementary slackness for variables $f_i(uv)$, we get that whenever $f_i^*(uv) > 0$, we have: $Z_i^*(v) - Z_i^*(u) + \ell^*(uv) = 0$. By the flow decomposition theorem, every f_i^* can be decomposed into up to m flows $f_{i,P}^*$ where each $f_{i,P}^*$ goes from s_i to t_i along some path P, with $f_{i,P}^*$ being constant along P. Hence, we have:

$$\begin{aligned}
\sum_{vw} f_i^*(vw)\ell(vw) &= \sum_{vw}\left[\sum_P f_{i,P}^*(vw)\right]\ell(vw)\\
&= \sum_P\left[\sum_{vw} f_{i,P}^*(vw)\ell(vw)\right]\\
&= \sum_P f_{i,P}^* \sum_{vw}\ell(vw)\\
(*)\qquad &= \sum_P f_{i,P}^*(Z_i^*(s_i) - Z_i^*(t_i))\\
&= d_i(Z_i^*(s_i) - Z_i^*(t_i)),
\end{aligned}$$

where (*) is implied by the complementary slackness.

To complete the proof it is sufficient to show that $Z_i^*(s_i) - Z_i^*(t_i) \le \mathrm{dist}_{\ell^*}(i)$. Consider an arbitrary path P between s_i and t_i. Since Z^* is feasible, we have $\ell^*(vw) \ge Z_i^*(v) - Z_i^*(w)$ for all edges vw on P. Summing up all these inequalities, we get:

$$\mathrm{length}(P) = \sum_{vw\in P}\ell^*(vw) \ge Z_i^*(s_i) - Z_i^*(t_i).$$

This implies that for all i, $Z_i^*(s_i) - Z_i^*(t_i) \le \mathrm{dist}_{\ell^*}(i)$. ∎

Lecture #7&8: April 22&27, 1992
Scribe: Anil Kamath and Avi Ziv

3.2 Uniform Multicommodity Flow Problem

This results described in this lecture are based on the paper by Tom Leighton and Satish Rao [10]. The *uniform multicommodity flow problem* (UMFP) consists of a graph $G = (V, E)$ with n nodes, a capacity $c(e)$ on the edges, and a unit demand between every pair of nodes. We are going to consider the following related quantities:

$\boldsymbol{\lambda}_{\min}$ - The minimum λ such that the problem is feasible on the graph G with capacities $\lambda \cdot c(e)$.

miminum cut - The *minimum cut* in graph is defined by

$$S = \min_{U \subseteq V} \frac{\text{capacity of cut } (U, \overline{U})}{|U| \cdot |\overline{U}|}$$

flux - The *flux* of the graph is defined by

$$\alpha = \min_{U \subseteq V} \frac{\text{capacity of cut } (U, \overline{U})}{\min(|U|, |\overline{U}|)}$$

Observe that $\lambda \cdot S \geq 1$ is a necessary condition if we want the problem to be feasible when the capacities are defined by $\lambda c(e)$, which implies that $\lambda_{\min} \geq \frac{1}{S}$. Following example will show that, unlike in the case of single commodity, we can not expect to prove that $\lambda_{\min} = \frac{1}{S}$ for an arbitrary graph. Consider a UMFP on a bounded-degree expander graph with identical unit edge capacities. Since most vertices are separated by a distance (in terms of the number of edges) of $\Omega(\log n)$ the sum of the edge capacities multiplied by $\lambda_{\min}$ must be at least $\Omega(n^2 \log n)$ this means that for there to be a feasible flow $\lambda_{\min} = \Omega(n \log n)$. Since for this graph $S = \Theta(1/n)$, we have:

$$\lambda_{\min} = \Omega(\frac{\log n}{S}).$$

The main result in this lecture will be the proof of the following theorem.

Theorem 32 Any n nodes UMFP has a feasible flow with $\lambda = O(\frac{\log n}{S})$.

Instead of working with the primal, we will work with the dual. Consider the statement of the UMFP where the variables correspond to paths from source to sink of each commodity. (Notice that the number of variables in this statement of the problem is exponential.) In the dual problem we will have a non-negative variable $\ell(e)$ for each edge. Moreover, the dual constraint that corresponds to the variable λ will be:

$$\sum_{e \in E} \ell(e)c(e) = 1. \tag{16}$$

Duality implies that

$$\lambda \geq \sum_i \mathrm{dist}_\ell \text{ (source to sink of commodity } i) \tag{17}$$

Moreover, there exists a length function ℓ^*, such that

$$\lambda_{\min} = \sum_i \mathrm{dist}_{\ell^*} \text{ (source to sink of commodity } i) \tag{18}$$

The Main Theorem

Let us assume that we have succeeded in proving that $\sum_i \mathrm{dist}_\ell$ (source to sink of commodity i) $\leq \mathcal{X}$ for some $\mathcal{X}$. From 17 it follows that $\lambda_{\min} \leq \mathcal{X}$. Therefore, it is sufficient to prove the bound on the sum of distances with $\mathcal{X} = O(\frac{\log n}{S}) = O(\frac{n \log n}{\alpha})$.

Observe that it is sufficient to show that for any legal length function (where by "legal" we mean non-negative, not trivially 0, and satisfies 16), there exists some node v such that $\sum_{u \in V} \mathrm{dist}_\ell$ (v to u) $= O(\frac{\log n}{\alpha})$ because there are only $n-1$ edges in the demand graph going out of u so the total sum of the distances is $O(n)$ times the sum of the distances to v (note that in general demand graph we can replace the n with the maximum degree of the demand graph.) Hence, instead of proving Theorem 32, we will prove the following lemma.

Lemma 33 There is some node v in G for which

$$\sum_{u \in V} \mathrm{dist}_\ell \; (v \text{ to } u) = O(\frac{\log n}{\alpha})$$

The proof to the lemma will be by a constructive algorithm that will find the node v that satisfies Lemma 33, given a distance function l.

The Algorithm

We start with the preprocessing stage. In this stage we break each edge to an integral number of tokens such that each edge e is replaced with $\max\left\{1, \left\lceil \frac{\ell(e)}{\tau} \right\rceil\right\}$ tokens, where $\tau = \frac{1}{\sum_e c(e)}$. Each token of edge e is given a weight $c(e)$. The total weight of all the tokens is at most $\frac{2}{\tau}$ because

$$\text{Total weight of all tokens} \leq \sum_{e\in E} c(e)\cdot\left(1+\frac{\ell(e)}{\tau}\right) = \frac{2}{\tau}$$

We define α_1 to be initially the 1-node minimum cut i.e.

$$\alpha_1 = \min_{v\in V}\left\{\sum_{u,(u,v)\in E} c(u,v)\right\}$$

and $\alpha_0 = \alpha_1$. We also define two sets of nodes R and Q, initially empty.

In the second stage of the algorithm we grow a shallow depth region that contains most of the nodes in G. We do it in the following steps

1. Select a node $v \in V - Q$ as the center of the region R.
2. Grow region R in BFS manner by one token.
3. Update α_0 to be the capacity of the cut $(Q \cup R, \overline{Q \cup R})$, If $\alpha_0 < \frac{\alpha_1}{2}$ set $\alpha_1 = \alpha_0$ and restart again.
4. If $W_i \geq (1+\epsilon)W_{i-1}$ goto step 2, where W_i is the sum of the tokens weights in the region after step i and $\epsilon = \frac{\alpha n\tau}{17}$
5. When we can not expand anymore, if $|R| \geq \frac{n}{2}$ stop and path R to the next stage.
6. When we can not expand anymore and $|R| < \frac{n}{2}$, set $Q = Q \cup R$, goto step 1.

Step 2 is used as sanity check because we might get stuck if we start from α_1 which is much bigger than α. This check will be used again in the third stage of the algorithm.

We need to prove two things for this stage of the algorithm, first that it stops with $|R| \geq \frac{n}{2}$ and then that the BFS tree we built is shallow enough.

Claim 34 The second stage halts with $|R| \geq \frac{n}{2}$.

Proof: Suppose the claim is false, look at the first time $|Q| \geq \frac{n}{4}$. We can say that $\frac{n}{4} \leq |Q| \leq \frac{3n}{4}$ at this point. From the definition of α_0 we know that the capacity of the cut $(Q, \overline{Q}) \geq \alpha_0 \cdot \min\{|Q|, n/2\} \geq \alpha_0 n/4$. The capacity of the cut is the sum of the weights of the tokens on the cut so the total weight of tokens on the cut $\geq \alpha_0 n/4$. From step 3 we know that the ration between the internal tokens and the cut tokens in Q is at least $\frac{1}{\epsilon}$ so the token weight of the internal tokens in Q is at least $\frac{\alpha_0 n}{4\epsilon} \geq \frac{\alpha_1 n}{8\epsilon} = \frac{17}{8\tau} > \frac{2}{\tau}$, which is the total token weight. ▮

Next we show that the constructed region is "shallow".

Claim 35 The depth of the region R is $O(\frac{\log n}{\epsilon})$.

Proof: Define $c_{\min}$ as minimum capacity of a single-node cut. Since we add tokens to R only as long as $W_i \geq (1+\epsilon)W_{i-1}$ then $W_i \geq (1+\epsilon)^{i-1}c_{\min}$ for all i and especially for $i = d$, the depth of the tree. We also know that $W_i \leq \frac{2}{\tau} = 2\sum_{e\in E} c(e)$ so

$$(1+\epsilon)^{d-1}c_{\min} \leq 2\sum_{e\in E} c(e),$$

or

$$d = O\left(\frac{\log \frac{\sum_{e\in E} c(e)}{c_{\min}}}{\log(1+\epsilon)}\right)$$

We can safely assume that the capacity of each edge is at most $nc_{\min}$ because larger capacities can not participate in the "congested cut", so $\sum_{e\in E} c(e) \leq n^3 c_{\min}$ and $\log \frac{\sum_{e\in E} c(e)}{c_{\min}} = O(\log n)$.

We also know that $\epsilon = \frac{\alpha_1 n}{17\sum_{e\in E} c(e)}$ and

$$\alpha_1 \leq \frac{1}{n}\sum_{v\in V} \text{capacity of cut } (\{v\}, V - \{v\}) = \frac{2}{n}\sum_{e\in E} c(e)$$

(the minimum is not greater than the average). So $\epsilon \leq \frac{2}{17} < 1$ and we can say that $\log(1+\epsilon) = \Omega(\epsilon)$ and $d = O(\frac{\log n}{\epsilon})$ ▮

So now we have a region of nodes R which contains at least half of the nodes in the graph, and for which the sum of distances from the root, v, to all the nodes in the region is $O(\frac{\tau n \log n}{\epsilon}) = O(\frac{\log n}{\alpha})$. We still need to extend the region to all the other nodes in the graph, and to show that the extension maintain the small total path length.

In the third stage of the algorithm we continue to extend the region R, constructed in stage two, in BFS manner, one token at a time, until all the nodes are reached. As in stage two here we also update α_0 after each step and restart the algorithm all over if α_0 falls below $\frac{\alpha_1}{2}$.

Claim 36 The total length added in the third stage is $O(\frac{1}{\alpha})$.

Proof: Define n_i to be the number of nodes outside R after step i of the third stage. As the flux of the cut at step i is at least α, or the cut capacity is at least αn_i and the total weight of the tokens is at most $\frac{2}{\tau}$ the following holds

$$\sum_{i=1}^{\infty} n_i\alpha \leq \sum_{i=1}^{\infty} \text{token weights on cut at step } i \leq \frac{2}{\tau}$$

or

$$\tau \sum_{i=1}^{\infty} n_i = O(\frac{1}{\alpha})$$

∎

This conclude the proof of Lemma 33 and hence Theorem 32.

Note that we always maintain that α_0 is within a constant factor of α_1 because of the "sanity check" we perform on each new cut that we encounter.

Remarks

1. Similar ideas led to an algorithm that can be used for a general demand graph. This algorithm, due to Klein, Agrawal, Ravi, and Rao, provides the following bound:

$$\lambda_{\min} \leq \frac{\log n \log \sum \text{demands}}{\tau}$$

   For some special case, like when the demand graph is a constant-degree expander the bound of theorem 32 holds.

