CS787: Advanced Algorithms

Scribe: Shijin Kong and David Malec | **Lecturer:** Shuchi Chawla
Topic: NP-Completeness, Approximation Algorithms | **Date:** 10/1/2007

As we've already seen in the preceding lecture, two important classes of problems we can consider are P and NP. One very prominent open question is whether or not these two classes of problems are in fact equal. One tool that proves useful when considering this question is the concept of a problem being complete for a class. In the first two portions of this lecture, we show two problems to be complete for the class NP, and demonstrate two techniques for proving completeness of a problem for a class. We then go on to consider approximation algorithms, which have as a goal providing efficient near-optimal solutions to problems for which no known efficient algorithm for finding an optimal solution exists.

9.1 Cook-Levin Theorem

The Cook-Levin Theorem gives a proof that the problem SAT is NP-Complete, via the technique of showing that any problem in NP may be reduced to it. Before proceeding to the theorem itself, we revisit some basic definitions relating to NP-Completeness.

First, we need to understand what problems belong to the classes P and NP. If we restrict ourselves to decision problems, every problem has a set of accepted inputs; we call this set of inputs a language, and can think of the original problem as reducing to the question of whether or not a given input is in this language. Then, P is the class of languages for which membership can be decided in polynomial time, and NP is the class of languages such that $x \in L$ if and only if there's a certificate of its membership that can be verified in polynomial time.

Second, we need the notion of reducibility of one problem to another. We say that a that a problem B can be reduced to A, or $B \leqslant_p A$ if, given access to a solution for A, we can solve B in polynomial time using polynomially many calls to this solution for A. This idea of reducibility is crucial to completeness of a problem for a class. If a problem $L \in NP$ satifies that for any $L' \in NP$, $L' \leqslant_p L$, then we say L is NP-Complete. It is worth noting that for proving NP-Completeness, typically a more restrictive form of reduction is used, namely a Karp reduction. It follows the same basic structure, but requires that only one query be made to the solution for the problem being reduced to, and that the solution be directly computable from the result of the query.

The Cook-Levin Theorem shows that SAT is NP-Complete, by showing that a reduction exists to SAT for any problem in NP. Before proving the theorem, we give a formal definition of the SAT problem (Satisfiability Problem):

Given a boolean formula ϕ in CNF (Conjuctive Normal Form), is the formula satisfiable? In other words, we are given some boolean formula ϕ with n boolean variables $x_1, x_2, \ldots, x_n$, $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$, where each of the C_i is a clause of the form $(t_{i1} \vee t_{i2} \vee \ldots \vee t_{il})$, with each l_{ij} is a literal drawn from the set $\{x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$. We need to decided whether or not the exists some setting of the $x_1, x_2, \ldots, x_n$ that cause the formula ϕ to be satisfied (take on a boolean

value of true).

Theorem 9.1.1 *Cook-Levin Theorem: SAT is NP-complete.*

Proof: Suppose L is a NP problem; then L has a polynomial time verifier V:

1. If $x \in L$, $\exists$ witness y, $V(x, y) = 1$
2. If $x \notin L$, $\forall$ witness y, $V(x, y) = 0$

We can build a circuit with polynomial size for the verifier V, since the verifier runs in polynomial time (note that this fact is nontrivial; however, it is left to the reader to verify that it is true). The circuit contains AND, OR and NOT gates. The circuit has $|x| + |y|$ sources, where $|x|$ of them are hardcoded to the values of the bits in x and the rest $|y|$ are variables.

Now to solve problem L, we only need to find a setting of $|y|$ variables in the input which causes the circuit to output a 1. That means problem the problem L has been reduced to determining whether or not the circuit can be caused to output a 1. The following shows how the circuit satisfication problem can be reduced to an instance SAT.

Each gate in the circuit can be represented by a 3CNF (each clause has exactly three terms). For example:

1. The functionality of an OR gate with input a, b and output Z_i is represented as:

   $(a \vee b \vee \bar{Z}_i) \wedge (Z_i \vee \bar{a}) \wedge (Z_i \vee \bar{b})$

2. The functionality of a NOT gate with input a and output Z_i is represented as:

   $(a \vee Z_i) \wedge (\bar{a} \vee \bar{Z}_i)$

   Notice although some of the clauses have fewer than 3 terms, it is quite easy to pad them with independant literals to form clauses in 3CNF. The value of an independent literal won't affect the overall boolean value of the clauses. We essentially include clauses for it taking the values 0 or 1; regardless of the actual value, our clauses have the same value as the simpler two-term clause.

