Mathematical Programming **Lecture 27**
OR 630 Fall 2006 **November 30, 2006**
adapted from notes by Omer Hancer

1 NP-Completeness (continued)

Before starting on complexity theory, we talked about running times, depending on the input size L, to be encoded in binary. However, note that in the last 3 problems that we have considered in complexity theory, there were no numbers, hence we never mentioned the input size L. These problems were all NP-complete and were hard because of their particular combinatorial structure.

Recall that when we discussed the *Knapsack Problem*, which, in its general form, is given as follows:

$$\max_{S \subseteq \{1,\ldots,n\}} \sum_{i \in S} v_i$$

$$\text{s.t.} \quad \textstyle\sum_{i \in S} w_i \leq W$$

we gave dynamic programming algorithms that ran in $O(nW)$ time (Recall that the number of bits required to encode W is $\log_2 W$, therefore $O(nW)$ is not a polynomial running time.)

When we want to talk about efficiency, NP-complete problems are intractable – we believe that no algorithm whose running time is bounded by a polynomial can be found. (One side note: we often say that a problem Π is NP-hard; this means that if there exists a polynomial-time algorithm for this problem, then $P = NP$. In this way, optimization problems can be NP-hard, whereas NP-completeness needs to refer to a decision problem only.)

Note that there is also the issue of hardness for NP-complete problems, i.e., not all NP-complete problems are *equally hard* to solve. Let us take the Knapsack problem as an example. Although the decision version of this problem is NP-complete, it is typically not very hard to solve. For example, it is completely routine to solve large inputs by standard IP branch-and-cut methods. Furthermore, when we try to decide whether $O(nW)$ is good or bad, we can claim that if we give up some accuracy, we can find an efficient algorithm that can approximate the optimal solution (by forcing W to be small), in polynomial time; so it is not that bad. So actually, the question "Can I solve it or not?" is highly problem dependent and NP-hardness is not a completely reliable yardstick. In constrast, for example, we can say that the *Quadratic Assignment Problem* (a problem once called *violently hard*), is still hard to solve whereas the Knapsack problem is typically not that hard. Nonetheless, we will prove the following, about the decision version of the knapsack problem, where we are also given a threshold V, and ask whether there is a feasible solution of value at least V.

Claim: Knapsack Problem is NP-complete.

Proof: We will start with *Vertex Cover* problem, reduce it to *Subsetsum* problem, which in turn will be reduced to the decision version of the *Knapsack* Problem, i.e.:

$$Vertex\ Cover \prec Subsetsum \prec Knapsack\ Problem$$

Subsetsum Problem can be defined as follows:
Given an input $\{a_1, a_2, ...a_N, T\}$, does there exist $S \subseteq \{1, ..., n\}$ $s.t.$ $\sum_{i \in S} a_i = T$?

In order to reduce this problem to knapsack problem, we will set $W = V = T$. It is straightforward to verify this reduction is correct.

Vertex Cover Problem can be defined as follows:

Given $G = (V, E); k$ (where V is the vertex set and E is the edge set), does there exist $S \subseteq V$ $s.t.$ $S = |k|$ and for each edge $(uv) \in E$; $\{u, v\} \cap S \neq \emptyset$?
(Note that whether the graph is connected or not does not matter for our proof but for the sake of simplicity, let us assume it is connected.)

The idea will be to show that there is a good vertex cover if and only if there exists a subsetsum $\sum_{i \in S} a_i = T$.

Note that there are almost no numbers in our vertex cover problem (except for k), therefore we will try to create some numbers by the following way:

		$\lceil \log_2 k \rceil + 1$							e_1		e_2		e_3		…		e_{n-1}		e_n	
T.........		1	0	0	0	…	…	0	1	0	1	0	1	0			1	0	1	0
1.....	vertices	0	0	0	0			1	0											
2.....		0	0	0	:			1	0											
3.....		0	0	:				1	:											
.		:	:	:				1	:		0	1								
.		:	:	:				1	:											
.		0	0	0				1	0		0	1								
.	edges	0						0	0	1										
		:						.			0	1								
.		:						.					…	…						
.		0						0												
N.....		0						0												