2. The algorithm can also be used to show that a bounded degree graph H can be embedded into graph G with congestion and dilation of $O(\frac{\log n}{\alpha})$. In our proof we only showed congestion of $O(\frac{n \log n}{\alpha})$.

Lecture #9: April 29, 1992
Scribe: W. Bernard Lee

3.3 Relation Between Multicommodity Flow and Minimum-Cut

In the previous lecture we have defined the *minimum cut* of the graph by:

$$S_{\min} = \min_{U \subseteq V} \frac{\text{capacity of cut } (U, \overline{U})}{|U| \cdot |\overline{U}|}.$$

We have considered the *uniform multicommodity flow problem* (UMFP), which consists of a graph $G = (V, E)$ with n nodes, a capacity $u(e)$ on the edges, and a unit demand between every pair of nodes. The main result was a proof that

$$\lambda^* = O\left(\frac{\log n}{S_{\min}}\right)$$

where λ^* is the minimum λ such that there exists a feasible solution to the UMFP in the G with capacities $\lambda \cdot u(e)$.

We have actually proved more than just the above claim. Given a *length function* $\ell : E \to \mathbb{R}^+ \cup \{0\}$ such that $\sum_{e \in E} \ell(e)u(e) = 1$, we have shown how to find a cut such that the value of S_{obs} observed for this cut satisfies:

$$\sum_{u,v \in V} \text{dist}_\ell \ (v \text{ to } u) = O\left(\frac{\log n}{S_{\text{obs}}}\right).$$

In other words,

$$S_{\text{obs}} = O\left(\frac{\log n}{\sum_{u,v \in V} \text{dist}_\ell \ (v \text{ to } u)}\right) \tag{19}$$

Therefore, if we want to find a cut whose corresponding S is as small as possible, we should:

$$\begin{array}{lrcl} \text{maximize} & \sum_{u,v \in V} \text{dist}_\ell \ (v \text{ to } u) & & \\ \text{subject to:} & \ell(e) & \geq & 0 \quad \forall e \in E \\ & \sum_{e \in E} u(e)\ell(e) & = & 1 \end{array}$$

Let ℓ^* be the solution of this LP. Since this LP is the dual of the UMFP LP that is defined as a problem to find minimum λ such that capacities $\lambda u(e)$ are sufficient for feasibility, we have:

$$\lambda^* = \sum_{u,v \in V} \text{dist}_{\ell^*}\ (v \text{ to } u)$$

By definition of $S_{\min}$ and λ^*, we have $S_{\min} \geq 1/\lambda^*$. Substituting into (19), we see that if we can find ℓ^*, we can find a cut such that the corresponding S_{obs} will satisfy:

$$\frac{1}{\lambda^*} \leq S_{\min} \leq S_{\text{obs}} = O\left(\frac{\log n}{\lambda^*}\right)$$

In other words, we see that by solving the dual of the UMFP, we can find a cut whose corresponding S is within $O(\log n)$ of the minimum one possible.

3.4 Historical Notes

Since we have seen that a solution to the multicommodity flow problem can be used to approximate the minimum cut, a natural question to consider is how to solve the multicommodity flow problem efficiently. MFP can be represented as LP, and hence can be solved in polynomial time using one of the interior-point methods. Unfortunately, these methods, when applied to the MFP LP, are quite slow. The best theoretical bound is due to Vaidya, who showed that the problem can be solved in $O(k^{3.5}n^3m^{.5}\log(nDU))$ time, where k is the number of commodities, n is the number of nodes, m is the number of edges, D is the largest demand, and U is the largest capacity. (Both demands and capacities are assumed to be integer. The first strongly polynomial algorithm for the multicommodity flow problem is due to Tardos [17]. The first strongly polynomial algorithm for the variant of the multicommodity flow where one minimizes λ is due to Norton, Plotkin, and Tardos [6]. Both algorithms are extremely slow and not practical.)

We will present a much faster *approximation* algorithm to the multicommodity flow problem. In particular, if our goal is to approximate $S_{\min}$, having an exact solution does not really help. The algorithm, as we will present it, will provide an *approximate primal* solution, *i.e.* flow values on all the edges. Converting this solution to an *approximate dual* solution is non trivial, and appears in [9].

The first polynomial time algorithm for the uniform capacity uniform demand multicommodity flow problem that is not based on general interior point LP methods, is due to Shahrokhi and Matula [15]. Their algorithm is extremely slow, even slower than the interior-point approach, and can not be applied to solve multicommodity problems with general demands. Klein, Plotkin, Stein, and Tardos [8] developed an efficient algorithm for the uniform capacity case with general demands. Their algorithm runs in expected $O((k+m)(m+n\log n)\log n)$ time. In [9], Leighton et. al. developed an algorithm that solves multicommodity flow with arbitratry demends and capacities, but with no costs. The running time of their algorithm, ignoring polylogarithmic factors, is $O^*(kmn)$, *i.e.* approximately k minimum-cost flows. In [13], Plotkin, Shmoys, and Tardos extended the approach to apply to any fractional packing/covering problem. In particular, one of the applications is the minimum-cost version of the multicommodity flow. (Other applications include scheduling, approximate bin packing, etc.)

In the following presentation, we will follow [13], and specialize it to the case of multicommodity flow with no costs.

3.5 Multicommodity Flow Algorithm

Main Ideas Behind the Algorithm

An instance of the *simple multicommodity flow problem* consists of an undirected graph $G = (V, E)$, a non-negative capacity $u(vw)$ for every edge $vw \in E$, and a specification of k commodities, numbered 1 through k, where the specification for commodity i consists of a source-sink pair $s_i, t_i \in V$ and a non-negative demand d_i. We will denote the number of different sources by k, the number of nodes by n, and the number of edges by m. For notational convenience we assume that $m \geq n$, and that the graph G is connected and has no parallel edges. Also, for notational convenience, we arbitrarily direct each edge. If there is an edge directed from v to w, this edge is unique by assumption, and we denote it by vw. We assume that the capacities and the demands are integral, and denote the largest capacity by U and the largest demand by D.

A multicommodity flow f consists of a function $f_i(vw)$ on the edges of G for every commodity i, which represents the *flow* of commodity i on edge vw. If the flow of commodity i on edge vw is oriented in the same direction as edge vw, then $f_i(vw)$ will be positive, otherwise it will be negative. The signs only serve to indicate the direction of the flows. For every commodity i we require the *conservation constraints*:

$$(20) \qquad \sum_{wv \in E} f_i(wv) - \sum_{vw \in E} f_i(vw) = 0 \quad \text{for every node } v \notin \{s_i, t_i\}.$$

We require also that $\sum_{vw \in E} f_i(vw) = d_i$ for $v = s_i$. We define the value of the total flow on edge vw to be $f(vw) = \sum_i |f_i(vw)|$, and say that a multicommodity flow f in G is *feasible* if $f(vw) \leq u(vw)$ for all edges vw. (Note that $f(vw)$ is always non-negative.)

Let $\ell : E \to \mathbb{R}^+ \cup \{0\}$ be the *length* function on edges, f a multicommodity flow, and $\lambda = \max_{vw \in E} \lambda(vw)$, where $\lambda(vw) = f(vw)/u(vw)$. Let C_i be the cost of the current flow for commodity i, using ℓ as the cost function. For a commodity i, let $C_i^*(\lambda)$ be the value of a minimum-cost flow f_i^* satisfying the demands of commodity i, subject to costs ℓ and capacities $\lambda \cdot u(vw)$, *i.e.* let f_i^* be a flow that satisfies $|f_i^*(vw)| \leq \lambda \cdot u(vw)$ and minimizes the cost $C_i^*(\lambda) = \sum_{vw} |f_i^*(vw)| \ell(vw)$. For brevity we shall sometimes use C_i^* to abbreviate $C_i^*(\lambda)$.

Theorem 37 For a multicommodity flow f satisfying capacities $\lambda \cdot u(vw)$, and a length function ℓ,

$$\lambda \sum_{vw \in E} \ell(vw)u(vw) \geq \sum_{i=1}^{k} \sum_{vw \in E} |f_i(vw)|\ell(vw) = \sum_{i=1}^{k} C_i \geq \sum_{i=1}^{k} C_i^*(\lambda).$$

A multicommodity flow f minimizes λ if and only if there exists a nonzero length function ℓ for which all the above terms are equal.

We would like to be able to say that the ratio of the last term and the multiplier of λ in the first term gives a lower bound on the optimal value λ^*. Observe, that the minimum cost of a flow subject to capacity constraints $\lambda \cdot u(vw)$ cannot increase if λ increases.

Lemma 38 Suppose that there exists a multicommodity flow satisfying capacities $\lambda \cdot u(vw)$. Then for any length function ℓ the value $\sum_{i=1}^{k} C_i^*(\lambda)/(\sum_{vw \in E} \ell(vw)u(vw))$ is a lower bound on λ^*.

The goal of the algorithm is to find a multicommodity flow f and a length function ℓ such that this lower bound is within a $(1+\epsilon)$ factor of optimal, in other words:

$$\lambda \leq (1+\epsilon)\frac{\sum_{i=1}^{k} C_i^*(\lambda)}{\sum_{vw \in E} \ell(vw)u(vw)}.$$

In this case, we say that f and ℓ are *ϵ-optimal.* Note that we are using ϵ-optimal to refer both to a flow and a flow and length function pair. Recall, that a flow is ϵ-optimal if it has congestion λ at most $(1+\epsilon)$ times the minimum possible congestion. If f and ℓ are ϵ-optimal then Lemma 38 implies that f is ϵ-optimal. Notice that in order to recognize that a flow is ϵ-optimal, we will need to have the "right" length function.

The complementary slackness conditions can be reformulated in terms of conditions on edges and individual commodities. A multicommodity flow f has minimum λ if and only if there exists a nonnegative and non-zero length function ℓ such that:

1. For every edge $vw \in E$, either $\ell(vw) = 0$ or $f(vw) = \lambda \cdot u(vw)$,
2. For every commodity i, $C_i = C_i^*(\lambda)$.

These two conditions characterize when f and ℓ are *optimal*; we shall give two conditions on a multicommodity flow f and a length function ℓ such that together they imply that f and ℓ are *ϵ-optimal.* These conditions will be relaxed versions of the complementary slackness conditions above.

Let $\epsilon > 0$ be an error parameter, f a multicommodity flow satisfying capacities $\lambda \cdot u(vw)$, and ℓ a length function.

(R1) $(1-\epsilon)\lambda \sum_{e \in E} \ell(e)u(e) \leq \sum_1^k C_i$.

(R2) $\sum_1^k C_i - \sum_1^k C_i^*(\lambda) \leq \epsilon \left(\sum_1^k C_i + \lambda \sum_1^k u(e)\ell(e)\right)$.

The following Theorem can be easily proved directly from the definition of $(R1)$ and $(R2)$:

Theorem 39 Suppose f, ℓ, and ϵ satisfy the relaxed optimality conditions and $\epsilon < 1/9$. Then f is $O(\epsilon)$-optimal, *i.e.* λ is at most a factor $(1 + O(\epsilon))$ more than the minimum possible value.

Proof: (sketch):
Rewrite (R1):

$$
\begin{aligned}
\lambda \sum u(e)\ell(e) &\leq \frac{1}{(1-\epsilon)} \sum C_i^* \\
&\leq \frac{1}{(1-\epsilon)} \left[\frac{1}{(1-\epsilon)} \left[\sum C_i^* + \epsilon\lambda \sum \ell(e)u(e) \right] \right] \\
\left[1 - \frac{\epsilon}{(1-\epsilon)^2} \right] \lambda \sum u(e)\ell(e) &\leq \frac{1}{(1-\epsilon)^2} \sum C_i^*(\lambda) \\
\sim \quad (1-\epsilon)\lambda \sum u(e)\ell(e) &\leq (1-2\epsilon) \sum C_i^*(\lambda) \\
\sim \quad \lambda \sum u(e)\ell(e) &\leq (1+\epsilon') \sum C_i^*(\lambda)
\end{aligned}
$$

Therefore, using Lemma 38:

$$
\begin{aligned}
\lambda &\leq (1+\epsilon') \frac{\sum C_i^*(\lambda)}{\sum u(e)\ell(e)} \\
\Rightarrow \quad \lambda &\leq (1+\epsilon')\lambda^*
\end{aligned}
$$

∎

Our approach will be to start from some (bad) flow, that satisfies the demands, define ℓ based on this flow in such a way that $R1$ is satisfied, and check $R2$. If $R2$ is satisfied, we stop and claim that the current solution is close to optimum, based on Theorem 39. If not, we will reroute a portion of flow, and claim that some progress was made.