Suppose we have q gates in V marked as $Z_1, Z_2, \ldots, Z_q$ with Z_q representing the final output of V. Each of them either takes some of the sources or some intermediate output Z_i as input. Therefore the whole circuit can be represented as a formula in CNF:

$\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_q \wedge Z_q$

where

$C_i = (t_1 \vee t_2 \vee t_3), t_1, t_2, t_3 \in (x, y, Z_1, Z_2, \ldots\ Z_q, \bar{Z}_1, \bar{Z}_2, \ldots, \bar{Z}_q)$

As we said before, even if the last clause of ϕ has only one term Z_q, we may extend ϕ to an equivalent formula in 3CNF by adding independent variables. Thus, we have shown that the circuit can be reduced to ϕ, a formula in 3CNF which is satisfiable if and only if the original circuit could be made to output a value of 1. Hence, $L \leqslant_p$ SAT. Since SAT can be trivially shown to be in NP (any

satisfying assignmant may be used as a certificate of membership), we may conclude that SAT is NP-Complete. ■

Furthermore, it should be noted that throughout the proof, we restricted ourselves to formulas that were in 3CNF. In general, if we restrict the formulas under consideration to be in k-CNF for some $k \in \mathbb{Z}$, we get a subproblem of SAT which we may refer to as k-SAT. Since the above proof ensured that ϕ was in 3CNF, it actually demonstrated not only that SAT is NP-Complete, but that 3-SAT is NP-Complete as well. In fact, one may prove in general that k-SAT is NP-Complete for $k \geq 3$; for $k = 2$, however, this problem is known to be in P.

9.2 Vertex Cover

One example of an NP-Complete problem is the vertex cover problem. Given a graph, it essentially asks for a subset of the vertices which cover the edges of the graph. It can be phrased both as a search problem and as a decision problem. Formally stated, the search version of the vertex cover problem is as follows:

Given: A graph $G = (V, E)$

Goal: A subset $V' \subseteq V$ such that for all $(u, v) \in E$, $u \in V'$ or $v \in V'$, where $|V'|$ is minimized.

For a concrete example of this problem, consider the following graph:

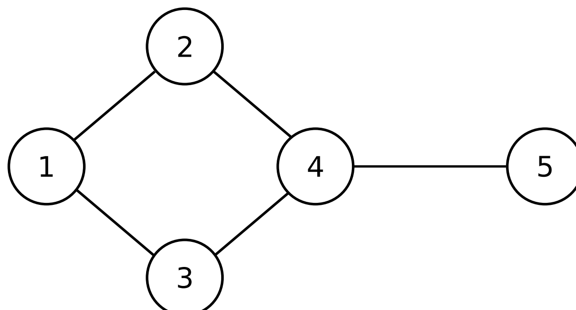

Figure 9.2.1: The nodes 1 and 4 form a minimal vertext cover

The decision variant of this problem makes only two changes to its specification: a positive integer k is added to the givens; and rather than searching for a V' of minimal size, the goal is to decide whether a V' exists with $|V'| = k$.

Furthermore, we can reduce the search version of this problem to the decision version. We know that the size of a minimal vertex cover must be between 0 and V, so if we are given access to a solution for the decision version we can just do a binary search over this range to find the exact value for the minimum possible size for a vertex cover. Once we know this, we can just check vertices one by one to see if they're in a minimal vertex cover. Let k be the size we found for a minimal vertex cover. For each vertex, remove it (and its incident edges) from the graph, and check whether it is possible to find a vertex cover of size $k - 1$ in the resultant graph. If so, the removed vertex is in a minimal cover, so reduce k by 1 and continue the process on the resulting graph. If

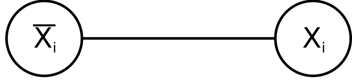

Figure 9.2.2: The gadget X_i

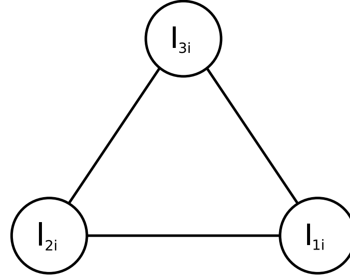

Figure 9.2.3: The gadget C_i

not, add the vertex back into the graph, and try the next one. When k reaches 0, we have our vertex cover. This property of being able to solve a search problem via queries to an oracle for its decision variant is called self-reducibility.

Theorem 9.2.1 *The decision version of Vertex Cover is NP-Complete.*

Proof: It is immediate that Vertex Cover $\in$ NP. We can verify a proposed vertex cover of the appropriate size in polynomial time, and so can use these as certificates of membership in the language.

All we need to show then is that VC is NP-Hard. We do so by reducing 3-SAT to it. Given some formula ϕ in 3-CNF, with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$, we will build a graph, starting with the following gadgets.