A vertex cover of size k will correspond to a subset of rows of this matrix that add up to T; the first $\lceil \log_2 k \rceil + 1$ columns (bits) of the top row, specifying T give the binary representation of the number k. Then these columns are separated from the edge columns (denoted on top of the columns

by $e_1, e_2, ..., e_n$) by the thick boundary in the figure. The edges are given two columns each, in which the binary number 10 appears on the top row (corresponding to T); each edge (with its two columns) is separated from the next by the double boundary in the figure. Each row (except the first) corresponds to either a vertex, or an edge. For each row corresponding to a vertex, in the leading bits (corresponding to those columns where we encoded k in T) we encode the value 1 (that is, leading 0's ending with a single 1). For each row corresponding to an edge, in the leading bits we encode the value 0 (that is, all 0's). Now we describe the entries in the columns corresponding to the edges. For each pair of columns value, the left bit (corresponding to 2^1) for each row (i.e., except for the row corresponding to T) is 0. The right bit of the pair for an edge uv (corresponding to 2^0) is 1 for the rows corresponding to u and v, and 0 for all other vertex rows. The right bit of the pair is equal to 1 for the edge row corresponding to uv, but 0 for all other edge rows. Hence, for each column pair, there are exactly 3 rows with value 1, and the rest are 0. This ensures that no matter which rows we select, we will never "carry" across our double line borders.
Suppose that there is a vertex cover of size k. We can build a set S by including all of those rows corresponding to vertices in this cover, as well as each edge row corresponding to edges for which exactly one endpoint is contained in S. The leading bits add up to the assigned target since k nodes are in S, and for each column pair, we have selected exactly two rows with the entries 01 in it, and hence they add up correctly. It is easy to deduce this backwards (given the "no carry" across thick lines structure). If there are a subset of rows that add up to T, the vertex rows selected must exactly correspond to a vertex cover of size k.

In contrast to the knapsack problem, the following problem is NP-hard even if the numbers we are dealing with are small:

In the *Scheduling Problem*, n is the number of jobs, m is the number of machines. Each job is processed by exactly one of the machines. If job j is processed by machine i, it takes p_{ij} time units. The goal is to minimize the maximum load assigned to any one machine. This problem is NP-complete and hard to solve even if $p_{ij} \in \{0, 1, 2, 3\}$, so its hardness is not due to the magnitude of the numbers and number of bits used to encode these numbers.

We can write the following IP for our scheduling problem:

$$x_{ij} = \begin{cases} 1, & \text{if job j is processed by machine i;} \\ 0, & \text{otherwise.} \end{cases}$$

$$T = max\ load\ value$$

$$\begin{array}{lll} & \min\ T & \\ s.t. & \sum_{i=1}^{m} x_{ij} = 1 & j = 1, ..., n \\ & \sum_{j=1}^{n} p_{ij} x_{ij} \leq T & i = 1, ..., m \\ & x_{ij} \geq 0,\ integer & \end{array}$$

The LP relaxation for this integer program is not good at all, for a very simple reason. We can have fractional use of machines. Imagine we have just 1 job, 1000 machines and $p_{ij} = M$. Now, setting $x_{i1} = 1/m$, we obtain $LP_{opt} = M/m$ but $IP_{opt} = M$. There is a huge difference.

Adding the following constraint to our constraint set improves our model and narrows the gap between the LP and IP optima:

$$\sum_{i=1}^{m} p_{ij} x_{ij} \leq T \qquad j = 1, ..., n$$

Theorem: For any input, $\frac{IP_{opt}}{LP_{opt}} \leq 4$

However, we can improve our LP even more. The following formulation is much stronger; suppose we have a target value T, and want to decide if there exists a feasible schedule of value T:

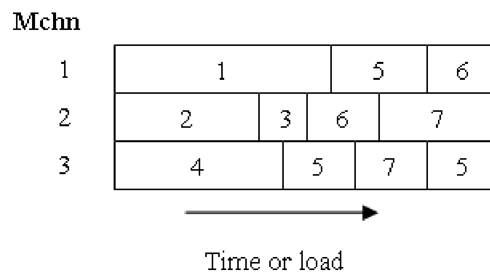
$$\begin{array}{ll} \sum_{i=1}^{m} x_{ij} = 1 & j = 1, ..., n \\ \sum_{j=1}^{n} p_{ij} x_{ij} \leq T & i = 1, ..., m \\ x_{ij} = 0 \quad if \quad p_{ij} > T & \\ x_{ij} \geq 0 & \forall m, n \end{array}$$

Theorem: If the above LP is feasible, then there exists an integer solution of max load at most equal to $2T$.

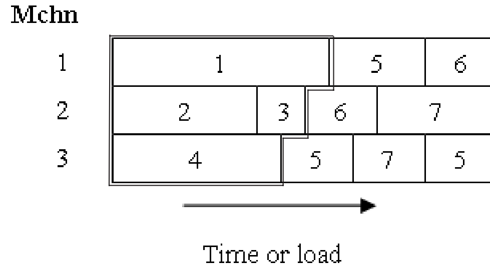
Proof: Whenever we have feasible solutions for an LP, we have extreme points. Let $\bar{x}$ be a basic feasible solution to this LP. Out of these $m + n + mn$ constraints, we will choose mn to hold with equality.
First observation $\Rightarrow \quad \leq m + n$ coordinates that are non-zero in $\bar{x}$.

Definition: A *split job* is one for which larger than 1 coordinate of $x_{ij} > 0$, i.e. the job is being processed on more than 1 machine.