The length function is defined by:

$$
\begin{aligned}
\lambda(vw) &= \frac{f(vw)}{u(vw)} \\
\ell(vw) &= \frac{1}{u(vw)} e^{\frac{\alpha f(vw)}{u(vw)}}.
\end{aligned}
$$

We use the potential function:

$$\Phi = \sum u(e)\ell(e)$$

The following lemma is left as an exercise:

Lemma 40 If f is a multicommodity flow which satisfies demands and $\alpha \geq (1+\epsilon)\lambda^{-1}\epsilon^{-1}\ln(m\epsilon^{-1})$ then f and length function $\ell(vw) = e^{\alpha\lambda(vw)}/u(vw)$ satisfy Relaxed Optimality Condition $R1$.

Notice that Φ and λ are not directly related because Φ can decrease while λ increases. However, observe that If $\frac{\lambda_0}{2} \leq \lambda \leq \lambda_0$, then:

$$e^{\frac{\alpha\lambda_0}{2}} \leq \Phi \leq me^{\alpha\lambda_0}$$

We will show how to reroute a fraction of the flow so that the $\hat{\Phi}$ after the rerouting will satisfy:

$$\hat{\Phi} \leq \Phi(1-\beta)$$

A natural approach to reduce Φ is to move in the direction of the steepest descent, *i.e.* define (for some small σ):

$$\hat{f} = (1-\sigma)f + \sigma(-\nabla\Phi)$$

Unfortunately, $\hat{f}$ defined this way will not necessary satisfy conservation constraints. Instead, we use $\nabla\Phi = \alpha\ell$ as the cost function, and compute minimum-cost flow f_i^* of each commodity with respect to this cost and capacities $\lambda u(e)$.

We take

$$\sigma \approx \frac{\epsilon}{8\alpha k\lambda}$$

and define:

$$\begin{aligned} \hat{f} &= (1-\sigma)f + \sigma f^* \\ \Delta\hat{f} &= \hat{f} - f = \sigma\,[f^* - f] \end{aligned}$$

Lemma 41 Let $\frac{\epsilon}{16\alpha k\lambda} \leq \sigma \leq \frac{\epsilon}{8\alpha k\lambda}$. Let the new flow for commodity i be defined by $f_i(vw) \leftarrow (1-\sigma)f_i(vw) + \sigma f_i^*(vw)$. Then $\Phi - \hat{\Phi} = \Omega(\frac{\epsilon^2}{k}\Phi)$, where $\hat{\Phi}$ is the value of the potential function associated with the new flow.

Proof:

$$\hat{\ell}(e) = \frac{1}{u(e)}\exp\left\{\frac{\alpha}{u(e)}f(e) + \frac{\alpha}{u(e)}\sigma[f^*(e) - f(e)]\right\}$$

Observe that our choice of σ implies:

$$\frac{\alpha}{u(e)}\sigma|f^*(e) - f(e)| \leq 2\alpha\sigma\lambda k \leq \frac{\epsilon}{4} \leq \frac{1}{4}$$

Recall the Taylor series expansion:

$$e^{x+\delta} \leq e^x + \delta e^x + \frac{\epsilon}{2}|\delta| e^x \quad \textit{where} \quad |\delta| \leq \frac{\epsilon}{4} \leq \frac{1}{4}$$

Applying the Taylor expansion and using the fact that $R2$ is not satisfied, we get:

$$\begin{aligned} \Phi - \hat{\Phi} &\geq \alpha\sigma \sum [f(e) - f^*(e)] - \frac{\epsilon}{2}\alpha\sigma \sum |f(e) + f^*(e)| \cdot \ell(e) \\ &\geq \alpha\sigma \sum (C_i - C_i^*) - \epsilon\alpha\sigma \sum C_i \\ &\geq \alpha\sigma\epsilon\Phi \end{aligned}$$

The claim follows by substituting the chosen values of σ and α. ∎

Lecture #10: May 11, 1992
Scribe: Robert Kennedy

The Algorithm

As suggested by the above discussion and the previous lecture, the algorithm works as follows:

- Compute starting flow f and $\lambda(f)$.
- While condition R2 is not satisfied,
  - Compute k min-cost flows f_i^* with respect to capacities λu and costs l.
  - Update the multicommodity flow: $f_i \leftarrow (1-\sigma)f_i + \sigma f_i^*$, where $\sigma = \epsilon/8\alpha\lambda k$.

In the previous lecture, we saw that when condition R2 does not hold, updating the flow according to this method reduces Φ by a factor of at least $(1-\epsilon^2/k)$. Included in that argument was the motivation for our choice of σ; we take as large a step as possible in the direction of the min-cost flow while maintaining that the improvement in Φ due to our linear approximation dominates the effects of higher-order terms.

Simple calculus provides then that $O(k\alpha\lambda/\epsilon^2)$ iterations suffice to reduce λ by a constant factor, and for ϵ polynomially small or larger in n, $k\alpha\lambda/\epsilon^2 = O(\epsilon^{-3}k\log n)$, when α is defined by (40). Observe also that once $\lambda < 2\lambda^*$, we are guaranteed that the algorithm will terminate with an ϵ-optimal multicommodity flow within at most $O(\epsilon^{-3}k\log n)$ additional iterations because so many iterations are guaranteed to reduce λ by a factor of 2 unless both relaxed optimality conditions are satisfied.

Getting Started

Now to specify the algorithm completely, we must say how we find a starting flow f^0. We use a maximum flow subroutine to find a flow for commodity i that obeys the capacity constraints $\lambda_i u$ and that satisfies the demand of commodity i, for λ_i minimum. Let g_i be a a maximum flow from s_i to t_i obeying capacities u,, and denote the value of this flow by $|g_i|$. It is straightforward to check that $f_i(e) = d_i g_i(e)/|g_i|$ satisfies capacity constraints $\lambda_i u(e)$, where $\lambda_i = d_i/|g_i|$ is the minimum possible. Now it is clear that $\forall i: \quad \lambda_i \leq \lambda^*$. Furthermore, we know that $\lambda^* \leq \sum_{i=1}^k \lambda_i$, because our starting flow is feasible. Thus the initial flow f^0 that we found satisfies capacity constraints $\lambda_0 u(e)$, where $\lambda_0 \leq k\lambda^*$.

In the case where each commodity may have more than one sink and more than one source, we must use a slightly more complicated idea to find an initial feasible primal solution. In this case, instead of talking about sources and sinks, it is more convenient to consider for each node/commodity pair, the amount of the commodity that is consumed by this node. Denote the amount of commodity i consumed by node v by $D_i(v)$. In this notation, the single source single sink case can be expressed by setting $D_i(s_i) = -d_i$; $D_i(t_i) = d_i$ and $D_i(v) = 0$ otherwise. To find the initial flow for commodity i, we introduce a new source and a new sink, connecting the new source to each node $\{v : D_i(v) < 0\}$ by an edge e with $u(e) = -qD_i(v)$, where q is a parameter to be determined. Similarly, we connect each node $\{v : D_i(v) > 0\}$ by an edge e with $u(e) = qD_i(v)$ to the new sink.

Assume that we have chosen q such that the new problem has a solution in which all the added edges are saturated. In this case we know that $\lambda_i \leq 1/q$. On the other hand, it is easy to see that for $q = 1/\lambda_i$, there exists a (maximum) flow that saturates all the new edges, and for $q > 1/\lambda_i$ such a flow does not exist. Therefore we can perform binary search on q. The range of the search corresponds to the range of possible values for λ_i. Let U denote the maximum capacity, and let $D_i = \max_v\{D_i(v)\}$ Then, since we have assumed integer capacities, $\lambda_i \leq D_i$ and $\lambda_i \geq 1/nU$. Hence the binary search entails $O(\log nU)$ iterations and provides us with an initial multicommodity flow and λ value with the property that $\lambda \leq 2k\lambda^*$.

ϵ-Scaling

We can improve the runnig time by "scaling" on ϵ. The idea is that a 2ϵ-optimal multicommodity flow is a good initial feasible primal solution for the computation of an ϵ-optimal multicommodity flow. In particular, if our current (f, λ) pair is 2ϵ-optimal but not ϵ-optimal, we know that

$$e^{\alpha(1+\epsilon)\lambda^*} \leq \Phi \leq me^{\alpha(1+2\epsilon)\lambda^*}.$$

Hence the number of iterations required to convert a 2ϵ-optimal flow to an ϵ-optimal flow is $O(\epsilon^{-2}k\log n)$, for polynomially small ϵ. Because this bound grows as the inverse square of ϵ and ϵ is halved at every stage, the final scaling phase dominates the number of iterations for all scaling phases. By the analysis above, with constant ϵ, $O(k\log n\log k)$ iterations are required to get the initial solution for the first scaling phase after the max flow computations, so the overall number of iterations with scaling is $O(\epsilon^{-2}k\log n + k\log n\log k)$.

Single-Commodity Updates

The lecture presented a sketch of a slightly different flow-updating scheme: Whereas before, we had updated at each iteration by computing k minimum-cost flows, one per commodity, we can update by computing a single minimum-cost flow at each iteration, and provided we reroute a "bad" commodity often enough, we can achieve similar runtimes to the previous technique. Intuitively, a "bad" commodity is one that contributes more than its share to the violation of condition R2. Recall from the previous lecture that to bound the error due to higher-order terms (and hence the number of iterations), we used the condition that

$$\frac{\alpha\sigma}{u(e)}\left|f(e) - \hat{f}(e)\right| \leq \frac{\epsilon}{4},$$

where $\hat{f}$ is the updated value of f. This relation constrained our choice of σ, the step size, and now because only one commodity's flow is changing per iteration, we may take larger steps and achieve the same bounds; we take σ a factor of k larger than before. There is a variety of ways we might find a bad commodity without computing a minimum-cost flow for each commodity; one way is simply to choose a commodity uniformly at random. The expected improvement is large, and using techniques for analyzing probabilistic recurrences due to Karp (not covered in lecture), we can show that with high probability, the asymptotic number of iterations is the same as in the previous scheme.

Precision Concerns in Min-Cost Flow

One point remains in determining the run time of the algorithm: Which minimum-cost flow routine should we use? The length function, as it is exponential, would seem to make standard scaling algorithms inappropriate, but it turns out we can use an approximation to the length function and still obtain essentially the same analysis. The main idea is to observe that instead of computing the minimum cost flow at each iteration, it is sufficient to compute a flow which cost is within $\frac{\epsilon}{2k}\lambda\Phi + \frac{\epsilon}{2}C_i$ additive error of the optimal.

Lecture #11: May 13, 1992
Scribe: Ray-Hon Sun

4 The Ellipsoid Method

Review

In 1979 the Soviet mathematician L.G. Khachian [7] developed the so-called **ellipsoid method**, originally devised for nonlinear nondifferentiable optimization [11, 16, 19]. Since the ellipsoid method was the first polynomial-time algorithm for linear programming, it received a great deal of attention not only in the scientific literature, but also in the nonscientific press. The ellipsoid method even appeared in the New York Times: *"An Approach to Difficult Problems,"* (Nov. 27, 1979) - see [12].

Based on work of mathematicians such as Levin [11], Shor [16], and Yudin et. al. [19] in the USSR, Yamnitsky and Levin, inspired by Khachian's ellipsoid method, published an algorithm [18], in which simplices are used instead of ellipsoids. This algorithm is somewhat slower than the ellipsoid method, but it is simpler and seems to have the same theoretical applicability.

From the previous lectures, we know that an LP: $\max c'x$, subject to $Ax \leq b$ and $x \geq 0$, is polynomial-time equivalent to *linear inequalities* (LI): find a feasible solution of $Ax \leq b$ (See Papadimitriou et. al. [12], pp.172 Theorem 8.2 for detail.)

