ORIE 6300 Mathematical Programming I November 4, 2008

Lecture 18

Lecturer: David P. Williamson *Scribe: Emma Qiu Wang*

1 Dantzig-Wolfe Decomposition

Recall from last time that we consider LPs of the following form:

$$\begin{array}{lccccccccll} \min & c_1^T x_1 & + & c_2^T x_2 & + & ... & + & c_m^T x_m & & \\ \text{s.t.} & A_{01}x_1 & + & A_{02}x_2 & + & ... & + & A_{2m}x_m & = b_0 & \\ & A_{11}x_1 & & & & & & & = b_1 & \\ & & & A_{22}x_2 & & & & & = b_2 & \\ & & & & & ... & & & & \\ & & & & & & & A_{mm}x_m & = b_m & \\ & & & & & & & x_j & \geq 0 & \forall j = 1, 2, ..., m, \end{array}$$

where $A_{ij} \in \Re^{m_i \times n_j}$, $c_j, x_j \in \Re^{n_j}$, and $b_i \in \Re^{m_i}$. So we have a set of linking constraints (row 1) followed by m smaller systems. Recall that each of the A_{ij} are matrices of size $m_i \times n_j$; these blocks are not necessarily square. Similarly, the $c_1, ..., c_m$, $x_1, ..., x_m$, and $b_1, ..., b_m$ are vectors. Our goal is to reformulate this problem so that we can take advantage of its special structure to solve it more easily. Our main assumption in doing this is that the following subproblem is relatively easy to solve for any cost vector $\hat{c}$:

$$\begin{array}{lll} \min & \hat{c}^T x_i & \\ \text{s.t.} & A_{ii}x_i & = b_i \\ & x_i & \geq 0 \end{array}$$

The feasible region of this subproblem is $Q_i = \{x_i \in \Re^{n_i} : A_{ii}x_i = b_i, x_i \geq 0\}$. We will assume for simplicity that Q_i is bounded. Then if we enumerate its vertices $\{v_{i1}, v_{i2}, ..., v_{iN_i}\}$, these v_{ij}'s completely define the feasible region. That is, for any $x_i \in Q_i$ we can write x_i as a convex combination of the v_{ij}. That is, $x_i = \sum_{j=1}^{N_i} \lambda_{ij} v_{ij}$ for some λ_{ij} such that $\sum_{j=1}^{N_i} \lambda_{ij} = 1$ and $\lambda_{ij} \geq 0$. Thus we can rewrite the original LP in terms of the variables λ_{ij} as follows:

$$\begin{array}{lll} \min & \sum_{i=1}^{m} \sum_{j=1}^{N_i} \lambda_{ij}(c_i v_{ij}) & \\ \text{s.t.} & \sum_{i=1}^{m} \sum_{j=1}^{N_i} \lambda_{ij}(A_{0i} v_{ij}) & = b_0 \\ & \sum_{j=1}^{N_i} \lambda_{ij} & = 1 \qquad \forall i = 1, 2, ..., m \\ & \lambda_{ij} & \geq 0 \qquad \forall i, j. \end{array}$$

This is sometimes called the *master problem.*

This new formulation reduces the number of constraints to $m_0 + m$, since there are now m_0 constraints for the linking constraints and only 1 constraint for each of the m subproblems. However, the number of variables is now huge! Therefore, to solve this problem we will want to use the revised simplex method, so that we do not need to keep track of all of these variables at once. As in the cutting stock problem, to do this, we need to be able to check whether the reduced cost corresponding to each variable is negative.

We first create the dual variables $y \in \mathbb{R}^{m_0}$ corresponding to the m_0 linking constraints, and we create $z \in \mathbb{R}^m$ corresponding to the m subproblem constraints (which set the sums of the λ_{ij} to 1). To check the reduced cost corresponding to variable λ_{ij}, we consider

$$\begin{aligned} \bar{c}_{ij} &= c_i^T v_{ij} - \left((A_{0i} v_{ij})^T y + e_i^T z\right) \\ &= c_i^T v_{ij} - y^T (A_{0i} v_{ij}) - z_i \\ &= (c_i - A_{0i}^T y)^T v_{ij} - z_i. \end{aligned}$$

which we get from taking the inner product of the dual variables with column for λ_{ij}:

$$\begin{bmatrix} y_0 & z \end{bmatrix} \begin{bmatrix} A_{0i} v_{ij} \\ \hline e_i \end{bmatrix}$$