Lemma: There are at most m split jobs in $\bar{x}$.
(At most $m + n$ non zero variables, a different non-zero must be "used up" for each of n jobs, leaving $\leq m$ non-zero left over to split jobs.



We start out with the solution given in the figure above. As it is shown, the jobs 5,6, and 7 are split. One thing we can do is to use the solution below, which uses unsplit jobs $\{1, 2, 3, 4\}$.

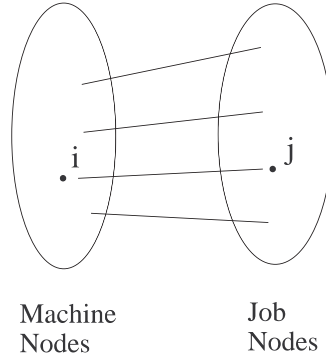


Therefore we use the solution $\bar{x}$ if it does not split the jobs.

Hope: Suppose we could pair up the split jobs 1-1 with machines, where each job is assigned to a machine on which some positive part was assigned by $\bar{x}$. What would this imply?
Note that we have the constraint $x_{ij} = 0 \quad if \quad p_{ij} > T$. Since j is assigned to i s.t. $x_{ij} \neq 0$, we have $p_{ij} \leq T$.
$\Rightarrow \leq T$ *additional work/machine.*
$\Rightarrow \leq 2T$ *units of work/machine.*

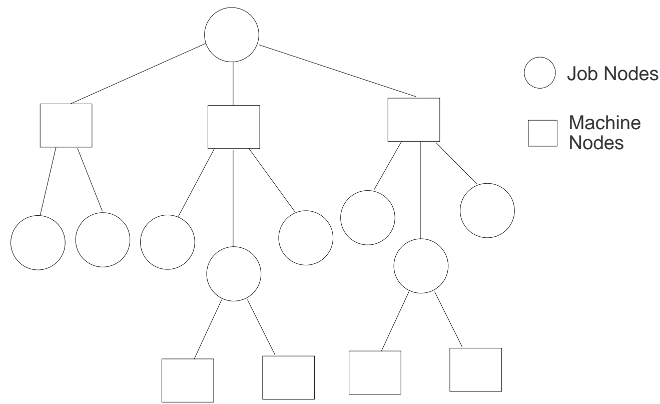
Indeed, our hope is a lemma. The following provides sufficient structure to yield this claim.

Lemma: If we construct the graph of non-zero coordinates for a basic feasible solution, each component is either a tree, or a tree plus one edge.

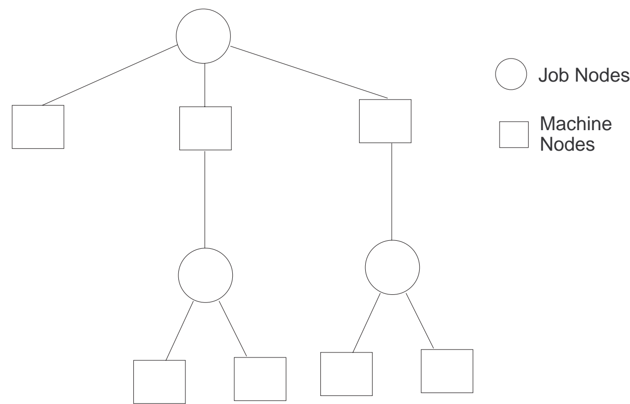


$\bar{x}_{ij} > 0$

We need to show then that, each component we can match up the split jobs in that component with the machines in that component. Suppose that the component is a tree (see the figure below).

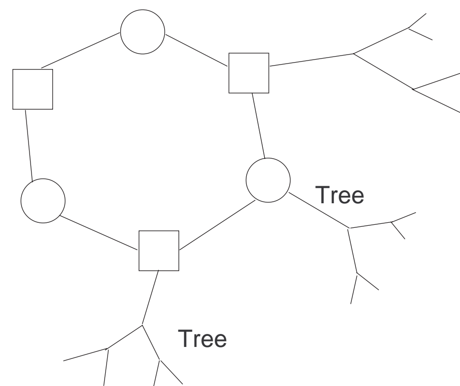


Note that the unsplit jobs must be leaves of this tree (i.e., nodes of degree 1) and that since they are already scheduled, we can delete them. Now observe that each job node is not a leaf in the tree remaining. If we pick one of the nodes of the tree (arbitrarily), hold the tree by this node, and dangle it, we get the following figure:



Every job node has a child (since it is not a leaf), which is a machine. Assign each job node to one of its children (if more than one, choose arbitraily).

Now assume that the graph is a tree plus one additional edge. This implies that we have a cycle, has the following structure:



By choosing alternate edges of this cycle, we can pair up its jobs with machines. What remains are trees, that can be dealt with as in the previous case. The "hope" has been proved, and we obtain the theorem.