Next we introduce the *strict linear inequalities* (SLI). We want to see whether the seemingly restrictive problem SLI captures the complexity of LI. By **Lemma 8.7** of Papadimitriou et. al. [12], the system of linear inequalities (LI): $a_i x_i \leq b_i$, $1 \leq i \leq m$, has a solution *iff* the system of strict linear inequalities (SLI): $a_i x_i < b_i + \epsilon$, $1 \leq i \leq m$ has a solution, where $\epsilon = (nB)^{-(n+1)}$ (here B is the largest absolute value of an input number). (See Papadimitriou et. al. [12], pp.174 Corollary for detail).

The algorithm of Yamnitsky and Levin [1982] solves SLI. Our presentation of this algorithm follows Chvátal's book [2].

The Yamnitsky-Levin (Modified Ellipsoid) Algorithm for SLI

The algorithm maintains a simplex that contains the feasible polytope. At each iteration the algorithm eitehr finds a feasible solution and terminates or replaces the current simplex with one of a smaller volume. If the volume of the simplex becomes too small, the algorithm concludes that the problems is infeasible.

Step 1. $S \leftarrow$ enclosing simplex.

Step 2. If *center(S)* is feasible
then return
else
consider the corresponding half-simplex $S_{\frac{1}{2}}$
find a smaller simplex $\tilde{S}$ containing $S_{\frac{1}{2}}$
$S \leftarrow \tilde{S}$

Step 3. If *vol(S)* is *"small"* then return (*"infeasible"*)
else
go to **Step 2.**

Definitions

To explain the details, a few definitions are required.

Definition 42 A set $S \subset \Re^n$ is a *convex hull* if S is the intersection of all convex sets containing S.

Definition 43 Points $v_0, v_1, ..., v_n$ are said to be in *general position* if

- the vectors $v_1 - v_0, ..., v_n - v_0$ are linearly independent,
- dim $\mathrm{V}(v_0, v_1, ..., v_n) = n$,
- no hyperplane passes through all of them.

Definition 44 An *n-simplex* is the convex hull of $n+1$ points in general position.

Definition 45 A set of vectors is a (convex) *polytope* if it is the convex hull of finitely many vectors.

Definition 46 Suppose S is an n-simplex in $\Re^n$, then its center is defined by

$$center(S) = \frac{\sum_{i=0}^{n} v_i}{n+1}$$

and its volume by

$$vol(S) = \frac{\det(A)}{n!} \quad \text{where } A = \begin{pmatrix} v_1 - v_0 \\ v_2 - v_0 \\ \vdots \\ v_n - v_0 \end{pmatrix}.$$

Definition 47 A *half-simplex* $S_{\frac{1}{2}}$ is the intersection of a simplex S with half-space whose bounding hyperplane passes through the center of S.

Exercise: How can we find the initial simplex of volume $n^{n^{O(1)}}B^{n^{O(1)}}$ (here B is the largest integer in the input)?

Exercise: Show that if an enclosing simplex has volume $n^{-n^{O(1)}}B^{-n^{O(1)}}$, then the feasible polytope is empty.

Key Lemma

Lemma 48 Let $S_{\frac{1}{2}}$ be a half-simplex of S, then $\exists$ simplex $\tilde{S} \supseteq S_{\frac{1}{2}}$ s.t. $r = \frac{vol(\tilde{S})}{vol(S)} < e^{-\frac{1}{2n^2}}$.

Proof: Let S be the convex hull of points $v_0, v_1, ...v_n$ and let $S_{\frac{1}{2}}$ be the intersection of S with half-space $a'x \le b$. Define $e(x) = b - a'x$ for all points x. Let k be the subscript that maximizes $e(v_k)$. Since $S_{\frac{1}{2}} \neq \emptyset$, we know $e(v_k) > 0$. Define points $\tilde{v_0}, \tilde{v_1}, ..., \tilde{v_n}$ by

$$\tilde{v_i} = v_k + \frac{v_i - v_k}{d_i}, \quad \text{with } d_i = 1 - \frac{e(v_i)}{n^2 e(v_k)}.$$

We show that the convex hull $\tilde{S}$ of $\tilde{v_0}, \tilde{v_1}, ..., \tilde{v_n}$ has the desired properties.

First we show that $\tilde{S} \supseteq S_{\frac{1}{2}}$. Pick $x \in S_{\frac{1}{2}}$. Since $x \in S$, $x = \sum_{i=0}^n t_i v_i$ for some t_i such that $\sum_{i=0}^n t_i = 1$ and $t_i \ge 0$. Define

$$\tilde{t_i} = \begin{cases} d_i t_i, & i \neq k, \\ d_k t_k + \frac{e(x)}{n^2 e(v_k)}, & i = k. \end{cases}$$

Since $e(x) = \sum_{i=0}^n t_i e(v_i)$,

$$\begin{aligned} \sum_{i=0}^n \tilde{t_i} &= \sum_{i=0}^n d_i t_i + \frac{e(x)}{n^2 e(v_k)} \\ &= \sum_{i=0}^n t_i - \frac{\sum_{i=0}^n t_i e(v_i)}{n^2 e(v_k)} + \frac{e(x)}{n^2 e(v_k)} \\ &= \sum_{i=0}^n t_i \\ &= 1. \end{aligned}$$

Since $e(v_i) \le e(v_k)$, we have $0 \le \tilde{t_i} \le 1$.

Next we show that $\sum_{i=0}^{n} \tilde{t}_i \tilde{v}_i = x$.

$$\begin{aligned}
\sum_{i=0}^{n} \tilde{t}_i \tilde{v}_i &= \sum_{i=0}^{n} d_i t_i (v_k + \frac{v_i - v_k}{d_i}) + \frac{e(x)}{n^2 e(v_k)} v_k \\
&= \underbrace{\sum_{i=0}^{n} v_i t_i}_{x} + \underbrace{v_k \frac{\sum_{i=0}^{n} t_i d_i}{n^2 e(v_k)} + \frac{e(x)}{n^2 e(v_k)} v_k}_{v_k} - v_k \underbrace{\sum_{i=0}^{n} t_i}_{1} \\
&= x.
\end{aligned}$$

Thus $x \in \tilde{S}$.

Next we show that the ratio r between the volume of $\tilde{S}$ and the volume S is less than $e^{-\frac{1}{2n^2}}$. Note that $\tilde{v_k} = v_k$, that each $\tilde{v_i}$ with $i \neq k$ lies on the line passing through v_k and v_i, and that the distance from v_k to $\tilde{v_i}$ equals the distance From v_k to v_i divided by d_i. Therefore, $r = \prod_{i \neq k} \frac{1}{d_i}$. Since $e(v_i) \leq e(v_k)$, we have $d_i \geq 1 - \frac{1}{n^2}$. For $c = center(S)$, $e(c) = 0$. We have

$$\begin{aligned}
\sum_{i \neq k} d_i &= \sum_{i \neq k} (1 - \frac{e(v_i)}{n^2 e(v_k)}) \\
&= n - \frac{1}{n^2 e(v_k)} \sum_{i \neq k} e(v_i) \\
&= n - \underbrace{\frac{e(c)}{ne(v_k)}}_{0} + \underbrace{\frac{e(v_k)}{n^2 e(v_k)}}_{\frac{1}{n^2}} \\
&= n + \frac{1}{n^2}.
\end{aligned}$$

We want to minimize a product of variables subject to (a) a positive lower bound on each variable, and (b) the sum of the variable equal to a prescribed number. It is easy to show that at most one variable exceeds it lower bound in every optimal solution. Therefore

$$\begin{aligned}
\prod_{i \neq k} d_i &\geq (1 - \frac{1}{n^2})^{n-1} (1 + \frac{1}{n}) \\
&= (1 - \frac{1}{n^2})^n \frac{1 + \frac{1}{n}}{1 - \frac{1}{n^2}} \\
&= (1 - \frac{1}{n^2})^n (1 - \frac{1}{n})^{-1}
\end{aligned}$$

Taking logarithm, we get

$$\begin{aligned}
\lg \left((1 - \frac{1}{n^2})^n (1 - \frac{1}{n})^{-1} \right) &= n \lg(1 - \frac{1}{n^2}) - \lg(1 - \frac{1}{n}) \\
&= -n \sum_{i=1}^{\infty} \frac{1}{i \cdot n^{2i}} + \sum_{i=1}^{\infty} \frac{1}{i \cdot n^i} \\
&> \frac{1}{2n^2}.
\end{aligned}$$

Therefore $\prod_{i \neq k} d_i > e^{\frac{1}{2n^2}}$, or

$$r = \prod_{i \neq k} \frac{1}{d_i} < e^{-\frac{1}{2n^2}}.$$

∎

Example

The following example illustrates how the algorithm works.

$x \leq 2, \quad e(x,y) = 2 - x, \quad k = 2, \quad v_k = (0,6),$
$\tilde{v_0} = (0,6) + \frac{1}{1-\frac{2}{2^2 \cdot 2}}(0,6) = (0,-2),$
$\tilde{v_1} = (0,6) + \frac{1}{1-\frac{-4}{2^2 \cdot 2}}(0,6) = (4,2),$
$\tilde{v_2} = (0,6).$

Lecture #12: May 18, 1992
Scribe: Tomek Radzik

5 Reduction of the General LP to the Multicommodity Flow Problem

This lecture was given by Prof. Cherkassky. He showed a polynomial reduction from the general linear program to the multicommodity flow problem. This reduction was devised by E. A. Dinitz in early seventies, before the first polynomial algorithm for the general linear programming problem was found.

Linear Program

We work with the following version of the linear programming problem.

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j = b_i, & \text{for} \quad i = 1, 2, \ldots, m \\ & x_j \geq 0, & \text{for} \quad j = 1, 2, \ldots, n \end{array}$$

All b_i's and a_{ij}'s in this program are integers. Our construction gives an instance of the multicommodity flow problem of size polynomial in n, $\max_i\{\log b_i\}$, and $\max_{i,j}\{\log a_{ij}\}$.

For our construction we need M such that if there is a solution, there is one with

$$|a_{ij}| x_j < M, \quad \text{for} \quad j = 1, 2, \ldots, n \quad \text{and} \quad i = 1, 2, \ldots, m.$$

For example $M = n^n \max_i\{b_i\}(\max_{i,j}\{a_{ij}\})^n$ will do.

Multicommodity Flow Problem

We reduce LP to the directed multicommodity flow problem, *i.e.*, the underlying networks are directed. The objective function is a linear combination of the values of the flows of the commodities. An instance of the problem consists of a directed graph $G = (V, E)$, capacities of arcs

$(b_u)_{u\in E}$, a list $(s_k, t_k)_{1\leq k\leq l}$ specifying the source and the destination for each of l commodities, and a vector $(\lambda_k)_{1\leq k\leq l}$ specifying the objective function. The task is to

$$\text{maximize} \quad \sum_{k=1}^{l} \lambda_k f_k, \qquad \text{subject to}$$

$$\sum_{tail(u)=i} x_u^k - \sum_{head(u)=i} x_u^k = \left\{ \begin{array}{ll} f_k & \text{if } i = s_k \\ -f_k & \text{if } i = t_k \\ 0 & \text{otherwise} \end{array} \right\} \text{ for } \; i \in V \; \text{ and } \; 1 \leq k \leq l,$$

$$x_u^k \geq 0, \quad \text{for } \; u \in E \; \text{ and } \; 1 \leq k \leq l,$$

$$\sum_{k=1}^{l} x_u^k \leq b_u \quad \text{for } \; u \in E.$$

The Reduction

We first present a construction using *special arcs*. The flow through a special arc has to be equal to its capacity instead of just being not greater than it. On the diagrams a special arc has a label of the form "$(= Y)$", which means that the flow on this arc has to be equal to Y. Later on we show how to get rid of special arcs.

The picture below shows the general structure of our reduction. Circles with a plus indicate the sources of commodities, and with a minus — the destinations. In our construction we have n principal commodities, which correspond to n variables in LP. We denote the flows of these commodities by $x_1, x_2, \ldots, x_n$. The objective function is the same as in LP: maximize $\sum c_j x_j$.

The network we construct is acyclic. For each commodity there is only one path from its source to its destination. For each line on the diagram there is only one commodity which can pass through it. All these commodities are different. The first column in the diagram consists of *multipliers*. When flow of value x_j enters multiplier M_j, then there is flow of value $|a_{1j}|x_j$, $|a_{2j}|x_j$, $\ldots$, $|a_{mj}|x_j$, on its m output lines, respectively.

