ORIE 6300 Mathematical Programming I November 13, 2008

Lecture 21

Lecturer: Robert Kleinberg *Scribe: Shanshan Zhang*

1 NP-Completeness (continued)

Theorem 1 *If* Π *is NP-complete and* $\Pi \in P$*, then* $P = NP$*.*

Proof: Let $\Pi' \in NP$. We want to show $\Pi' \in P$. Since Π is NP-complete, we have $\Pi' \prec \Pi$. In other words, there is a function f computable in time $p_1(n)$ such that $x \in \Pi'$ iff $f(x) \in \Pi$. Since $\Pi \in P$, there is an algorithm running in time $P_2(n)$ that decides Π. To see if $x \in \Pi'$, compute $f(x)$ then decide if $f(x) \in \Pi$. Therefore, the total running time is $P_1(n) + P_2(P_1(n)) = \text{poly}(n)$. Thus, if Π is NP-Complete and $\Pi \in P$, then $P = NP$. □

Theorem 2 *If* $\Pi_0 \prec \Pi_1 \prec \Pi_2$*, then* $\Pi_0 \prec \Pi_2$*.*

Proof: Suppose f, g are reductions from Π_0 to Π_1, Π_1 to Π_2, respectively. Then, the following statements are satisfied:

1. runtime(f)=$P_f(n)$;
2. runtime(g)=$P_g(n)$;
3. $f(x) \in \Pi_1$ iff $x \in \Pi_0$;
4. $g(y) \in \Pi_2$ iff $y \in \Pi_1$.

Thus, the reduction from Π_0 to Π_2 is $g \circ f$. $x \in \Pi_0$ iff $f(x) \in \Pi_1$ iff $g(f(x)) \in \Pi_2$. The runtime is: $P_f(n) + P_g(P_f(n)) = \text{poly}(n)$. Therefore, if $\Pi_0 \prec \Pi_1 \prec \Pi_2$, then $\Pi_0 \prec \Pi_2$. □

Corollary 3 *If* Π *is NP-complete and* $\Pi \prec \Pi'$ *and* $\Pi' \in NP$*, then* Π' *is NP-complete.*

Proof: Given $\Pi_0 \in$ NP, $\Pi_0 \prec \Pi \prec \Pi'$. According to the above theorem, $\Pi_0 \prec \Pi'$. Thus, Π' is NP-complete. □

We use the following steps to prove Π is NP-Complete.

1. Show $\Pi \in NP$;
2. Choose an NP-complete Π';
3. Design the reduction from Π' to Π;
4. Prove the reduction runs in polynomial time;
5. Prove that $x \in \Pi'$ iff $f(x) \in \Pi$;

In common cases, the step (2) and (3) are hard, while other steps are relatively easy. To get started with proving problems are NP-complete, we first need some NP-complete problem to start with.

Theorem 4 (Cook-Levin Theorem) *SAT is NP-Complete.*

We skip the proof here. We just give several examples using this result to show some problems are NP-complete.

Example 1: SAT $\prec$ 0-1 Integer Programming, thus 0-1 IP is NP-Complete.

Example 2: Define 3-SAT as SAT restricted to instances with less than or equal to 3 variables per clause. Then we have SAT $\prec$ 3-SAT. We show that by splitting a clause. Take a clause C of size $K \geq 4$. Rewrite it as $C_1 \bigwedge C_2$ of sizes $K = 1, 2, 3$. For example, let $C = (x_1 \bigvee \bar{x}_2 \bigvee \bar{x}_3 \bigvee x_4 \bigvee \bar{x}_5)$. We rewrite it as $C_1 \bigwedge C2$, where $C_1 = (x_1 \bigvee \bar{x}_2 \bigvee \bar{x}_3 \bigvee z)$ and $C_2 = (\bar{Z} \bigvee x_4 \bigvee x_5)$. In every boolean assignment satisfying $C_1 \bigwedge C_2$, C is also satisfied. If a boolean assignment $\vec{X}$satisfies C then either $(\vec{X}, T)$ or $(\vec{X}, F)$ satisfies $C_1 \bigwedge C_2$. We use the following algorithm to get the reduction from SAT to 3-SAT.

Inputs: formula Φ
  while (Φ contains clause C of size $K \geq 4$)
    split C to $C_1 \bigwedge C_2$
    replace C in Φ by C_1 and C_2
  endwhile.

So why the process is completed in a polynomial time? Think about it offline.

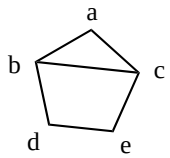
Next we will talk about NP-complete graph theoretic problems. First up, we will give some basic definitions. Suppose $G = (V, E)$ is an undirected graph.

Definition 1 *A clique in G is a subset $S \subseteq V$ such that every pair $u, v \in S$ are jointed by an edge in G.*

Definition 2 *An independent set is $S \subseteq V$ such that every pair $u, v \in S$ is not joined by edge in G.*

Definition 3 *A vertex cover is a $S \subseteq V$ such that every edge has at least one endpoint in S.*

For example, for the following graph, $\{a, b, c\}$ is a clique; $\{b, e\}$ is an independent set; $\{a, b, e\}$ is a vertex cover.



Given a graph G and $k \in \mathbb{N}$, are following questions NP-complete?