For each variable x_i, we will build a gadget X_i. X_i will have two vertices, one corresponding to each of x_i and $\bar{x_i}$, and these two vertices will be connected by an edge (see figure 9.2.2). Notice that by construction, any vertex cover must pick one of these vertices because of the edge between them.

For each clause c_i, we will build a gadget C_i. Since ϕ is in 3-CNF, we may assume that $c_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ for some set of literals l_{i1}, l_{i2}, l_{i3}. The gadget C_i will consist of a complete graph of size 3, with a vertex corresponding to each of the literals in the clause c_i (see figure 9.2.3). Notice that since all of the vertices are connected by edges, any vertex cover must include at least two of them.

Given the preceding gadgets for each of the variables x_i and each of the clauses c_i, we describe how the gadgets are connected. Consider the gadget C_i, representing the clause $c_i = (l_{1i} \vee l_{2i} \vee l_{3i})$. For each of the literals in c_i, we add an edge from the corresponding vertex to the vertex for the appropriate boolean value of the literal. So if $l_{i1} = x_k$, we will add an edge from the vertex for l_{i1} in the clause gadget C_i to the vertex for x_k in the variable gadget X_k; similarly, if $l_{i1} = \bar{x_k}$, we add and edge to the vertex for $\bar{x_k}$ in X_k. Figure 9.2.4 gives the connections for an example where $c_i = (x_1 \vee \bar{x_2} \vee x_4)$.

Now that we have our graph, we consider what we can say about its minimum vertex cover. As previously noted, at least one vertex from any variable gadget must be included in a vertex cover; similarly, at least 2 vertices from any clause gadget must be included. Since we have n variables and m clauses, we can thus see that no vertex cover can have size strictly less than $n + 2m$.

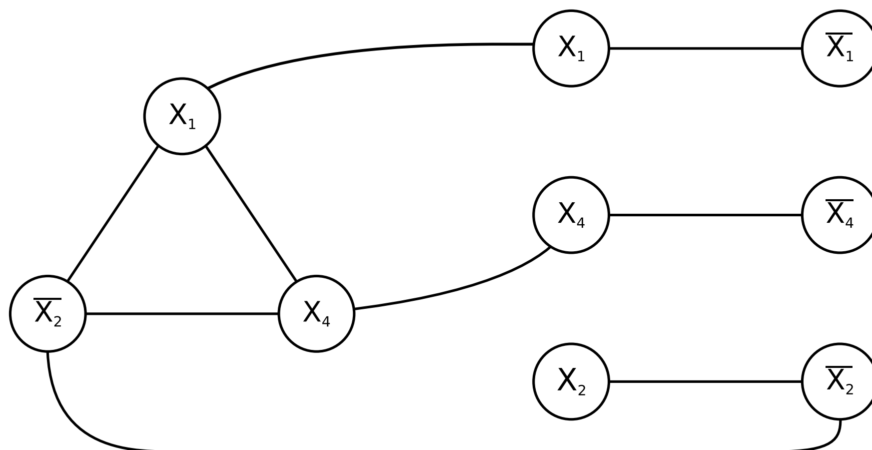

Figure 9.2.4: Connecting the gadgets

Furthermore, we show that a vertex cover of exactly this size exists if and only if the original formula ϕ is satisfiable.

Assume that such a vertex cover exists. Then, for each of the gadgets X_i, it contains exactly one vertex. If we set each x_i so that the node in the vertex cover corresponds to a boolean value of true, we will have a satisfying assignment. To see this, consider any clause c_i. Since the vertex cover is of size exactly $n + 2m$, only two of the vertices in the gadget C_i are included. Now, the vertex which wasn't selected is connected to the node corresponding to its associated literal. Thus, since we have a vertex cover, that node must be in it. So, the unselected literal must have a boolean value of true in our setting of the x_i, and so the clause is satisfied.

Similarly, if an assignment of the x_i exists which satisfies ϕ, we can produce a vertex cover of size $n+2m$. In each gadget X_i, we simply choose the vertex corresponding to a boolean value of true in our satisying assignment. Then, since it is a satisfying assignment, each clause contains at least one literal with a boolean value of true. If we leave the corresponding vertex, and include the others, we get a vertex cover. The edges inside the gadget C_i are satisfied, since we have chosen all but one of its vertices; the edges from it to variable gadgets are satisfied, since for each one, either the vertex in C_i has been included or the vertex in the corresponding X_i has been chosen.