The flows from the multipliers pass to the adders, which form appropriate sums to be checked for equality with b_i's (using special arcs). Had all a_{ij}'s been nonnegative, our general scheme would have been complete at this point. We use inverters to deal with negative coefficients. An inverter changes the input flow $y \leq M$ to flow $M - y$. If there are r_i negative coefficients in $\sum_j a_{ij} x_j$, then the flow on the output of adder A_i is equal to $\sum_j a_{ij} x_j + r_i M$. We want this flow to be equal to $b_i + r_i M$.

Basic Black Boxes

Now we describe the construction of simple black boxes: inverters, adders, and duplicators. Duplicators will be needed in the construction of multipliers. Multipliers are more complex and will be described in a separate section.

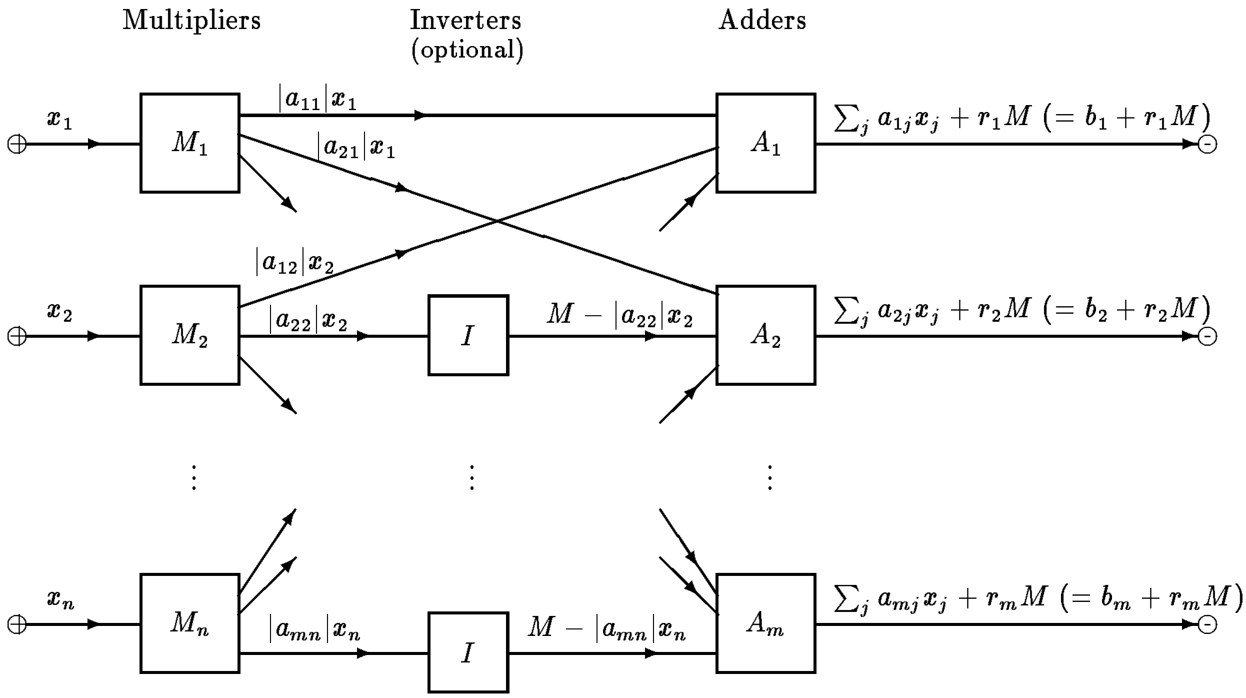
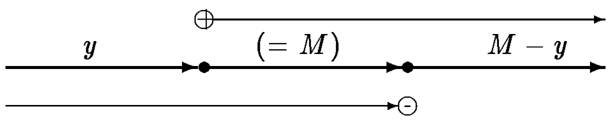


Figure 7: The main construction

Thick arrows on the diagrams denote arcs of the graph. Thin arrows show commodities. A circle denoting a source or a destination indicates that the nearest vertex is the source or the destination of the appropriate commodity.

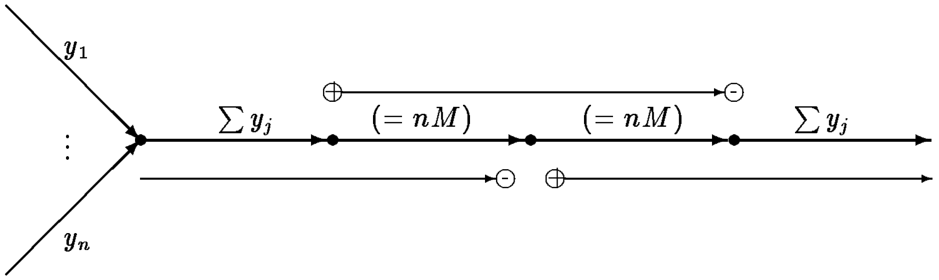
Inverter



The middle arc forces the sum of the flows of the incoming commodity and the outgoing commodity to be equal to M.

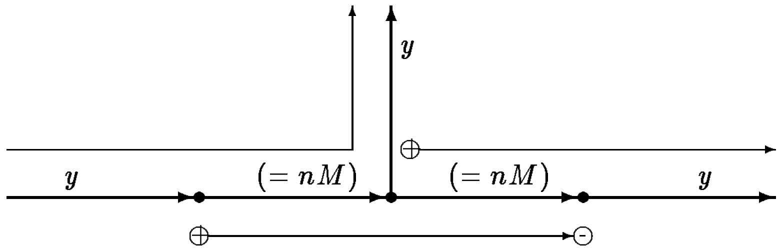
Adder

The first bottom thin arrow corresponds to all incoming commodities. They have the same destination. Actually the adder could be simpler: one vertex with n incoming arc and one outgoing arc. The reason for our construction is to have one outgoing commodity and keep the convention that there is flow of exactly one commodity on every line on the diagram 7. Observe that here we have special arcs with capacities nM. This is because in our global construction we can have a flow of value up to M on each input to an adder.



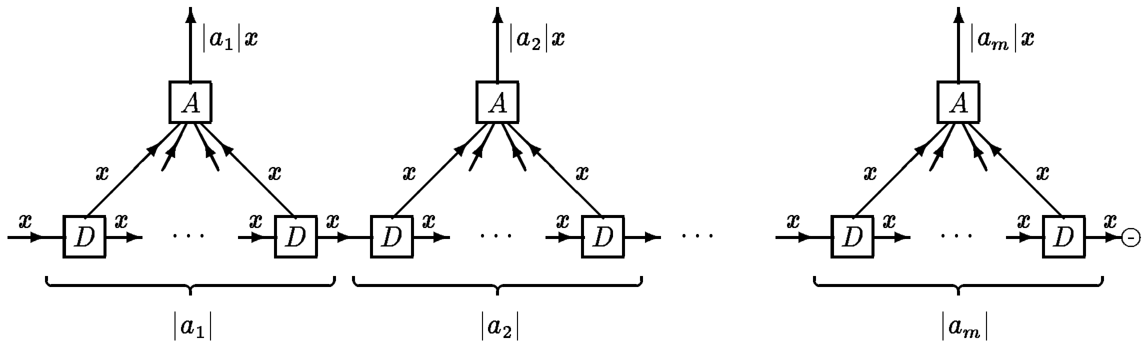
Duplicator

A duplicator sends out on each of its two outgoing lines a flow of value equal to the value of the incoming flow. Duplicators are essential in the construction of multipliers.



Multiplier

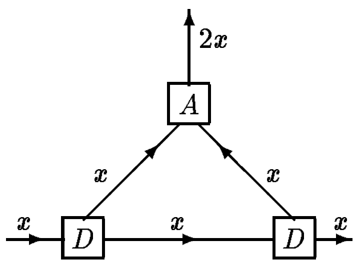
The following construction is straightforward. Unfortunately it is not polynomial.



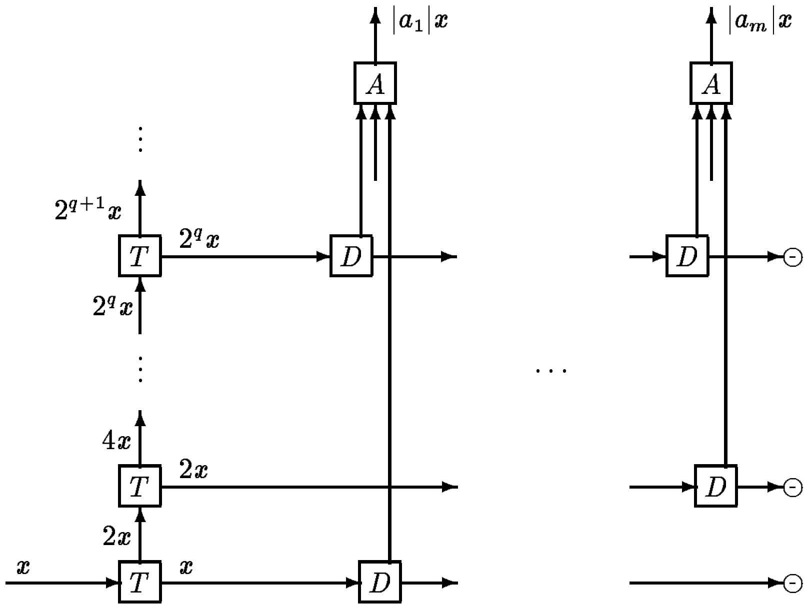
To get a polynomial construction we perform multiplications by successive doubling. To multiply x by a we generate values x, $2x$, $4x$, ..., and sum up the appropriate terms, according to the binary representation of a.

$$xa = 2^{i_1}x + 2^{i_2}x + \cdots + 2^{i_q}x, \quad \text{where } a = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_q}, \;\; i_1 < i_2 < \cdots < i_q.$$

The following gadget doubles the input flow. The value of the flow on one of the outputs is equal to the value of the input flow. The value of the flow on the other output is twice the value of the input flow. We will denote this gadget with letter T.



This is the construction of (polynomial) multiplier.



Getting Rid of Special Arcs

Let $y_1, y_2, \ldots, y_p$ denote the flows of p commodities on an arc e with capacity C. We want to impose the equality on this arc:

$$y_1 + y_2 + \cdots + y_p = C.$$

It can be done by adding $B \sum_j y_j$ to the objective function, where B is some large number. If B is large enough, then any optimal solution has to saturate e.

We have to apply this transformation to all special arcs in our construction. We can find an appropriate B which has size polynomial in the size of LP.

Lecture #14: May 27, 1992
Scribe: Avi Ziv

6 Interior Point Algorithm

This lecture is based on C. C. Gonzaga, "*An Algorithm for Solving Linear Programming in $O(n^3L)$ Operations*" [4].

6.1 Definition of the problem

The problem we want to solve is the regular linear programming problem. We want to minimize an objective function $\bar{c} \cdot x$ under the constrains $\bar{A}x = b$ and $x \geq 0$. In order to simplify the problem we make the following assumptions:

- $\bar{A}$ has full rank, *i.e.*, $(\bar{A}\bar{A}^T)^{-1}$ exist.
- The optimal value is 0.
- The feasible set is bound.
- Initial interior solution, x^0, is available.

6.2 The Algorithm

In the algorithm shown here we try to minimize a potential function $f(x)$ defined as

$$f(x) = q \log(\bar{c} \cdot x) - \sum_{i=1}^{n} \log x_i, \qquad \text{where } q = n + \sqrt{n}$$

We will do it by moving, in small steps, in opposite direction of a projection of the gradient of the potential function given by

$$\bigtriangledown f(x) = \frac{q}{\bar{c} \cdot x}\bar{c} - [x^{-1}].$$

To make life easier we will scale the problem such that the current point x will move to $\underline{1}$ ($\underline{1}$ is the unit vector $(1, 1, \ldots, 1)$.)

Figure 8 shows the Interior Point Algorithm and the next sections explain it and prove its correctness.