1. G has a clique whose size is greater than or equal to k;
2. G has an independent set whose size is greater than or equal to k;
3. G has a vertex cover whose size is less than or equal to k.

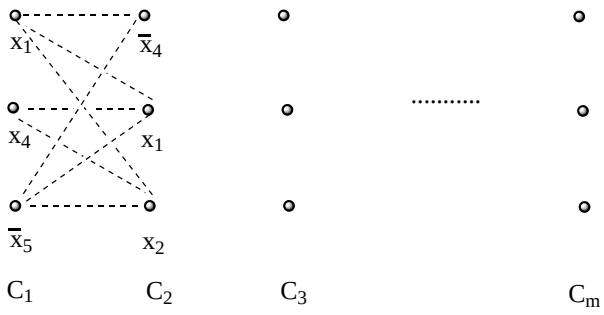
Actually, they are all NP-complete. We will prove them one by one.

Theorem 5 ***Clique*** *is NP-complete.*

Proof: An input to the Clique problem is a pair (G, k), where G is a graph and k is a number, and we want to decide if $\exists$ a complete subgraph (a subset of nodes such that each pair of nodes have an edge between them) of G with k nodes.

We can prove that Clique is an NP-complete problem by proving that 3-SAT $\prec$ Clique.

The idea is the following: Given an input $(l_{1_1} \vee l_{1_2} \vee l_{1_3}) \wedge \cdots \wedge (l_{m_1} \vee l_{m_2} \vee l_{m_3})$ to the 3-SAT problem, we can set $k = m$, and create the following graph, where we have exactly one node for each literal in each clause, and we only connect nodes that come from different clauses, provided that the first node is not the negation of the second.



Now, we prove that the 3-SAT formula is satisfiable if and only if there is a clique of size k in the graph. Suppose that the 3-SAT formula is satisfiable. Focus on one satisfying assignment for the formula, and for each clause, choose one of the literals that are made "true" by it. These literals correspond to nodes in the graph that form a clique of size $m = k$. (If there isn't an edge between a pair of nodes, then the two endpoints must correspond to a literal and its negation, both of which could not have been made 'true" in the assignment.) On the other hand, suppose that there is a clique of size k. For each clause, there is no edge between any pair of the nodes corresponding to it, so from each triple of nodes, there can be at most one node in the clique. But there are only $m = k$ such triples, and so, in fact, the clique must have, for each clause, exactly one node corresponding to a literal in that clause. But now, we see that we can make a consistent assignment to those literals (because there is an edge between each pair of them) so that they are all true. And this yields an assignment that makes the 3-SAT formula "true". □

Theorem 6 ***Clique*** $\prec$ ***Independent Set.***

Proof: S is a clique of size k in G iff S is an independent set of size k in $\bar{G}$. □

Theorem 7 ***Independent Set*** $\prec$ ***Vertex Cover.***

Proof: S is an independent set iff $\bar{S}$ is a vertex cover. □

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 $O(nW^2)$ time (Recall that the number of bits required to encode W is $\log_2 W$, therefore $O(nW^2)$ is not a polynomial running time.)

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 contrast, 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.

Theorem 8 *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.:

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

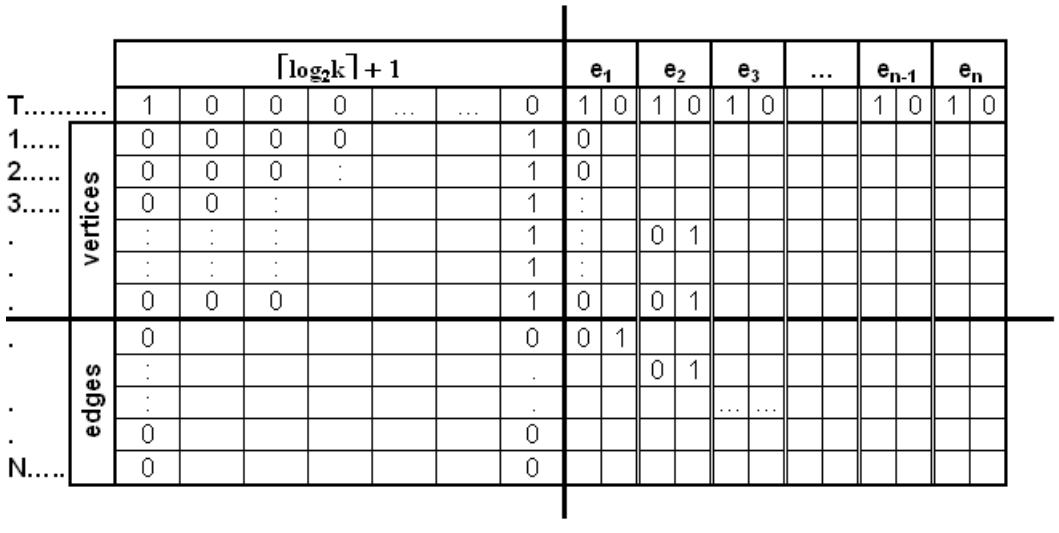
Definition 4 *The* Subsetsum Problem *can be defined as follows: Given an input* $\{a_1, a_2, ...a_N, T\}$, *does there exist* $S \subseteq \{1, ..., n\}$ *s.t.* $\sum_{i \in S} a_i = T$ *?*

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

As for the vertex cover problem, 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:



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. □