Thus, we can see that the original formula is satisfiable if and only if the constructed graph has a vertex cover of size $n+2m$. Thus, since 3SAT is NP-Hard, we get that VC is NP-Hard as well. As we have previously noted, VC $\in$ NP, and hence VC is NP-Complete. ■

9.3 Approximation Algorithms

In this section we consider another topic: approximation algorithms. There are many cases where no polynomial time algorithm is known for finding an optimal solution. In those situations, we can compromise by seeking a polynomial time near-optimal solution. However, we want to guarantee in this case that this alternative is close to optimal in some well-defined sense. To demonstrate

that a solution generated by an approximation algorithm is close to optimal, we need to compare its solution to the optimal one. One way of formalizing the idea of "close enough" is that of α-approximations which may be formalized with the following definitions.

For an NP optimization problem $A = (F,\ val)$, where given an instance $I \in A$:

1. $F(I)$ denotes the set of feasible solutions to this problem.
2. $val(x, I)$ denotes the value of solution $x \in F(I)$.

Our purpose is to find the optimal solution $OPT(I)$:

$$OPT(I) = \min_{x \in F(I)} val(x, I) \tag{9.3.1}$$

Suppose y^* is the optimal solution for instance I, then an α-approximation y to the optimal solution must satisfy:

$$\frac{val(y, I)}{OPT(I)} \leqslant \alpha \tag{9.3.2}$$

if A is a minimization problem, and

$$\frac{OPT(I)}{val(y, I)} \leqslant \alpha \tag{9.3.3}$$

if A is a maximization problem.

(9.3.4)

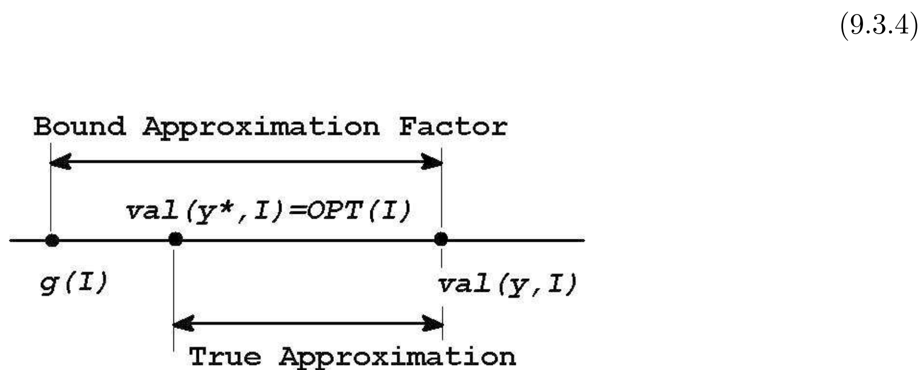

Figure 9.3.5: Lower bound for optimal solution

An optimal solution is usually computationally hard to find, so we can't directly judge whether a solution y is α-approximation or not; however, for some problems, it is possible to find a lower bound for the optimal solution $g(I)$ (Figure 9.3.5) which is good enough. Instead of computing the exact value of α, we obtain a reasonably upper bound for it using our lower bound on the optimal solution. If we can show $\frac{val(y,I)}{g(I)} \leqslant \alpha$, then we know y to be an α-approximation.

9.3.1 α-Approximation of Vertex Cover Problem

Now, we try to get an α-approximation for the vertex cover problem.

Lemma 9.3.1 *Given a graph G, the size of any matching in G is a lower bound on the size of any vertex cover.*

Proof: The key insight here is that the edges in a matching do not share any common vertices. So according to the definition of the vertex cover problem, any vertex cover problem must choose at least one vertex from any edge in the matching. Therefore, the size of any matching provides a lower bound for the size of any vertex cover. ■

Using a maximal matching with the above will give us a sufficiently good bound to derive an α-approximation.

Lemma 9.3.2 *Given a graph G and any maximal matching M on G, we can find a vertex cover for G of size at most $2|M|$.*

Proof: First, we prove that for any maximal matching M, there exists a solution for the vertex cover problem with size no more than $2|M|$. We start by choosing both vertices incident on each of the edges in M. This gives us a set of vertices of size $2|M|$. We claim this vertex set constitutes a valid vertex cover. Assume by way of contradiction that it does not; then we can find a edge e of which neither node is picked present in our set of vertices. But then, we may add e to our original matching M to form a larger matching M'. But by assumption M is a maximal, and so this is a contradiction. Thus, we have found a vertex cover for G of size $2|M|$ ■

Theorem 9.3.3 *There exists a polynomial time 2-approximation for vertex cover problem.*

Proof: Finding a maximum matching M can be done in polynomial time. Once we find M, we simply apply the strategy outlined above to get our vertex cover. Since we showed that the size of any matching provides a lower bound for the size of any vertex cover, we may conclude that the optimal vertex cover must be of size at least $|M|$. Thus, since our proposed vertex cover will have size $2|M|$, our algorithm will provide a 2-approximation for the vertex cover problem. ■