$k \leftarrow 0$
repeat
  scaling
    $A \leftarrow \bar{A}D^{-1}$
    $c \leftarrow \bar{c}D^{-1}$
    $y \leftarrow Dx^k$
  projection
    $h \leftarrow P(-\bigtriangledown f(\underline{1}))$, where $P = (\underline{1} - A^T(AA^T)^{-1}A)$
    $y^* \leftarrow \underline{1} + \lambda h$, where $\lambda = \frac{0.3}{\|h\|}$
  transform back
    $x^{k+1} \leftarrow D^{-1}y^{-1}$
    $k \leftarrow k+1$
until $\bar{c}x < \epsilon$

Figure 8: The Interior Point Algorithm

Scaling

Scaling transforms the current point, x^k, to the unit vector, $\underline{1}$, using the transformation matrix D

$$D = \text{diagonal}(x_1^{-1}, x_2^{-1}, \ldots, x_n^{-1}).$$

This gives

$$Ay = \bar{A}DD^{-1}x = \bar{A}x$$
$$cy = \bar{c}DD^{-1}x = \bar{c}x$$

so the original problem is not effected. Scaling serves two purposes. First, after scaling is done, relatively large step can be taken in any direction without violating the nonnegativity constraints. Second, scaling moves the current point to a point where the potential function is well-approximated by its gradient..

If we look at the potential function after the scaling we get

$$f(y) = q\log(c\cdot y) - \sum_{i=1}^{n}\log y_i = q\log(\bar{c}\cdot x) - \sum_{i=1}^{n}\log D_i x_i = f(x) + \sum_{i=1}^{n} D_i$$

so every change in the potential function of y is the same for x because during the transform from x to y and back we gain and lose the same amount, $\sum_{i=1}^{n} D_i$.

Projection

In the projection step we project the gradient of the potential function into the null space of A. The projection matrix is

$$P = (I - A^T(AA^T)^{-1}A).$$

We will show here that the projection of any vector is in the null space of A and that projection of projected vector is the projected vector itself. The complete proof of the fact that P is the projection matrix is left as an exercise.

1. To show that the projection of a vector is in the null space of A we need to show that for all y $AP(y) = 0$.
$$AP(y) = A(I - A^T(AA^T)^{-1}A)y = (A - A)y = 0$$
2. To show that projection of projection is the same as a projection we need to show that $P(P(\cdot)) = P(\cdot)$
$$\begin{aligned}
P(P(\cdot)) &= (I - A^T(AA^T)^{-1}A)(I - A^T(AA^T)^{-1}A) \\
&= (I - 2A^T(AA^T)^{-1}A + A^T(AA^T)^{-1}AA^T(AA^T)^{-1}A) \\
&= (I - A^T(AA^T)^{-1}A) \\
&= P(\cdot)
\end{aligned}$$

6.3 Convergence

In this section we show that in every step of the algorithm the potential function decreases by at least constant value. The proof is done using three lemmas.

Lemma 49

$$f(\underline{1} + \lambda h) \leq f(\underline{1}) + \lambda(\bigtriangledown f(\underline{1}))^T h + 2\lambda^2 \|h\| \qquad \text{for } \lambda\|h\| \leq \tfrac{1}{2}.$$

Proof: We start the proof by stating simple calculus result

$$\begin{aligned}
\log(1+\delta) &= \delta - \frac{\delta^2}{2(1-\theta)^2}, \qquad 0 \leq \theta \leq \delta \\
&\geq \delta - \frac{\delta^2}{2(1-|\delta|)^2} \\
&\geq \delta - 2\delta^2, \qquad \text{for } |\delta| < \tfrac{1}{2}.
\end{aligned}$$

We also can see that $\underline{1} + \lambda h > 0$ because of the way we chose λ. we divide the potential functions into to parts

$$f(y) = \overbrace{q \log(c \cdot y)}^{f_1} - \overbrace{\sum_{i=1}^{n} \log y_i}^{f_2}$$

$$\begin{aligned}
f_2(\underline{1} + \lambda h) &= \sum_{i=1}^{n} \log(1 + \lambda h_i) \\
&\geq \sum_{i=1}^{n} \lambda h_i - 2\sum_{i=1}^{n} \lambda^2 h_i^2, \qquad \text{for } |\lambda h_i| < \tfrac{1}{2} \\
&= \underbrace{f_2(\underline{1})}_{0} + \lambda \underbrace{(\bigtriangledown f_2(\underline{1}))^T}_{\underline{1}} h - 2\lambda^2 \|h\|^2.
\end{aligned}$$

$f_1(\cdot)$ is concave function so

$$f_1(\underline{1}+\lambda h) \leq f_1(\underline{1}) + \lambda(\nabla f_1(\underline{1}))^T h$$

And using both bounds for f_1 and f_2 gives the required bound on f. ∎

Lemma 50

$$||h|| \geq 1.$$

Proof: Let $\hat{y}$ be the optimal solution so $c\hat{y} = 0$. Note that $(\hat{y} - \underline{1}) \in \text{Null}(A)$ because both $\hat{y}$ and $\underline{1}$ are solutions of $Ay = b$.

$$\begin{aligned} h(\hat{y} - \underline{1}) &= -(P \nabla f(\underline{1}))(\hat{y} - \underline{1}) \\ &= -\nabla f(\underline{1})(\hat{y} - \underline{1}) \\ &= (-\frac{qc}{c \cdot \underline{1}} + \underline{1})(\hat{y} - \underline{1}) \\ &= -\frac{qc}{c \cdot \underline{1}}\hat{y} + \underline{1}\hat{y} + q - n \\ &= \sum_{i=1}^{n} \hat{y}_i + \sqrt{n} \\ &\geq ||\hat{y}|| + \sqrt{n}. \end{aligned}$$

So

$$||h||\,||\hat{y} - \underline{1}|| \geq ||\hat{y}|| + \sqrt{n}.$$

or

$$||h|| \geq \frac{||\hat{y}|| + \sqrt{n}}{||\hat{y}|| + \underline{1}} = 1.$$

∎

The third lemma is the main lemma. It shows that in each iteration the potential function decreases by at least constant number.

Lemma 51

$$f(y^*) - f(\underline{1}) \leq -0.1.$$

Proof: We chose $\lambda = \frac{0.3}{||h||}$ so we can say that $\max(\lambda|h_i|) \leq 0.5$. Using the bound of Lemma 49, we can say

$$f(y^*) = f(\underline{1} + \lambda h) \leq f(\underline{1}) + \lambda \overbrace{(\nabla f(\underline{1}))^T h}^{-||h||^2} + 2\lambda^2 ||h||^2$$

Putting the value of λ gives

$$f(y^*) - f(\underline{1}) \leq -\lambda ||h||^2(1 - 2\lambda) = -0.3||h||(1 - \frac{0.6}{||h||}).$$

Now using the bound $||h|| \geq 1$ form lemma 50 gives

$$f(y^*) - f(\underline{1}) \leq -0.3(1 - 0.6) < -0.1$$

▮

6.4 Running Time

Lemma 52 If $\epsilon = 2^{-L}$ then the running time of the algorithm is $O(nL)$ iterations, where n is the number of variables and L is the number of bits needed to represent the problem.

Proof: Let k be the number of iteration performed until the stopping condition is met. As the size of the input is L we can say that the initial point, x^0, satisfies $x^0 \leq 2^L$, or $f(x^0) = O(L)$. Because the feasible set is bound, there is some constant M such that $\sum_{i=1}^{n} \log x_i^k < M$ In Lemma 51 we showed that in each iteration $f(x)$ decreases by at least 0.1 so

$$f(x^k) \leq f(x^0) - 0.1k = O(L) - 0.1k.$$

The value of ϵ that we chose gives us

$$f(x^k) = q \log(\hat{c}x^k) - \sum_{i=1}^{n} \log x_i^k \geq -qL - M.$$

Combining the two equations above and using the fact that $M = O(L)$ and $q = O(n)$ gives

$$k \leq 10(O(L) + qL) = O(nL).$$

▮

After x^k is found, we find a vertex for which the objective function value is at least as good as the value at x^k. Such a vertex can be found by moving in the direction of the objective function until a boundary is reached and then continue on the smaller dimension problem. Using Cramer's rule we can show that this vertex is the optimal.

Remarks

1. The value of λ we used is good for theoretical proof. In practice, line search is used to find a better value of λ.

2. We can improve the efficiency of the algorithm by using an approximation to the matrix A. Because the changes in x between iterations are small we can use old values of A (and thus P) and and calculate new A only after big move is made.

7 L^3 Lattice Reduction

The following lectures are based on the book by M. Grötschel, L. Lovász, and A. Schrijver [5].

A *lattice* generated by a set n-dimensional vectors is the set of points in R^n which are *integer* linear combination of the vectors, *i.e.*

$$\mathcal{L}[b_1, b_2, \ldots, b_m] = \{\sum_{i=1}^{m} \lambda_i b_i \mid \lambda_i \in \mathbb{Z}\}.$$

Figure 9 shows an example of two dimensional lattice created by the vectors $(3, 1)$ and $(1, 2)$.

This definition makes sense for any set of vectors; however, in this most general sense it is possible to define lattices which have undesirable properties (for example, there may be no lower bound on the distance between lattice points. We therefore restrict ourselves to a special case, namely the case where $m = n$ and the generators form a basis in $\mathbb{R}_n$. This is not as big a restriction as it may seem—for example, if the generators are all rational then it is always possible to find a different set of generators for the same lattice which do form a basis.

7.1 Why Do We Need Short Vectors ?

We will present the L^3 *lattice reduction* algorithm that finds a "relatively short" vector in a given lattice. Before presenting the algorithm, we will discuss several related results.

In one dimension, the shortest vector in the lattice is the greatest common divisor of the vectors generating the lattice, and it can be found using the Euclid's GCD algorithm. In two dimensions there is a similar algorithm, due to Gauss, in which the longer vector is replaced by the difference of it and the shorter vector multiplied by an appropriate integer. Notice that this operation does not change the lattice. The correctness and complexity of this algorithm are left as an exercise.

Theorem 53 (Dirichlet) Let $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$ then for every $\epsilon > 0$, $\exists q \in \mathbb{Z}$ such that $1 \leq q \leq \epsilon^{-n}$ and $\exists p_1, \ldots, p_n \in \mathbb{Z}$ such that $\forall i : \quad |q\alpha_i - p_i| \leq \epsilon$.

Proof: Let $N = \frac{1}{\epsilon}$ and $x = (\alpha_1, \ldots, \alpha_n)$. Define $x_q = qx - [qx]$, where $[qx]$ is the closest integer coordinatewise. Observe that for all q, x_q is in the unit cube centered at the origin. Divide this cube into cubes of size $\epsilon \times \epsilon \times \cdots \times \epsilon$. There are N^n such cubes.

Case 1: There exists $1 \leq q \leq \epsilon^{-n}$ such that x_q in the middle ϵ-cube. In this case we are done.

Case 2: For all $1 \leq q \leq \epsilon^{-n}$, we have that x_q is not in the middle ϵ cube. In this case, by the pigeonhole principle, there are two integers q_1 and q_2 such that x_{q_1} and x_{q_2} are in the same ϵ-cube, *i.e.* $\forall i : \quad |x_{q_1}(i) - x_{q_2}(i)| \leq \epsilon$. Choose $q = q_1 - q_2$ and $p = [q_1 x] - [q_2 x]$.

$$|q\alpha_i - p_i| = |\alpha_i(q_1 - q_2) - [q_1 x(i)] + [q_2 x(i)]| = |x_{q_1}(i) - x_{q_2}(i)| \leq \epsilon$$

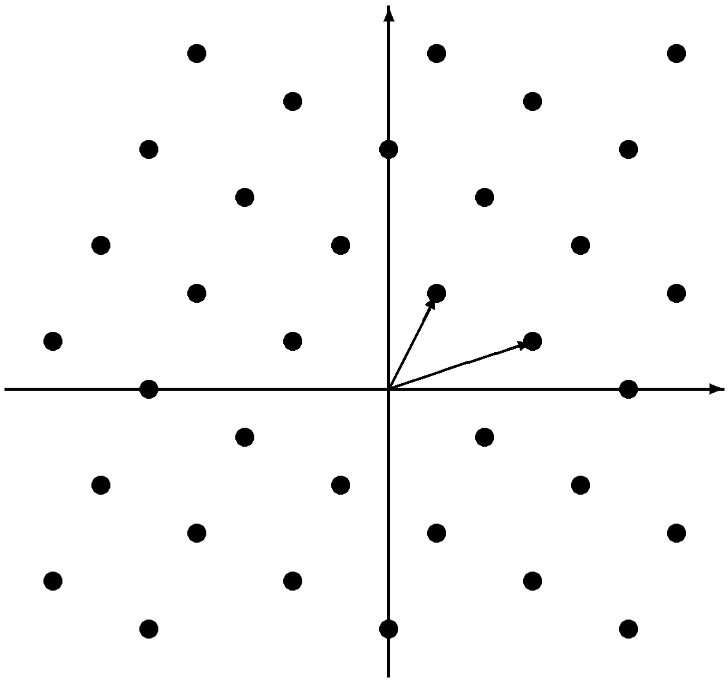


Figure 9: 2 Dimensions lattice generated by the vectors $(3, 1)$ and $(1, 2)$.