Does there exist v_{ij} such that $\bar{c}_{ij} < 0$ or equivalently such that $(c_i - y_0 A_{0i})^T v_{ij} < z_i$? How can we answer this? Consider $\bar{c}_i = c_i - A_{0i}^T y$ in the following subproblem:

$$\begin{array}{lll} \min & \bar{c}_i^T x_i & \\ \text{s.t.} & A_{ii} x_i & = b_i \\ & x_i & \geq 0. \end{array}$$

Since we assume that Q_i is nonempty and bounded, this problem must have an optimal solution which is a vertex; call it $v = v_{ik}$ for some k. If the optimal value $\bar{c}_i v < z_i$ then $\bar{c}_{ik} < 0$, which implies that the index of variable λ_{ik} should enter the basis. If $\bar{c}_i v \geq z_i$, then this implies that $\bar{c}_{ij} \geq 0$ for all vertices v_{ij} for $j = 1, \ldots, N_i$.

Therefore, by solving the auxiliary optimization problem for each of $Q_1, ..., Q_m$ we either find a negative reduced cost column or else we prove that the current solution to the master problem is optimal.

Notice that what we have really done here is simply the revised simplex method. Using this method is good because it vastly reduced the amount of space required to solve the problem. Also, it is good computationally because the slave problems can be solved in parallel; we simply choose the column corresponding to whichever problem answers back first with negative reduced cost to enter the basis.

2 Good algorithms

Our next task will be to consider provably efficient algorithms for solving linear programs. What do we mean by this? We have said that the simplex method works very well in practice. Why isn't this enough? We'd like to get some mathematical sense of when problems have good algorithms – or are not like to have them – as long as there is roughly a correspondence between this and reality.

One of the first definitions of a good algorithm was given in the context of the perfect matching problem. In this problem, we are given an undirected graph $G = (V, E)$, with edge costs $c_e \geq 0$ for all $e \in E$, and $|V|$ even. The goal is to find a minimum-cost set of edges $M \subseteq E$ such that each vertex has exactly one edge of M incident to it.

Definition 1 (Edmonds (1965)) *An algorithm for this problem is* good *if for any input, the amount of time necessary for the algorithm to complete is no more than some polynomial function of the input size. That is* ***run time*** $\leq p(n)$*, where n is the input size and p is some polynomial.*

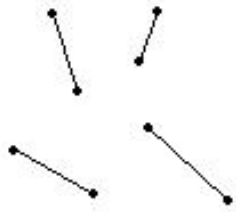

Figure 1: Example of a matching problem

Edmonds gave an $O(n^5)$ time algorithm for the perfect matching problem, where $n = |V|$. Note that this is much better than consider all possible matchings, which in a complete graph can be as many as $(n-1) \cdot (n-3) \cdots 1$ possibilities. We could not possibly consider all of these possibilities in $O(n^5)$ time, so somehow the algorithm is circumventing the complete enumeration of all options.

Now to ask the question: is the simplex method a good algorithm under this definition? The answer is no, since as we've seen, there are examples for which most of the known pivot rules require $2^n - 1$ pivots.

There are some subtleties in the definition of a good algorithm of which we should be aware. For instance, Euclid gave an algorithm for computing the greatest common denominator (gcd) of two numbers. It takes two numbers as input. What makes this a good algorithm? What is the corresponding input size? Unless the algorithm runs in constant time, we'll need to consider something different than simply saying that it is the number of input numbers. Euler gave an algorithm for this. What is the input size n for this algorithm?

In this case, the input size is the total number of bits needed to write the input numbers in binary. Then we can show that the running time of Euclid's algorithm is bounded by a polynomial in this input size.

Now for the question: Are there good algorithms for solving linear programs? The answer is yes! There are two kinds which we will be discussing in the coming weeks. The first is called the *Ellipsoid Method*, due to Khachiyan in 1979. The algorithm is not practical, but still has some nice theoretical properties. The second is a class of algorithms called *interior-point methods*, which started with a paper of Karmarkar in 1984. Both have running times bounded by polynomials in n, m, and L, where L is the number of bits needed to encode the inputs A,b, and c in binary.

Here's one more definition. A *strongly polynomial* time algorithm for LP would have a running time bounded by a polynomial in m and n; that is, the algorithm would depend only on the number of inputs, and not their size. It is a significant open research problem to show whether there exists a strongly polynomial-time algorithm for linear programming or not.

Another interesting question is to know when there does *not* exist a good algorithm for a problem. For instance, we can ask if there exists a good algorithm for the knapsack problem. We saw in a previous class an algorithm for the knapsack problem running in $O(mW^2)$ time, where W is the size of the knapsack. This is not a good algorithm, since the input size of the knapsack is $O(\log W)$, not W. Does there exist a good algorithm for the knapsack problem? This is currently unknown. There are many problems in this class that are of interest. We will discuss this in future lectures when we discuss NP-completeness.