∎

The proof does not gives us a constructive way to find q. We will show that by applying the L^3 algorithm, we can find $q \leq 2^{\frac{n(n+1)}{4}} \epsilon^{-n}$.

As an example of usage of the above Theorem, consider an equality

$$ax = \sum_i a_i x_i = b \tag{21}$$

where $a = (a_1, a_2, \ldots, a_n) \in \mathbb{Z}^n$, $b \in \mathbb{Z}$, and $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$. Our goal is to round x to $x' = (p_1/q, p_2/q, \ldots, p_n/q)$, where $p_i, q \in \mathbb{Z}$ and such that the equality in (21) will be preserved, *i.e.* $ax' = b$. We also would like to use as small as possible q so that the coordinates of x' will be represented by a relatively small number of bits. Note that we can not expect a simple rounding to preserve the equality in (21).

Choose ϵ small enough so that $\|a\|_1 = \sum_i |a_i| < \frac{1}{\epsilon}$. Suppose we found q and $\{p_i\}$ such that Theorem 53 holds. Replacing x_i with $\frac{p_i}{q}$ gives

$$b - \|a\|_1 \cdot \frac{\epsilon}{q} \leq \sum_{i=1}^{n} a_i \frac{p_i}{q} \leq b + \|a\|_1 \cdot \frac{\epsilon}{q}$$

Since $\|a\|_1 < \frac{1}{\epsilon}$ we have:

$$b - \frac{1}{q} < \sum_{i=1}^{n} a_i \frac{p_i}{q} < b + \frac{1}{q}$$

As the sum must be an integer multiply of $\frac{1}{q}$ we get

$$\sum_{i=1}^{n} a_i \frac{p_i}{q} = b$$

Lecture #15: June 1, 1992
Scribe: David Karger

Application: Diaphontine Approximation

Last time, we were working on solving the following problem: given a real vector $x = \alpha_1, \ldots, \alpha_n$, find a "good" rational vector approximation. The particular form we studied is called the problem of *Simultaneous Diaphontine Approximation*. We are given some error parameter ϵ, and want to find a common denominator $q \leq \epsilon^{-n}$ and numerators $p_1, \ldots, p_n$ such that $|q\alpha_i - p| \leq \epsilon$. We now show how this problem can be reduced to the problem of finding a short vector in a lattice. Let $e_1, \ldots, e_n$ be the standard basis in $\mathbb{R}^n$. Consider the lattice $\mathcal{L}[e_1, \ldots, e_n, x]$. It is easy to see that the lattice generated by these vectors consists precisely of all vectors of the form $(q\alpha_1 - p_1, \ldots, q\alpha_n - p_n)$ for integer q and p_i. It follows that all that we need to find in order to solve our Diaphontine Approximation Problem is a lattice vector which is contained in the ϵ-cube.

The L^3 algorithm which we will study finds a (relatively) short lattice vector in any lattice which is specified by a basis. We can use the algorithm to solve our problem as follows. First, we extend our lattice to $\mathbb{R}^{n+1}$ by adding a new components to all the vectors mentioned above: 0 is appended to each of the e_i, and some small number δ is apppended to x. These extended vectors clearly form a basis in $\mathbb{R}^{n+1}$. Let

$$R = \begin{pmatrix} e_1 & e_2 & \ldots & e_n & x \\ 0 & 0 & 0 & 0 & \delta \end{pmatrix}.$$

As we will see later, the L^3 algorithm finds a lattice vector y such that

$$\begin{aligned} \|y\| &\leq 2^{n/4} |\det R|^{1/(n+1)} \\ &= 2^{n/4} \delta^{1/(n+1)}, \end{aligned}$$

where $\det R$ denotes the determinant of R. Therefore, if we take $\delta = 2^{-n(n+1)/4} \epsilon^{n+1}$, we end up with $\|y\| \leq \epsilon$, which gives us the desired quality of approximation. It follows that $q\delta < \epsilon$, *i.e.* $q \leq \epsilon^{-n} 2^{n(n+1)/4}$. Note that an algorithm that approximates the shortest vector better will lead to the corresponding reduction in q.

7.2 Some Properties of Lattices

As was mentioned before, we are going to restrict ourselves to the case of lattices generated by a basis $(b_1, \ldots, b_n)$. In fact, we will restrict further to the case where our basis vectors are vectors over the rationals, rather than the reals. Since we can immediately clear denominators, we will in fact assume the the generators are integer vectors. The notation $(b_1 \cdots b_n)$ will denote the matrix whose columns are equal to the vectors listed.

Theorem 54 If $\mathcal{L}(b_1, \ldots, b_n) = \mathcal{L}(b'_1, \ldots, b'_n)$, then $|\det(b_1 \cdots b_n)| = |\det(b'_1 \cdots b'_n)|$.

Proof: Write each set of generators in terms of the other:

$$\begin{aligned} b'_i &= \sum \lambda_{ij} b_j \qquad (\lambda_{ij} \in \mathbb{Z}). \\ b_i &= \sum h_{ij} b_j \qquad (h_{ij} \in \mathbb{Z}). \end{aligned}$$

This defines two matrices Λ and H such that $B' = \Lambda B$ and $B = HB'$. It follows that $\Lambda H B' = B'$. Since $\det B' \neq 0$, we immediately deduce that $\det \Lambda H = 1$. Since Λ and H are integer matrices and must therefore have integer determinants, we immediately deduce that $\det H = \det \Lambda = \pm 1$, and the result follows from the fact that $B' = \Lambda B$. ∎

Definition 55 The *Gramm-Schmidt orthogonalization* of a basis $\{b_i\}$ is another basis defined by

$$\begin{aligned} b_1^* &= b_1 \\ b_i^* &= b_i - \sum_{j<i} \frac{b_i b_j^*}{\|b_i b_j^*\|} b_j^*. \end{aligned}$$

Theorem 56 $|\det(b_1, \cdots, b_n)| = |\det(b_1^*, \cdots, b_n^*)| = \|b_1^*\| \cdots \|b_n^*\|$.

Proof: Let $B^* = (b_1^*, \cdots, b_n^*)$. The first equality follows because the Gramm-Schmidt orthogonalization procedure can be seen as a sequence of column operations on the matrix B which transform it into B^*, with each individual column operation preserving the determinant. The first equality thus follows. To deduce the second equality, note that $|\det B^*| = \sqrt{|\det(B^*)^t B|}$, and that $(B^*)^t B$ is a diagonal matrix with diagonal entries $\|b_1^*\|^2, \ldots, \|b_n^*\|^2$. ∎

Corollary 57 (Hadamard's Inequality) $|\det B| \leq \prod \|b_i\|$.

Proof: Since b_i^* is a projection of b_i, $\|b_i^*\| \leq \|b_i\|$. ∎

This theorem can be interpreted in a more geometrical way, if we observe that taking the product of the lengths of the orthogonalized basis is a natural way to measure volume (this can be clearly seen in two dimensions).

Fact 58 $|\det(b_1 \cdots b_n)|$ is the volume of the parallelipiped defined by the vectors b_i.

Definition 59 The length of the shortest vector in $\mathcal{L}$ is denoted by $\lambda(\mathcal{L})$.

Theorem 60 $\lambda(\mathcal{L}) \geq \min ||b_i^*||$.

Proof: Suppose $x \in \mathcal{L}$, so $x = \sum \lambda_i b_i$. Let λ_k be the last non-zero coefficient. Then

$$\begin{aligned} x &= \lambda_k b_k + \sum_{i<k} \lambda_i b_i \\ &= \lambda_k b_k^* + \sum_{i<k} \lambda_i' b_i^* \end{aligned}$$

It follows from the fact that the b_i^* are orthogonal that

$$\begin{aligned} ||x|| &= |\lambda_k| ||b_k^*|| + || \sum_{i<k} \lambda_i' b_i^* || \\ &\geq |\lambda_k| ||b_k^*|| \\ &\geq ||b_k^*||. \end{aligned}$$

∎

The above theorem gives a lower bound on $\lambda(\mathcal{L})$. The following one (taken without proof) yields an upper bound.

Theorem 61 (Minkowski) $\prod ||b_i|| \leq n^n |\det B|$.

Corollary 62 $\lambda(\mathcal{L}) \leq n \sqrt[n]{\det B}$.

Proof: Just note that Minkowski's theorem bounds the "average" (in the sense of products) of the $||b_i||$. ∎

Note that Theorem 61 together with Corollary 57 imply that:

$$\prod ||b_i|| \leq n^n |\det B| \leq n^n \prod ||b_i||$$

.

7.3 The L^3 Algorithm

The above theorems suggest several constructive questions about lattices:

1. Find $\lambda(\mathcal{L})$ (and a lattice vector of this length).
2. Find a lattice vector of length $n \sqrt[n]{\det \mathcal{L}}$.
3. Find a lattice basis with $\prod ||b_i|| \leq n^n \det \mathcal{L}$.

The L^3 algorithm does not completely solve this question, but is a step in this direction. It finds approximate solutions to each of the above questions, with multiplicative error factors of $2^{(n-1)/2}$, $2^{(n-1)/4}$, and $2^{n(n-1)/4}$ respectively. The L^3 algorithm works by finding a basis of "short" vectors:

Definition 63 A *reduced basis* $b_1, \ldots, b_n$ is a basis whose Gramm-Schmidt orthogonalization $b_i^*, \ldots, b_n^*$ defines certain values μ_{ij} through the following equations:

$$b_i = b_i^* + \sum_{j<i} \mu_{ij} b_j^*$$

such that the following properties are satisifed:

$$|\mu_{ij}| \leq 1/2 \qquad \text{(pseudo-orthogonality)} \tag{22}$$

$$||b_{j+1} + \mu_{j+1,j} b_j^*|| \geq \sqrt{\frac{3}{4}} ||b_j^*|| \qquad \text{(pseudo-order)} \tag{23}$$

In the next lecture, we will show how to construct such a basis. For now, we show that if we have found such a basis, it satisfies the properties we want.

Lemma 64 $||b_1|| \leq 2^{(n-1)/2} \lambda(\mathcal{L})$.

Proof: Observe that

$$\begin{aligned} \frac{3}{4} ||b_{i-1}^*||^2 &\leq ||b_i^* + \mu_{i,i-1} b_{i-1}^*||^2 \\ &= ||b_i||^2 + |\mu_{i,i-1}|^2 ||b_{i-1}^*||^2 \\ &\leq ||b_i^*||^2 + \frac{1}{4} ||b_{i-1}^*||^2. \end{aligned}$$

It follows that $||b_i^*||^2 \geq \frac{1}{2} ||b_{i-1}^*||^2$. Thus, by induction, for $i \geq j$, we have:

$$||b_j^*|| \leq 2^{(i-j)/2} ||b_i^*||.$$

Now note that $\det \mathcal{L} = \prod ||b_i^*||$. It follows that

$$\begin{aligned} ||b_1^*||^n &\leq \prod_i 2^{(i-1)/2} ||b_i^*|| \\ &= 2^{n(n-1)/4} \prod ||b_i^*|| \\ &= 2^{n(n-1)/4} \det \mathcal{L}. \end{aligned}$$

∎

Lecture #16: June 3, 1992
Scribe: Rakesh Nigam

In this lecture we prove the last one of the properties of the reduced basis, describe the algorithm to compute the reduced basis, and sketch an application of lattice reduction to the "subset-sum" problem.

Lemma 65 $||b_1|| \ldots ||b_n|| \leq 2^{\frac{\binom{n}{2}}{2}} det(L)$

Proof: By definition of the reduced basis, we have:

$$\begin{aligned} ||b_j||^2 &= \sum_{i=1}^{j} \mu_{ji}^2 ||b_i^*||^2 \\ &= \mu_{jj}^2 ||b_j^*||^2 + \sum_{i=1}^{j-1} \mu_{ji}^2 ||b_i^*||^2 \\ &\leq ||b_j^*||^2 + \frac{1}{4} \sum_{i=1}^{j-1} ||b_i^*||^2 \end{aligned}$$

In the previous lecture we proved that for all $1 \leq i < j \leq n$ we have $||b_j^*||^2 \geq 2^{i-j} ||b_i^*||^2$. This implies that

$$||b_j||^2 \leq (1 + \sum_{i=1}^{j-1} 2^{j-i}/4) ||b_j^*||^2 \leq 2^{j-1} ||b_j^*||^2$$

Multiplying these inequalities for $j = 1, 2, \ldots, n$, we get the claim of the lemma. ∎

Theorem 66 Given a non-singular matrix $A \in Q^{n \times n}$ there is an algorithm to find the reduced basis of lattice $\mathcal{L}(A)$ in polynomial time.

Proof: We first give the algorithm which clearly achieves the desired result if it ever terminates, and then we prove that it terminates in polynomial time. Without loss of generality assume that $a_1, \ldots, a_n$ are integral.

Step (0): *(Initialization)* We maintain a basis $(b_1, \ldots, b_n)$ which we initialize by $(b_1, \ldots, b_n) := (a_1, \ldots, a_n)$. So we start with an ordered basis. Let $(b_1^*, \ldots, b_n^*)$ be the Gram-Schmidt orthogonalization of this basis.

Step (I) :

For $j := 1, 2, \ldots, n$ and, given j
 for $i := j-1, j-2, \ldots, 1$
 $b_j \leftarrow b_j - [\mu_{ji}] b_i$
 for $k = i, i-1, \ldots, 1$
 $\mu_{jk} \leftarrow \mu_{jk} - [\mu_{ji}]\mu_{ik}$

Step (II) : If there is a subscript j violating $||b^*_{j+1} + \mu_{j+1,j} b^*_j||^2 \geq \frac{3}{4}||b^*_j||^2$ for $j = 1, \ldots, n-1$, then interchange b_j and b_{j+1} and return to Step (I). Else terminate and return the current $b_1, b_2, \ldots, b_n$.

It is clear that if the algorithm terminates than it produces a weakly reduced basis. We now show that the algorithm terminates in polynomial time. Observe that Step (I) does not change $b^*_1, b^*_2, \ldots, b^*_n$. These vectors change only as a result of the switch that occurs at Step (II).

Define a potential function $\Phi = \Phi(b_1, \ldots, b_n)$

$$\Phi = \prod_{i=0}^{n-1} ||b^*_i||^{2(n-j)}$$

Let B_i be a matrix with columns $b_1, b_2, \ldots b_i$ and let B^*_i have columns $b^*_1, b^*_2, \ldots b^*_i$. Let $D_i = B^t_i B_i$ and $D^*_i = (B^*_i)^t (B^*_i)$. Recall that in the last lecture we have proved that

$$\prod_{k=1}^{i} ||b^*_k||^2 = \det(D^*_i) = \det(D_i).$$

This and the fact that Φ can be rewritten as:

$$\Phi = \prod_{k=1}^{n} \det(D^*_k) = \prod_{k=1}^{n} \det(D_k)$$

imply that Φ is integral.

Consider the effect of the switch of b_j and b_{j+1}. To emphasize the difference between the basis and the ortogonalization before and after the switch, let c_i, c^*_i and C^*_i denote the correspoinding entities after the switch. Note that $c^*_i = b^*_i$ unless $i \in \{j, j+1\}$. The switch occured because

$$\frac{3}{4}||b^*_j||^2 > ||b^*_{j+1} + \mu_{j+1,j} b^*_j||^2 = ||c^*_j||^2.$$

Hence

$$\det((B^*_i)^t (B^*_i)) = \det((C^*_i)^t (C^*_i)) \quad \text{for all} \quad i \neq j \tag{24}$$

$$\frac{3}{4}\det((B^*_j)^t (B^*_i)) \geq \det((C^*_j)^t (C^*_j)) \tag{25}$$

In other words, each switch causes a constant factor reduction in the value of Φ. Since $\Phi \leq (\max_i ||a_i||)^{O(n^2)}$, we get polynomial running time. ∎

7.4 Subset Sum as Application of the L^3 Algorithm

The subset Sum problem is defined as follows: Given n positive integers $a_1, \dots, a_n$ and a positive integer b, find a subset $\{i_1, \dots, i_k\}$ of $\{1, \dots, n\}$ such that $a_{i_1} + \dots + a_{i_k} = b$

Rewriting this in terms of vector $a = (a_1, a_2, \dots, a_n)$, we look for a non-trivial $0/1$ vector x and $0/1$ scalar y, such that

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n - by = 0 \tag{26}$$

This problem is known to be $\mathcal{NP}$-complete. Consider the case when we know that a solution exists. The following lemma will imply that under certain (reasonable) conditions we can expect that the problem will satisfy the following condition:

Condition (*): The problem has a unique non-zero $\{0,1\}$-solution, and every nonzero integral solution that is not a multiple of this unique $0/1$ solution has at least one component whose absolute value is at least 2^n.

Lemma 67 Let $\{i_1, \dots, i_k\} \subseteq \{1, \dots, n\}$ be nonempty and let N be a positive integer. Suppose that the integers $\{a_i | i = 1, \dots, n\}$ are chosen independently and uniformly from $\{1, \dots, N\}$ and let $b := a_{i_1} + \dots + a_{i_k}$. Then *Condition (*)* is satisfied with probability larger than $1 - \frac{2^{(n+2)n}}{N}$.

Proof: By the choice of b the problem has a nonzero $\{0,1\}$-solution $(x_1, \dots, x_n, y)$. Suppose there exists a nonzero integral vector $(\tilde{x}_1, \dots, \tilde{x}_n, \tilde{y})$ satisfying $a\tilde{x} - b\tilde{y} = 0$ and $|\tilde{x}| < 2^n$ for $i = 1, \dots, n$ and $|\tilde{y}| < 2^n$ that is not a multiple of $(x_1, \dots, x_n, y)$.

Define

$$\bar{x}_i := \begin{cases} \tilde{x}_i - \tilde{y} & \text{if } \ i \in \{i_1, \dots, i_k\} \\ \tilde{x}_i & \text{otherwise} \end{cases}$$

Observe that because $(\tilde{x}, \tilde{y})$ are not multiples of (x, y), the vector $\bar{x} \neq 0$. Moreover:

$$a_1\bar{x}_1 + \dots + n\bar{x}_n = 0 \tag{27}$$

$$-2^{(n+1)} < \bar{x}_i < 2^{(n+1)} \tag{28}$$

Consider a fixed vector $\bar{x}$. Since the a_i are chosen uniformly and independently from $\{1, \dots, N\}$, the probability that a_i satisfy (27) is not larger than $\frac{1}{N}$. Since the number of integral vectors satisfying (28) is smaller than $2^{(n+2)n}$, it follows that the probability that (27) has a solution satisfying (28) is smaller than $2^{(n+2)n}/N$, which implies the claim of the lemma. ∎

The significance of *Condition (*)* is not only that it is satisfied with high probability if N is large enough, but also that, if it is satisfied, the subset sum problem can be solved in polynomial time, using basis reduction.

Lemma 68 There exists a polynomial time algorithm that, for a given positive integers $a_1, \ldots, a_n$ and b, either:

(I) Supplies a nonzero integral solution of equation (26) where all components have absolute value less than 2^n, or

(II) Prove that the subset sum problem has no solution.

In particular, if *Condition (*)* is satisfied, the algorithm solves the subset sum problem.

Proof: For $n = 1$ the problem is trivial, and hence we assume $n \geq 2$. First, consider the integral vectors $z = (x_1, \ldots, x_n, y)$ that satisfy (26), *i.e.* integral vectors z that are orthogonal to the vector $(a_1, \ldots, a_n, -y)$. It is possible to show that vectors z form a lattice. It is also known that it is possible to find a basis for such lattice in polynomial time. We omit the proof of these claims.[1]

Apply the L^3 algorithm to the basis of the resulting lattice. Let $b_1, b_2, \ldots, b_{n+1}$ be the reduced basis. Consider two cases:

1. $|b_1|_\infty < 2^n$: In this case we can output the reduced basis and claim that we are in case (I) of the lemma.
2. $|b_1|_\infty \geq 2^n$. In this case, since b_1 is a good approximation to the shortest vector in the lattice, we have that the length of shortest vector is at least

$$||b_1||/2^{(n/2)} \geq 2^{(n/2)} > \sqrt{n+1}$$

   This shows that the lattice contains no nonzero $\{0, 1\}$- vector, and thus we can conclude that (II) holds.

∎

Combining Lemmas 67 and 68, we have the following theorem

Theorem 69 There exists a polynomial time algorithm that finds a solution of the subset sum problem with probability at least $1 - \frac{2^{(n+2)n}}{N}$ if the given integers $a_1, \ldots, a_n$ are chosen independently and uniformly from $\{1, \ldots, N\}$ and if the given integer b is chosen such that the subset sum problem has a solution.

References

[1] J. Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Netwrok Flow. *SIAM J. Comput.*, 18:1057–1086, 1989.

[1] For a complete proof see [5], page 155.

[2] V. Chvátal. *Linear Programming.* W.H. Freeman and Co., New York, 1980.

[3] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988. A preliminary version appeared in *Proc. 18th ACM Symp. on Theory of Comp.*, 136-146, 1986.

[4] C. C. Gonzaga. An Algorithm for Solving Linear Programming in $O(n^3L)$ Operations. In N. Megiddo, editor, *Progress in Mathematical Programming*, pages 1–28. Springer Verlag, Berlin, 1989.

[5] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer Verlag, 1988.

[6] C. Haibt-Norton, S. Plotkin, and É. Tardos. Using Separation Algorithms in Fixed Dimension. *Journal of Algorithms*, 1991. Accepted for Publication.

[7] L.G. Khachian. A polynomial algorithm for Linear Programming. *Doklady Akad. Nauk USSR*, pages 1093–1096, 1979. English translation: *Soviet Math. Doklady*, 20, no.1, pp.191-194.

[8] P. Klein, S. A. Plotkin, C. Stein, and É. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *Siam Journal on Computing*, 1992. Accepted for Publication.

[9] T. Leighton, F. Makedon, S. Plotkin, C. Stein, S. Tragoudas, and É. Tardos. Fast approximation algorithms for multicommodity flow problem. In *Proc. 23th ACM Symposium on the Theory of Computing*, pages 101–111, May 1991.

[10] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.

[11] A.Y. Levin. On an algorithm for convex functions minimization. *Doklady Akad. Nauk USSR*, (160):1244–1247, 1965. English translation: *Soviet Math. Doklady* 6, pp.286-290.

[12] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[13] S. Plotkin, D. Shmoys, and É Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proc. 32nd IEEE Annual Symposium on Foundations of Computer Science*, October 1991.

[14] A. Schrijver. *Theory of Linear and Integer Programming.* J. Wiley & Sons, 1986.

[15] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318 – 334, 1990.

[16] N.Z. Shor. Utilization of the operation of space dilatation in the minimization of convex functions. *Kibernetika*, 6:6–12, 1970. English translation: *Cybernetics* 13, pp.94-96.

[17] É. Tardos. A strongly polynomial algorithm for solving combinatorial linear programs. *Oper. Res.*, pages 250–256, 1986.

[18] B. Yamnitsky and L. Levin. An old linear programming algorithm runs in polynominal time. In *Proc. of 23rd IEEE Symp. on Foundation of Computer Science*, pages 327–328, 1982.

[19] D.B. Yudin and A.S. Nemirovskii. Informational complexity and effective methods for solving convex extremum problems. *Econ. i. Math. Metody*, 12:357–369, 1976. English translation: *Maketon* 13 (3) (1977), pp. 25-45.